

1) write a query second height salary in emp table.

Ans- Consider below simple table:

Name	Salary
abc	100000
bcd	1000000
efg	40000
ghi	500000

How to find the employee whose salary is second highest. For example, in above table, "ghi" has the second highest salary as 500000.

Below is simple query to find the employee whose salary is highest.

```
SELECT name, MAX(salary) as salary FROM employee
```

We can nest the above query to find the second largest salary.

```
SELECT name, MAX(salary) AS salary  
FROM employee  
WHERE salary < (SELECT MAX(salary)  
                FROM employee);
```

There are other ways also as suggested by RajnishKrJha.

```
SELECT name, MAX(salary) AS salary  
FROM employee  
WHERE salary IN  
(SELECT salary FROM employee MINUS SELECT MAX(salary)  
FROM employee);  
  
SELECT name, MAX(salary) AS salary  
FROM employee  
WHERE salary < (SELECT MAX(salary)  
                FROM employee);
```

One way as suggested by **Arka Poddar**.

IN SQL Server using Common Table Expression or **CTE**, we can find the second highest salary:

```

WITH T AS
(
SELECT *
    DENSE_RANK() OVER (ORDER BY Salary Desc) AS Rnk
FROM Employees
)
SELECT Name
FROM T
WHERE Rnk=2;

```

How to find the third largest salary?

Simple, we can do one more nesting.

```

SELECT name, MAX(salary) AS salary
    FROM employee
    WHERE salary < (SELECT MAX(salary)
        FROM employee
        WHERE salary < (SELECT MAX(salary)
            FROM employee))
    );

```

Note that instead of nesting for second, third largest salary, we can find nth salary using general query like in mysql:

```
select salary from employee order by salary desc limit n-1,1;
```

Or

```
Select name,salary from employee A where n-1 = (Select count(1) from
employee B where B.salary>A.salary)
```

2) write a query Highest Salary in each department

Ans-select * from (select ename,deptno,sal,hiredate,dense_rank() over(partition by deptno order by sal desc,hiredate asc) rn from emp)
where rn=1;

3)write a program repeated charter in a string

Ans-

```

// Java program to find the first
// repeated character in a string
import java.util.*;

class Main
{
    // This function prints the first repeated
    // character in str[]
    static char firstRepeating(char str[])
    {
        // Creates an empty hashset
        HashSet<Character> h = new HashSet<>();

        // Traverse the input array from left to right
        for (int i=0; i<=str.length-1; i++)
        {
            char c = str[i];

            // If element is already in hash set, update x
            // and then break
            if (h.contains(c))
                return c;

            else // Else add element to hash set
                h.add(c);
        }

        return '\0';
    }

    // Driver method to test above method
    public static void main (String[] args)
    {
        String str = "geeksforgeeks";
        char[] arr = str.toCharArray();
        System.out.println(firstRepeating(arr));
    }
}

```

Output:

e

4) write a program first character occurrence in a string.

Ans-

```

class NoOfOccurrenceOfCharacters {
    static final int MAX_CHAR = 256;

    static void getOccuringChar(String str)
    {
        // Create an array of size 256 i.e. ASCII_SIZE
        int count[] = new int[MAX_CHAR];

```

```

        int len = str.length();

        // Initialize count array index
        for (int i = 0; i < len; i++)
            count[str.charAt(i)]++;

        // Create an array of given String size
        char ch[] = new char[str.length()];
        for (int i = 0; i < len; i++) {
            ch[i] = str.charAt(i);
            int find = 0;
            for (int j = 0; j <= i; j++) {

                // If any matches found
                if (str.charAt(i) == ch[j])
                    find++;
            }

            if (find == 1)
                System.out.println("Number of Occurrence of " +
                    str.charAt(i) + " is:" + count[str.charAt(i)]);
        }
    }
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        String str = "geeksforgeeks";
        getOccuringChar(str);
    }
}

```

Output:

```

Number of Occurrence of g is:2
Number of Occurrence of e is:4
Number of Occurrence of k is:2
Number of Occurrence of s is:2
Number of Occurrence of f is:1
Number of Occurrence of o is:1
Number of Occurrence of r is:1

```

5)difference between arraylist and linked list

Ans-ArrayList:-

- ArrayList uses a dynamic array.

- ArrayList is not efficient for manipulation because a lot of shifting is required.
- ArrayList is better to store and fetch data.

LinkedList:-

- LinkedList uses doubly linked list.
- LinkedList is efficient for manipulation.
- LinkedList is better to manipulate data
- ArrayList
- Manipulation with ArrayList is **slow** because it internally uses array. If any element is removed from the array, all the bits are shifted in memory.
- ArrayList is **better for storing and accessing** data.
- linked list
- Manipulation with LinkedList is **faster** than ArrayList because it uses doubly linked list so no bit shifting is required in memory.
- LinkedList is **better for manipulating** data.

Two popular lists in Java are:

1. **ArrayList**:-Implemented with the concept of dynamic array.

```
ArrayList<Type> arrL = new ArrayList<Type>();
```

Here Type is the data type of elements in ArrayList
to be created

2. **LinkedList**:-Implemented with the concept of doubly linked list.

```
LinkedList<Type> linkL = new LinkedList<Type>();
```

Here Type is the data type of elements in LinkedList
to be created

Comparision between **ArrayList** and **LinkedList**:-

1. Insertions are easy and fast in LinkedList as compared to ArrayList because there is no risk of resizing array and copying content to new array if array gets full which makes adding into ArrayList of $O(n)$ in worst case, while adding is $O(1)$ operation in LinkedList

in Java. ArrayList also needs to be update its index if you insert something anywhere except at the end of array.

2. Removal also better in LinkedList than ArrayList due to same reasons as insertion.
3. LinkedList has more memory overhead than ArrayList because in ArrayList each index only holds actual object (data) but in case of LinkedList each node holds both data and address of next and previous node.
4. Both LinkedList and ArrayList require O(n) time to find if an element is present or not. However we can do Binary Search on ArrayList if it is sorted and therefore can search in O(Log n) time.

filter_none

edit

play_arrow

brightness_4

```
// Java program to demonstrate difference between ArrayList and
// LinkedList.
import java.util.ArrayList;
import java.util.LinkedList;

public class ArrayListLinkedListExample
{
    public static void main(String[] args)
    {
        ArrayList<String> arrlistobj = new ArrayList<String>();
        arrlistobj.add("0. Practice.GeeksforGeeks.org");
        arrlistobj.add("1. Quiz.GeeksforGeeks.org");
        arrlistobj.add("2. Code.GeeksforGeeks.org");
        arrlistobj.remove(1); // Remove value at index 2
        System.out.println("ArrayList object output :" + arrlistobj);

        // Checking if an element is present.
        if (arrlistobj.contains("2. Code.GeeksforGeeks.org"))
            System.out.println("Found");
        else
            System.out.println("Not found");

        LinkedList llobj = new LinkedList();
        llobj.add("0. Practice.GeeksforGeeks.org");
        llobj.add("1. Quiz.GeeksforGeeks.org");
        llobj.add("2. Code.GeeksforGeeks.org");
        llobj.remove("1. Quiz.GeeksforGeeks.org");
        System.out.println("LinkedList object output :" + llobj);

        // Checking if an element is present.
        if (llobj.contains("2. Code.GeeksforGeeks.org"))
            System.out.println("Found");
    }
}
```

```

        else
            System.out.println("Not found");
    }
}

```

Output:

```
ArrayList object output :[0. Practice.GeeksforGeeks.org, 2.
Code.GeeksforGeeks.org]
```

```
Found
```

```
LinkedList object output :[0. Practice.GeeksforGeeks.org, 2.
Code.GeeksforGeeks.org]
```

```
Found
```

6)difference between comparable and comparator

Ans-Comparable interface: Class whose objects to be sorted must implement this interface.In this,we have to implement compareTo(Object) method.

For example:

```

1  public class Country implements Comparable{
2      @Override
3      public int compareTo(Object arg0) {
4          Country country=(Country) arg0;
5          return (this.countryId < country.countryId ) ? -1: (this.countryId > country.co
6      }

```

If any class implements comparable interface then collection of that object can be sorted automatically using Collection.sort() or Arrays.sort().Object will be sort on the basis of compareTo method in that class.

Objects which implement Comparable in java can be used as keys in a SortedMap like TreeMap or SortedSet like TreeSet without implementing any other interface.

Comparator interface: Class whose objects to be sorted do not need to implement this interface.Some third class can implement this interface to sort.e.g.CountrySortByIdComparator class can implement Comparator interface to sort collection of country object by id. For example:

```

1  public class CountrySortByIdComparator implements Comparator<Country>{
2
3      @Override
4      public int compare(Country country1, Country country2) {
5
6          return (country1.getCountryId() < country2.getCountryId() ) ? -1: (country1.ge
7      }
8
9  }

```

Using Comparator interface,we can write different sorting based on different attributes of objects to be sorted.You can use anonymous comparator to compare at particular line of code. For example:

```
01           Country indiaCountry=new Country(1, 'India');
```

```

02     Country indiaCountry=new Country(1, 'India');
03     Country chinaCountry=new Country(4, 'China');
04     Country nepalCountry=new Country(3, 'Nepal');
05     Country bhutanCountry=new Country(2, 'Bhutan');
06
07     List<Country> listOfCountries = new ArrayList<Country>();
08     listOfCountries.add(indiaCountry);
09     listOfCountries.add(chinaCountry);
10    listOfCountries.add(nepalCountry);
11    listOfCountries.add(bhutanCountry);
12
13    //Sort by countryName
14
15    Collections.sort(listOfCountries, new Comparator<Country>() {
16
17        @Override
18        public int compare(Country o1, Country o2) {
19
20            return o1.getCountryName().compareTo(o2.getCountryName());
21        }
22    });

```

Comparator vs Comparable

Parameter	Comparable	Comparator
Sorting logic	Sorting logic must be in same class whose objects are being sorted. Hence this is called natural ordering of objects	Sorting logic is in separate class. Hence we can write different sorting based on different attributes of objects to be sorted. E.g. Sorting using id, name etc.
Implementation	Class whose objects to be sorted must implement this interface.e.g Country class needs to implement comparable to collection of country object by id	Class whose objects to be sorted do not need to implement this interface. Some other class can implement this interface. E.g.- CountrySortByIdComparator class can implement Comparator interface to sort collection of country object by id
Sorting method	int compareTo(Object o1) This method compares this object with o1 object and returns a integer. Its value has following meaning 1. positive – this object is greater than o1 2. zero – this object equals to o1 3. negative – this object is less than o1	int compare(Object o1, Object o2) This method compares o1 and o2 objects. and returns a integer. Its value has following meaning. 1. positive – o1 is greater than o2 2. zero – o1 equals to o2 3. negative – o1 is less than o1
Calling method	Collections.sort(List) Here objects will be sorted on the basis of CompareTo method	Collections.sort(List, Comparator) Here objects will be sorted on the basis of Compare method in Comparator
Package		Java.lang.Comparable Java.util.Comparator

Java code:

For Comparable: We will create class country having attribute id and name. This class will implement Comparable interface and implement CompareTo method to sort collection of country object by id.

1. Country.java

```
01
02
03
04     package org.arpit.javapostsforlearning;
05     //If this.countryId < country.countryId:then compare method will return -1
06     //If this.countryId > country.countryId:then compare method will return 1
07     //If this.countryId==country.countryId:then compare method will return 0
08     public class Country implements Comparable{
09         int countryId;
10         String countryName;
11
12         public Country(int countryId, String countryName) {
13             super();
14             this.countryId = countryId;
15             this.countryName = countryName;
16         }
17
18         @Override
19         public int compareTo(Object arg0) {
20             Country country=(Country) arg0;
21             return (this.countryId < country.countryId ) ? -1: (this.countryId > country.
22             }
23
24         public int getCountryId() {
25             return countryId;
26         }
27
28         public void setCountryId(int countryId) {
29             this.countryId = countryId;
30         }
31
32         public String getCountryName() {
33             return countryName;
34         }
35
36
37 }
```

2.ComparatorMain.java

```
01     package org.arpit.javapostsforlearning;
02
03     import java.util.ArrayList;
04     import java.util.Collections;
05     import java.util.List;
06
07     public class ComparatorMain {
08
09         /**
```

```

08     * @author Arpit Mandliya
09     */
10    public static void main(String[] args) {
11        Country indiaCountry=new Country(1, 'India');
12        Country chinaCountry=new Country(4, 'China');
13        Country nepalCountry=new Country(3, 'Nepal');
14        Country bhutanCountry=new Country(2, 'Bhutan');
15
16        List<Country> listOfCountries = new ArrayList<Country>();
17        listOfCountries.add(indiaCountry);
18        listOfCountries.add(chinaCountry);
19        listOfCountries.add(nepalCountry);
20        listOfCountries.add(bhutanCountry);
21
22        System.out.println('Before Sort : ');
23        for (int i = 0; i < listOfCountries.size(); i++) {
24            Country country=(Country) listOfCountries.get(i);
25            System.out.println('Country Id: '+country.getCountryId()+'||'+Country
26
27        }
28        Collections.sort(listOfCountries);
29
30    }
31
32
33
34
35
36
37
38

```

Output:

```

01
02 Before Sort :
03 Country Id: 1||Country name: India
04 Country Id: 4||Country name: China
05 Country Id: 3||Country name: Nepal
06 Country Id: 2||Country name: Bhutan
07 After Sort :
08 Country Id: 1|| Country name: India
09 Country Id: 2|| Country name: Bhutan
10 Country Id: 3|| Country name: Nepal
11 Country Id: 4|| Country name: China

```

For Comparator: We will create class country having attribute id and name and will create another class CountrySortByIdComparator which will implement Comparator interface and implement compare method to sort collection of country object by id and we will also see how to use anonymous comparator.

1.Country.java

```
01  
02  
03 package org.arpit.javapostsforlearning;  
04  
05 public class Country{  
06     int countryId;  
07     String countryName;  
08  
09     public Country(int countryId, String countryName) {  
10         super();  
11         this.countryId = countryId;  
12         this.countryName = countryName;  
13     }  
14  
15     public int getCountryId() {  
16         return countryId;  
17     }  
18  
19     public void setCountryId(int countryId) {  
20         this.countryId = countryId;  
21     }  
22  
23     public String getCountryName() {  
24         return countryName;  
25     }  
26  
27     public void setCountryName(String countryName) {  
28         this.countryName = countryName;  
29     }  
30 }
```

2.CountrySortbyIdComparator.java

```
01  
02 package org.arpit.javapostsforlearning;  
03  
04 import java.util.Comparator;  
05 //If country1.getCountryId()<country2.getCountryId():then compare method will return  
06 //If country1.getCountryId()>country2.getCountryId():then compare method will return  
07 //If country1.getCountryId()==country2.getCountryId():then compare method will return  
08 public class CountrySortByIdComparator implements Comparator<Country>{  
09  
10     @Override  
11     public int compare(Country country1, Country country2) {  
12         return (country1.getCountryId() < country2.getCountryId() ) ? -1: (country1.g  
13     }  
14 }  
15 }
```

3.ComparatorMain.java

```
01  package org.arpit.javapostsforlearning;
02
03  import java.util.ArrayList;
04  import java.util.Collections;
05  import java.util.Comparator;
06  import java.util.List;
07
08  public class ComparatorMain {
09
10      /**
11      * @author Arpit Mandliya
12      */
13
14      public static void main(String[] args) {
15          Country indiaCountry=new Country(1, 'India');
16          Country chinaCountry=new Country(4, 'China');
17          Country nepalCountry=new Country(3, 'Nepal');
18          Country bhutanCountry=new Country(2, 'Bhutan');
19
20
21          List<Country> listOfCountries = new ArrayList<Country>();
22          listOfCountries.add(indiaCountry);
23          listOfCountries.add(chinaCountry);
24          listOfCountries.add(nepalCountry);
25          listOfCountries.add(bhutanCountry);
26
27          System.out.println('Before Sort by id : ');
28          for (int i = 0; i < listOfCountries.size(); i++) {
29              Country country=(Country) listOfCountries.get(i);
30              System.out.println('Country Id: '+country.getCountryId()+' || '+Country
31          }
32          Collections.sort(listOfCountries,new CountrySortByIdComparator());
33
34
35          //Sort by countryName
36          Collections.sort(listOfCountries,new Comparator<Country>() {
37
38              @Override
39              public int compare(Country o1, Country o2) {
40                  return o1.getCountryName().compareTo(o2.getCountryName());
41              }
42          });
43
44
45      }
```

```

46    }
47
48
49
50
51
52
53
54
Output:
01
02 Before Sort by id :
03 Country Id: 1||Country name: India
04 Country Id: 4||Country name: China
05 Country Id: 3||Country name: Nepal
06 Country Id: 2||Country name: Bhutan
06 After Sort by id:
07 Country Id: 1|| Country name: India
08 Country Id: 2|| Country name: Bhutan
09 Country Id: 3|| Country name: Nepal
10 Country Id: 4|| Country name: China
10 After Sort by name:
11 Country Id: 2|| Country name: Bhutan
12 Country Id: 4|| Country name: China
13 Country Id: 1|| Country name: India
14 Country Id: 3|| Country name: Nepal
15

```

7) how to sort without using collection sorting method

Ans-`ArrayList < Integer > arraylist = new ArrayList < Integer > ();`

```

arraylist.add(10010);
arraylist.add(5);
arraylist.add(4);
arraylist.add(2);

for (int i = 0; i < arraylist.size(); i++) {

    for (int j = arraylist.size() - 1; j > i; j--) {
        if (arraylist.get(i) > arraylist.get(j)) {

            int tmp = arraylist.get(i);
            arraylist.get(i) = arraylist.get(j);
            arraylist.get(j) = tmp;

        }
    }
}

```

```
for (int i: arraylist) {  
    System.out.println(i);  
}
```

It is giving error while swapping, The LHS should be variable. I understand it. Set method works here but I do not want to use. Is there a way to do it without using set method? Help is really appreciated.

Or

```
arraylist.get(i)= arraylist.get(i);  
arraylist.get(j) =tmp;
```

You can't assign a value to a method call. As the compiler told you, the left hand side of an assignment must be a variable.

Use set method :

```
arraylist.set(i,arraylist.get(j));  
arraylist.set(j,tmp);
```

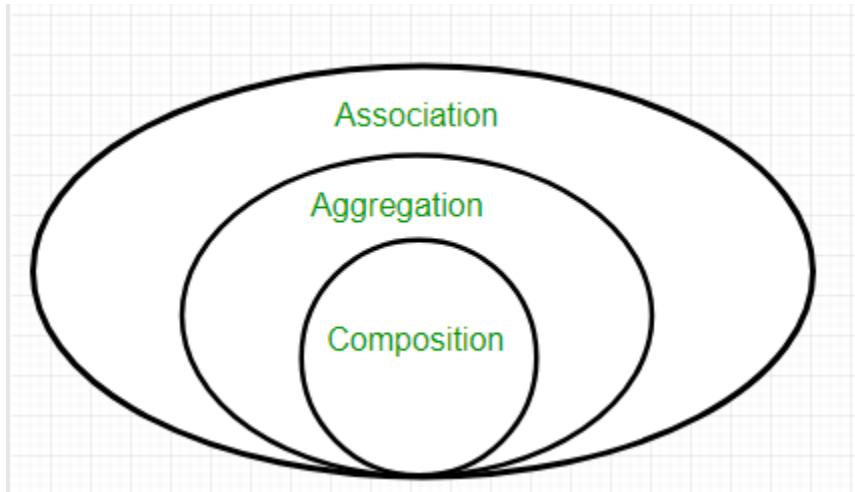
Is there a way to do it without using set method?

No. Unless you wish to convert your ArrayList to an array, sort the array, and update the ArrayList with the sorted array.

8) difference between Association, Composition and Aggregation

Ans-Association

Association is relation between two separate classes which establishes through their Objects. Association can be one-to-one, one-to-many, many-to-one, many-to-many. In Object-Oriented programming, an Object communicates to other Object to use functionality and services provided by that object. **Composition** and **Aggregation** are the two forms of association.



filter_none

edit

play_arrow brightness_4

```
// Java program to illustrate the
// concept of Association
import java.io.*;

// class bank
class Bank
{
    private String name;

    // bank name
    Bank(String name)
    {
        this.name = name;
    }

    public String getBankName()
    {
        return this.name;
    }
}

// employee class
class Employee
{
    private String name;

    // employee name
    Employee(String name)
    {
        this.name = name;
    }

    public String getEmployeeName()
    {
        return this.name;
    }
}

// Association between both the
// classes in main method
class Association
{
    public static void main (String[] args)
    {
        Bank bank = new Bank("Axis");
        Employee emp = new Employee("Neha");

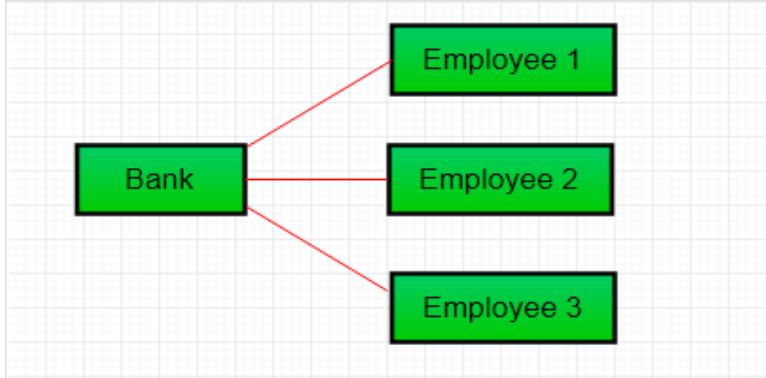
        System.out.println(emp.getEmployeeName() +
                           " is employee of " + bank.getBankName());
    }
}
```

}

Output:

```
Neha is employee of Axis
```

In above example two separate classes Bank and Employee are associated through their Objects. Bank can have many employees, So it is a one-to-many relationship.



Aggregation

It is a special form of Association where:

- It represents **Has-A** relationship.
- It is a **unidirectional association** i.e. a one way relationship. For example, department can have students but vice versa is not possible and thus unidirectional in nature.
- In Aggregation, **both the entries can survive individually** which means ending one entity will not effect the other entity

filter_none

edit

play_arrow

brightness_4

```
// Java program to illustrate  
//the concept of Aggregation.  
import java.io.*;  
import java.util.*;  
  
// student class  
class Student  
{  
    String name;  
    int id ;  
    String dept;  
  
    Student(String name, int id, String dept)  
    {
```

```

        this.name = name;
        this.id = id;
        this.dept = dept;

    }

}

/* Department class contains list of student
Objects. It is associated with student
class through its Object(s). */
class Department
{

    String name;
    private List<Student> students;
    Department(String name, List<Student> students)
    {

        this.name = name;
        this.students = students;

    }

    public List<Student> getStudents()
    {
        return students;
    }
}

/* Institute class contains list of Department
Objects. It is asoociated with Department
class through its Object(s).*/
class Institute
{

    String instituteName;
    private List<Department> departments;

    Institute(String instituteName, List<Department> departments)
    {
        this.instituteName = instituteName;
        this.departments = departments;
    }

    // count total students of all departments
    // in a given institute
    public int getTotalStudentsInInstitute()
    {
        int noOfStudents = 0;
        List<Student> students;
        for(Department dept : departments)
        {
            students = dept.getStudents();

```

```

        for(Student s : students)
        {
            noOfStudents++;
        }
    }
    return noOfStudents;
}

}

// main method
class GFG
{
    public static void main (String[] args)
    {
        Student s1 = new Student("Mia", 1, "CSE");
        Student s2 = new Student("Priya", 2, "CSE");
        Student s3 = new Student("John", 1, "EE");
        Student s4 = new Student("Rahul", 2, "EE");

        // making a List of
        // CSE Students.
        List <Student> cse_students = new ArrayList<Student>();
        cse_students.add(s1);
        cse_students.add(s2);

        // making a List of
        // EE Students
        List <Student> ee_students = new ArrayList<Student>();
        ee_students.add(s3);
        ee_students.add(s4);

        Department CSE = new Department("CSE", cse_students);
        Department EE = new Department("EE", ee_students);

        List <Department> departments = new ArrayList<Department>();
        departments.add(CSE);
        departments.add(EE);

        // creating an instance of Institute.
        Institute institute = new Institute("BITS", departments);

        System.out.print("Total students in institute: ");
        System.out.print(institute.getTotalStudentsInInstitute());
    }
}

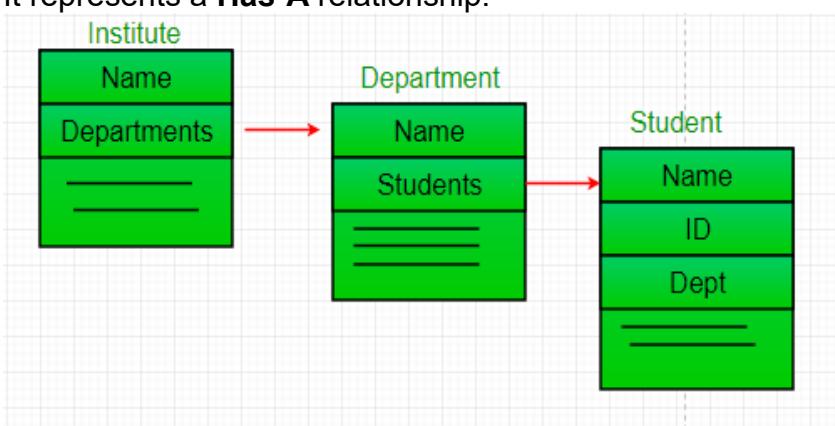
```

Output:

Total students in institute: 4

In this example, there is an Institute which has no. of departments like CSE, EE. Every department has no. of students. So, we make a Institute class which has a reference to Object or no. of Objects (i.e. List of Objects) of the Department class. That means Institute class is associated with Department class through its Object(s). And

Department class has also a reference to Object or Objects (i.e. List of Objects) of Student class means it is associated with Student class through its Object(s). It represents a **Has-A** relationship.



When do we use Aggregation ??

Code reuse is best achieved by aggregation.

Composition

Composition is a restricted form of Aggregation in which two entities are highly dependent on each other.

- It represents **part-of** relationship.
- In composition, both the entities are dependent on each other.
- When there is a composition between two entities, the composed object **cannot exist** without the other entity.

Lets take example of **Library**.

filter_none

edit

play_arrow

brightness_4

```
// Java program to illustrate
// the concept of Composition
import java.io.*;
import java.util.*;

// class book
class Book
{

    public String title;
    public String author;

    Book(String title, String author)
    {
```

```

        this.title = title;
        this.author = author;
    }
}

// Library class contains
// list of books.
class Library
{

    // reference to refer to list of books.
    private final List<Book> books;

    Library (List<Book> books)
    {
        this.books = books;
    }

    public List<Book> getTotalBooksInLibrary() {

        return books;
    }

}

// main method
class GFG
{
    public static void main (String[] args)
    {

        // Creating the Objects of Book class.
        Book b1 = new Book("EffectiveJ Java", "Joshua Bloch");
        Book b2 = new Book("Thinking in Java", "Bruce Eckel");
        Book b3 = new Book("Java: The Complete Reference", "Herbert Schildt");

        // Creating the list which contains the
        // no. of books.
        List<Book> books = new ArrayList<Book>();
        books.add(b1);
        books.add(b2);
        books.add(b3);

        Library library = new Library(books);

        List<Book> bks = library.getTotalBooksInLibrary();
        for(Book bk : bks){

            System.out.println("Title : " + bk.title + " and "
            +" Author : " + bk.author);
        }
    }
}

```

```
}
```

Output

```
Title : Effective Java and Author : Joshua Bloch
```

```
Title : Thinking in Java and Author : Bruce Eckel
```

```
Title : Java: The Complete Reference and Author : Herbert Schildt
```

In above example a library can have no. of **books** on same or different subjects. So, If Library gets destroyed then All books within that particular library will be destroyed. i.e. book can not exist without library. That's why it is composition.

Aggregation vs Composition

1. **Dependency:** Aggregation implies a relationship where the child **can exist independently** of the parent. For example, Bank and Employee, delete the Bank and the Employee still exist. whereas Composition implies a relationship where the child **cannot exist independent** of the parent. Example: Human and heart, heart don't exist separate to a Human
2. **Type of Relationship:** Aggregation relation is “**has-a**” and composition is “**part-of**”relation.
3. **Type of association:** Composition is a **strong** Association whereas Aggregation is a **weak** Association.

filter_none

edit

play_arrow

brightness_4

```
// Java program to illustrate the
// difference between Aggregation
// Composition.

import java.io.*;

// Engine class which will
// be used by car. so 'Car'
// class will have a field
// of Engine type.
class Engine
{
    // starting an engine.
    public void work()
    {

        System.out.println("Engine of car has been started ");

    }
}

// Engine class
```

```

final class Car
{
    // For a car to move,
    // it need to have a engine.
    private final Engine engine; // Composition
    //private Engine engine;      // Aggregation

    Car(Engine engine)
    {
        this.engine = engine;
    }

    // car start moving by starting engine
    public void move()
    {

        //if(engine != null)
        //{
        //    engine.work();
        //    System.out.println("Car is moving ");
        //}
    }
}

class GFG
{
    public static void main (String[] args)
    {

        // making an engine by creating
        // an instance of Engine class.
        Engine engine = new Engine();

        // Making a car with engine.
        // so we are passing a engine
        // instance as an argument while
        // creating instace of Car.
        Car car = new Car(engine);
        car.move();

    }
}

```

Output:

Engine of car has been started

Car is moving

In case of aggregation, the Car also performs its functions through an Engine. but the Engine is not always an internal part of the Car. An engine can be swapped out or even can be removed from the car. That's why we make The Engine type field non-final.

9)hashset internal flow.

Ans-HashSet uses **HashMap** internally to store it's objects. Whenever you create a **HashSet** object, one **HashMap** object associated with it is also created. This **HashMap** object is used to store the elements you enter in the **HashSet**. The elements you add into **HashSet** are stored as **keys** of this **HashMap** object. The value associated with those keys will be a **constant**.

Every constructor of **HashSet** class internally creates one **HashMap** object. You can check this in the source code of **HashSet** class in JDK installation directory. Below is the some sample code of the constructors of **HashSet** class.

```
1     private transient HashMap<E, Object> map;
2
3     //Constructor - 1
4
5     public HashSet()
6     {
7         map = new HashMap<>();           //Creating internally backing HashMap object
8     }
9
10    //Constructor - 2
11
12    public HashSet(Collection<? extends E> c)
13    {
14        map = new HashMap<>(Math.max((int) (c.size()/.75f) + 1, 16));      //Creating
15        addAll(c);
16    }
17
18    //Constructor - 3
19
20    public HashSet(int initialCapacity, float loadFactor)
21    {
22        map = new HashMap<>(initialCapacity, loadFactor);          //Creating internal
```

```
22
23     //Constructor - 4
24
25     public HashSet(int initialCapacity)
26     {
27         map = new HashMap<>(initialCapacity);           //Creating internally backing
28     }
29
30
```

You can notice that each and every constructor internally creates one new HashMap object.

How HashSet Works Internally In Java?

Whenever you insert an element into HashSet using **add()** method, it actually creates an entry in the internally backing HashMap object with element you have specified as it's key and constant called “**PRESENT**” as it's value. This “**PRESENT**” is defined in the HashSet class as below.

```
1    // Dummy value to associate with an Object in the backing Map
2    private static final Object PRESENT = new Object();
```

Let's have a look at add() method of HashSet class.

```
1    public boolean add(E e)
2    {
3        return map.put(e, PRESENT)==null;
4    }
```

You can notice that, add() method of HashSet class internally calls put() method of backing HashMap object by passing the element you have specified as a key and constant “**PRESENT**” as it's value.

remove() method also works in the same manner.

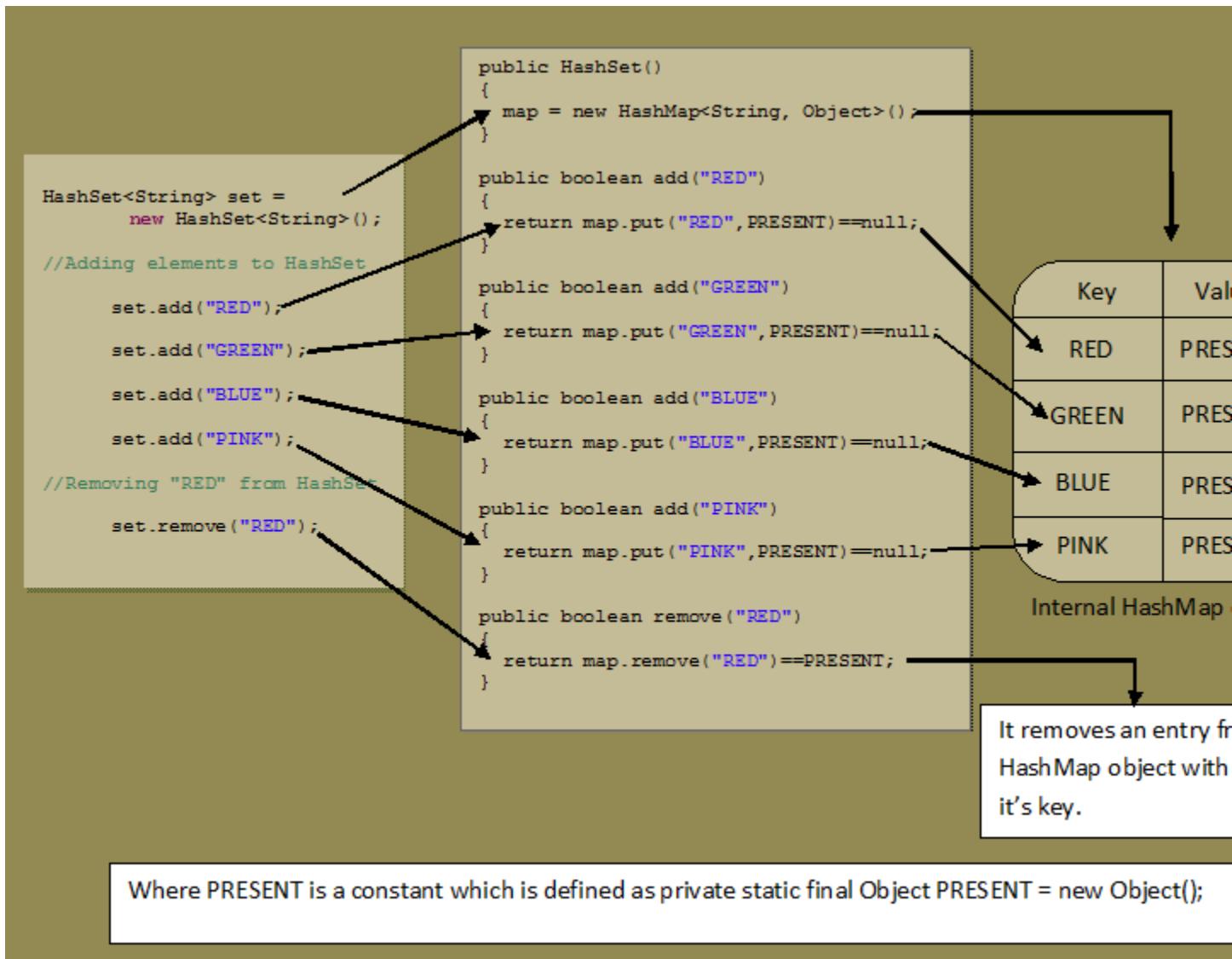
```
1    public boolean remove(Object o)
```

```
2     {
3         return map.remove(o)==PRESENT;
4     }
```

Let's see one example of HashSet and how it maintains HashMap internally.

```
1
2     public class HashSetExample
3     {
4         public static void main(String[] args)
5         {
6             //Creating One HashSet object
7
8             HashSet<String> set = new HashSet<String>();
9
10            //Adding elements to HashSet
11
12            set.add("RED");
13
14            set.add("GREEN");
15
16            set.add("BLUE");
17
18            set.add("PINK");
19
20            //Removing "RED" from HashSet
21            set.remove("RED");
22        }
23    }
```

See the below picture how above program works internally. You can observe that internal HashMap object contains elements of HashSet as keys and constant “PRESENT” as their value.



In the same manner, all methods of HashSet class process internally backing HashMap object to get the desired result. If you know how HashMap works, it will be easy for you to understand how HashSet works. You go through the source code of HashSet class once, you will get a clear picture about how HashSet works internally in Java.

10)difference between fail fast and failsafe

Ans-In this article, we'll introduce the concept of Fail-Fast and Fail-Safe Iterators.**Fail-Fast systems abort operation as-fast-as-possible exposing failures immediately and stopping the whole operation.**

Whereas, **Fail-Safe systems don't abort an operation in the case of a failure. Such systems try to avoid raising failures as much as possible.**

2. Fail-Fast Iterators

Fail-fast iterators in Java don't play along when the underlying collection gets modified.

Collections maintain an internal counter called *modCount*. Each time an item is added or removed from the *Collection*, this counter gets incremented.

When iterating, on each *next()* call, the current value of *modCount* gets compared with the initial value. If there's a mismatch, it throws *ConcurrentModificationException* which aborts the entire operation.

Default iterators for *Collections* from *java.util package* such as *ArrayList*, *HashMap*, etc. are Fail-Fast.

```
1 ArrayList<Integer> numbers = // ...
2
3 Iterator<Integer> iterator = numbers.iterator();
4 while (iterator.hasNext()) {
5     Integer number = iterator.next();
6     numbers.add(50);
7 }
```

In the code snippet above, the *ConcurrentModificationException* gets thrown at the beginning of a next iteration cycle after the modification was performed.

The Fail-Fast behavior isn't guaranteed to happen in all scenarios as it's impossible to predict behavior in case of concurrent modifications. **These iterators throw *ConcurrentModificationException* on a best effort basis.**

If during iteration over a *Collection*, **an item is removed using *Iterator*'s *remove()* method, that's entirely safe and doesn't throw an exception.**

However, if the *Collection*'s *remove()* method is used for removing an element, it throws an exception:

```
1      ArrayList<Integer> numbers = // ...
2
3
4      Iterator<Integer> iterator = numbers.iterator();
5      while (iterator.hasNext()) {
6          if (iterator.next() == 30) {
7              iterator.remove(); // ok!
8          }
9
10     iterator = numbers.iterator();
11     while (iterator.hasNext()) {
12         if (iterator.next() == 40) {
13             numbers.remove(2); // exception
14         }
15     }
```

3. Fail-Safe Iterators

Fail-Safe iterators favor lack of failures over the inconvenience of exception handling.

Those iterators create a clone of the actual *Collection* and iterate over it. If any modification happens after the iterator is created, the copy still remains untouched. Hence, these *Iterators* continue looping over the *Collection* even if it's modified.

However, it's important to remember that there's no such thing as a truly Fail-Safe iterator. The correct term is Weakly Consistent.

That means, **if a *Collection* is modified while being iterated over, what the *Iterator* sees is weakly guaranteed**. This behavior may be different for different *Collections* and is documented in Javadocs of each such *Collection*.

The Fail-Safe *Iterators* have a few disadvantages, though. One disadvantage is that **the *Iterator* isn't guaranteed to return updated data from the *Collection***, as it's working on the clone instead of the actual *Collection*.

Another disadvantage is the overhead of creating a copy of the *Collection*, both regarding time and memory.

Iterators on *Collections* from `java.util.concurrent` package such as `ConcurrentHashMap`, `CopyOnWriteArrayList`, etc. are Fail-Safe in nature.

```
1   ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();
2
3   map.put("First", 10);
4   map.put("Second", 20);
5   map.put("Third", 30);
6   map.put("Fourth", 40);
7
8   Iterator<String> iterator = map.keySet().iterator();
9
10  while (iterator.hasNext()) {
11      String key = iterator.next();
12      map.put("Fifth", 50);
13  }
```

In the code snippet above, we're using Fail-Safe *Iterator*. Hence, even though a new element is added to the *Collection* during the iteration, it doesn't throw an exception.

The default iterator for the `ConcurrentHashMap` is weakly consistent. This means that this *Iterator* can tolerate concurrent modification, traverses elements as they existed when *Iterator* was constructed and may (but isn't guaranteed to) reflect modifications to the *Collection* after the construction of the *Iterator*.

Hence, in the code snippet above, the iteration loops five times, which means **it does detect the newly added element to the *Collection***.

11) how to create datasource in hibernate.

Ans- I'm creating SessionFactory and I have my datasource as object in code where I'm creating SessionFactory, but I cannot set datasource to Hibernate Configuration object. So how can I set my datasource to my SessionFactory?

```
Configuration configuration = new Configuration();
Properties properties = new Properties();
properties.setProperty("hibernate.dialect", "org.hibernate.dialect.MySQLInnoDBDialect");
configuration.setProperties(properties);
configuration.setProperty("packagesToScan", "com.my.app");
SessionFactory sessionFactory = configuration.configure().buildSessionFactory();
```

12).how create multiple data source

Ans- **Spring Boot** with **Spring Data** makes it easy to access a database through so called Repositories. But what if you want to access **multiple databases** maybe even with different Database Management Systems?

Luckily Spring provides a way of doing this.

I provided an example project with two PostgreSQL datasources on GitHub: <https://github.com/jahe/spring-boot-multiple-datasources>

1. Add an additional datasource configuration to your application.properties

```
# Oracle DB - "foo"
spring.datasource.url=jdbc:oracle:thin:@//db-server-foo:1521/FOO
spring.datasource.username=fooadmin
spring.datasource.password=foo123
spring.datasource.driver-class-name=oracle.jdbc.OracleDriver
# PostgreSQL DB - "bar"
bar.datasource.url=jdbc:postgresql://db-server-bar:5432/bar
bar.datasource.username=baradmin
```

```
bar.datasource.password=bar123  
bar.datasource.driver-class-name=org.postgresql.Driver
```

2. Set the SQL Dialect to “default” in your application.properties to let Spring autodetect the different SQL Dialects of each datasource

```
spring.jpa.database=default
```

3. Create a Java Package for each datasource with two nested Packages “domain” and “repo”

```
src/main/java  
- com.foobar  
  - foo  
    - domain  
    - repo  
  - bar  
    - domain  
    - repo
```

4. Create a Configuration Class for the Oracle database “foo” named “FooDbConfig.java”

```
package com.foobar;  
  
@Configuration  
@EnableTransactionManagement  
@EnableJpaRepositories(  
    entityManagerFactoryRef = "entityManagerFactory",
```

```
basePackages = { "com.foobar.foo.repo" }
)
public class FooDbConfig {

    @Primary
    @Bean(name = "dataSource")
    @ConfigurationProperties(prefix = "spring.datasource")
    public DataSource dataSource() {
        return DataSourceBuilder.create().build();
    }

    @Primary
    @Bean(name = "entityManagerFactory")
    public LocalContainerEntityManagerFactoryBean
    entityManagerFactory(
        EntityManagerFactoryBuilder builder,
        @Qualifier("dataSource") DataSource dataSource
    ) {
        return builder
            .dataSource(dataSource)
            .packages("com.foobar.foo.domain")
            .persistenceUnit("foo")
            .build();
    }

    @Primary
    @Bean(name = "transactionManager")
    public PlatformTransactionManager transactionManager(
        @Qualifier("entityManagerFactory") EntityManagerFactory
        entityManagerFactory
    ) {
        return new JpaTransactionManager(entityManagerFactory);
    }
}
```

5. Create a Configuration Class for the PostgreSQL database “bar” named “BarDbConfig.java”

```
package com.foobar;

@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(
    entityManagerFactoryRef = "barEntityManagerFactory",
    transactionManagerRef = "barTransactionManager",
    basePackages = { "com.foobar.bar.repo" }
)
public class BarDbConfig {

    @Bean(name = "barDataSource")
    @ConfigurationProperties(prefix = "bar.datasource")
    public DataSource dataSource() {
        return DataSourceBuilder.create().build();
    }

    @Bean(name = "barEntityManagerFactory")
    public LocalContainerEntityManagerFactoryBean
    barEntityManagerFactory(
        EntityManagerFactoryBuilder builder,
        @Qualifier("barDataSource") DataSource dataSource
    ) {
        return
            builder
                .dataSource(dataSource)
                .packages("com.foobar.bar.domain")
                .persistenceUnit("bar")
                .build();
    }

    @Bean(name = "barTransactionManager")
    public PlatformTransactionManager barTransactionManager(
        @Qualifier("barEntityManagerFactory") EntityManagerFactory
        barEntityManagerFactory
    ) {
        return new JpaTransactionManager(barEntityManagerFactory);
    }
}
```

6. Create an Entity “`Foo.java`” for the Oracle database “`foo`”

```
package com.foobar.foo.domain;

@Entity
@Table(name = "FOO")
public class Foo {

    @Id
    @GeneratedValue
    @Column(name = "ID")
    private Long id;

    @Column(name = "FOO")
    private String foo;

    Foo(String foo) {
        this.foo = foo;
    }

    Foo() {
        // Default constructor needed by JPA
    }
}
```

7. Create a Repository “`FooRepository.java`” for the Oracle database “`foo`”

```
package com.foobar.foo.repo;

@Repository
public interface FooRepository extends JpaRepository<Foo, Long> {

    Foo findById(Long id);

}
```

8. Create an Entity “Bar.java” for the PostgreSQL database “bar”

```
package com.foobar.bar.domain;

@Entity
@Table(name = "BAR")
public class Bar {

    @Id
    @GeneratedValue
    @Column(name = "ID")
    private Long id;

    @Column(name = "BAR")
    private String bar;

    Bar(String bar) {
        this.bar = bar;
    }

    Bar() {
        // Default constructor needed by JPA
    }
}
```

9. Create a Repository “BarRepository.java” for the PostgreSQL database “bar”

```
package com.foobar.bar.repo;

@Repository
public interface BarRepository extends JpaRepository<Bar, Long> {

    Bar findById(Long id);

}
```

10. Create the Spring Boot Main Class “Application.java”

```
package com.foobar;

@SpringBootApplication
public class Application {

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

}
```

11. Use the Repositories in a REST Controller (or somewhere else)

```
package com.foobar;

@RestController
public class FooBarController {

    private final FooRepository fooRepo;
    private final BarRepository barRepo;

    @Autowired
    FooBarController(FooRepository fooRepo, BarRepository barRepo)
    {
        this.fooRepo = fooRepo;
        this.barRepo = barRepo;
    }

    @RequestMapping("/foobar/{id}")
    public String fooBar(@PathVariable("id") Long id) {
        Foo foo = fooRepo.findById(id);
        Bar bar = barRepo.findById(id);

        return foo.getFoo() + " " + bar.getBar();
    }
}
```

```
}
```

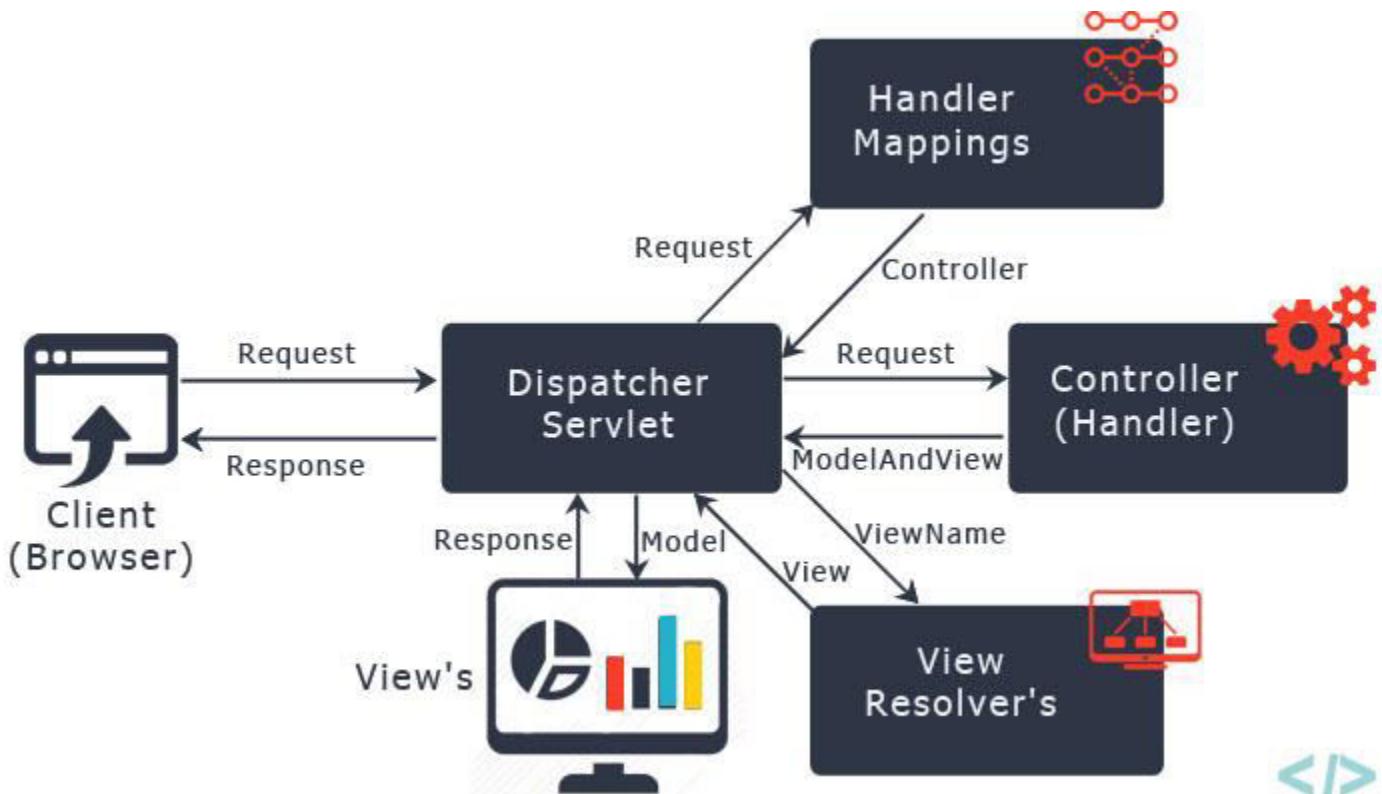
13) explain spring mvc

Ans- Spring MVC is mostly used with Spring for any web application development. It is very powerful and nice layered architecture for flow and configuration. It is very flexible to integrate with other web frameworks like struts.

In Spring Web MVC, DispatcherServlet class works as the front controller. It is responsible to manage the flow of the spring mvc application.

The @Controller annotation is used to mark the class as the controller in Spring 3.

The @RequestMapping annotation is used to map the request url. It is applied on the method.



Spring MVC Architecture

Spring MVC Execution Flow

Step 1: First request will be received by DispatcherServlet.

Step 2: DispatcherServlet will take the help of HandlerMapping and get to know the Controller class name associated with the given request.

Step 3: So request transfer to the Controller, and then controller will process the request by executing appropriate methods and returns ModelAndView object (contains Model data and View name) back to the DispatcherServlet.

Step 4: Now DispatcherServlet send the model object to the ViewResolver to get the actual view page.

Step 5: Finally DispatcherServlet will pass the Model object to the View page to display the result.

Now Let us understand Spring MVC flow with hello world application Example.

First of all, we create Dynamic web project in eclipse and following are the required files for the same:

- JavaController.java
- welcomePage.jsp
- web.xml
- welcome-servlet.xml
- index.jsp

JavaController.java

JavaController.java

Java



```
1 package com.codeNuclear;
2
3 import javax.servlet.http.HttpServletRequest;
4
5 import org.springframework.stereotype.Controller;
6 import org.springframework.ui.ModelMap;
7 import org.springframework.web.bind.annotation.RequestMapping;
8 import org.springframework.web.servlet.ModelAndView;
9
10
11 @Controller
12 public class JavaController
13 {
14     @RequestMapping("/codeNuclear")
15     public ModelAndView helloWorld(ModelMap model, HttpServletRequest request)
16     {
17
18         String message = "Welcome to codeNuclear.com Spring MVC Sessions";
19         message += "<br>You Did it....!";
20
21         System.out.println(message);
22         model.addAttribute("message", message);
23
24         return new ModelAndView("welcomePage", "welcomeMessage", message);
25     }//ModelAndView closed
26 }
27
```

welcomePage.jsp

welcomePage.jsp

Java



0

```
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1">
3 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
4 <html>
5 <body>
6   <font face="verdana" size="2">
7     ${welcomeMessage}
8   </font>
9 </body>
10 </html>
```

11

web.xml

web.xml

XHTML



0

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
3 id="WebApp_ID" version="2.5">
4   <display-name>SpringMvcExample</display-name>
5
6   <servlet>
7     <servlet-name>welcome</servlet-name>
```

```
8 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
9 <load-on-startup>1</load-on-startup>
10 </servlet>
11
12 <servlet-mapping>
13 <servlet-name>welcome</servlet-name>
14 <url-pattern>/</url-pattern>
15 </servlet-mapping>
16 <welcome-file-list>
17 <welcome-file>/</welcome-file>
18 </welcome-file-list>
19 </web-app>
```

welcome-servlet.xml

servlet.xml

XHTML



0

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3   xmlns:context="http://www.springframework.org/schema/context"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5   xsi:schemaLocation="
6     http://www.springframework.org/schema/beans
7     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
8     http://www.springframework.org/schema/context
9     http://www.springframework.org/schema/context/spring-context-3.0.xsd">
10
11 <context:component-scan base-package="com.codeNuclear" />
```

```
12
13 <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
14 <property name="prefix" value="/jsp/" />
15 <property name="suffix" value=".jsp" />
16 </bean>
17
18 </beans>
19
```

index.jsp

index.jsp

Java



0

```
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
4 <html>
5 <head>
6 <title>codeNuclear.com Spring MVC 3.x</title>
7 </head>
8 <body>
9
10 <font size="2px" face="verdana">
11   Welcome...
12   <a href="codeNuclear.html"><br> Click here to check the output.</a>
13 </font>
14
15 </body>
16 </html>
```

Practical Execution Flow

Step 1) Run the application, then index.jsp file will be executed then click on the link given (I have given Click here to check the output.).

Step 2) Once you click on that link, container will check the URL pattern at web.xml and passes the request to the DispatcherServlet.

Step 3) DispatcherServlet then passes that request to our controller class.

Step 4) DispatcherServlet verifies this ‘codeNuclear’ name with the string in @RequestMapping(“-“) in our controller class if same it will executes the ModelAndView method, which gives ModelAndView object as return type.

Java



0

1 return new ModelAndView("welcomePage", "welcomeMessage", message);

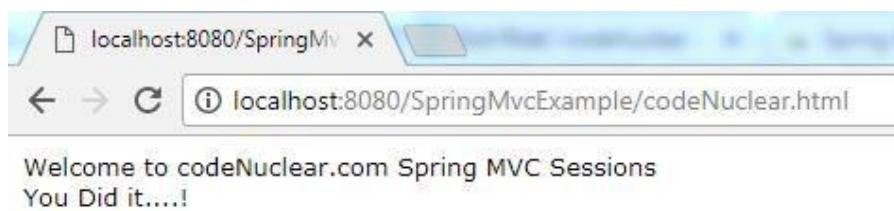
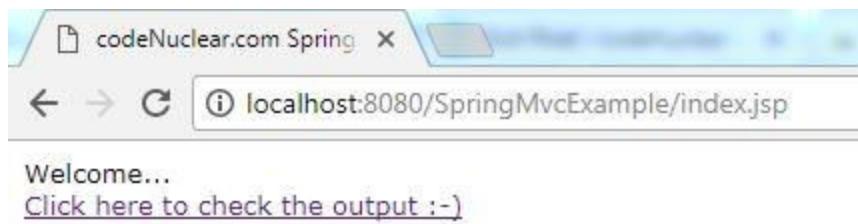
2

Means first argument is ‘View’ page name [Where we are sending our result], second, third arguments are key,values.

Step 5) So DispatcherServlet search for the name welcomePage in /jsp folder with extension .jsp , once the file was opened you can access the data by using the key welcomeMessage [2nd parameter in ModelAndView object].

Check welcomePage.jsp and printing the result by calling the key \${welcomeMessage} from welcomePage.jsp.

Output



14) how to give security in spring mvc

Ans- <https://docs.spring.io/spring-security/site/docs/current/guides/html5/helломvc-javaproject.html>

15) what is dependency injection

Ans- Dependency injection is basically providing the objects that an object needs (its dependencies) instead of having it construct them itself. It's a very useful technique for testing, since it allows dependencies to be mocked or stubbed out.

Dependencies can be injected into objects by many means (such as constructor injection or setter injection). One can even use specialized dependency injection frameworks (e.g. Spring) to do that, but they certainly aren't required. You don't need those frameworks to have dependency injection. Instantiating and passing objects (dependencies) explicitly is just as good an injection as injection by framework..

16) how many ways create auto wired.

Ans- have `@Autowired` service which has to be used from within a static method. I know this is wrong but I cannot change the current design as it would require a lot of work, so I need some simple hack for that. I can't change `randomMethod()` to be non-static and I need to use this autowired bean. Any clues how to do that?

```
@Service  
public class Foo {  
    public int doStuff() {  
        return 1;  
    }
```

```

        }

}

public class Boo {
    @Autowired
    Foo foo;

    public static void randomMethod() {
        foo.doStuff();
    }
}

```

17) which on better setter based or constructor based DI.

Ans- A class that takes a required dependency as a constructor argument can only be instantiated if that argument is provided (you should have a guard clause to make sure the argument is not null.) A constructor therefore enforces the dependency requirement whether or not you're using Spring, making it container-agnostic.

If you use setter injection, the setter may or may not be called, so the instance may never be provided with its dependency. The only way to force the setter to be called is using `@Required` or `@Autowired`, which is specific to Spring and is therefore not container-agnostic.

So to keep your code independent of Spring, use constructor arguments for injection.

18) how many ways create thread.

Ans- Threads can be created mainly in 3 different ways

1. Extend the `java.lang.Thread` class'

```

class SampleThread extends Thread {

    //method where the thread execution will start
    public void run(){
        //logic to execute in a thread
    }

    //let's see how to start the threads
    public static void main(String[] args){
        Thread t1 = new SampleThread();
        Thread t2 = new SampleThread();
        t1.start(); //start the first thread. This calls the run() method.
        t2.start(); //this starts the 2nd thread. This calls the run() method.
    }
}

```

2. Implement the `java.lang.Runnable` interface

```

class A implements Runnable{

```

```

@Override
public void run() {

    // implement run method here
}

public static void main() {
    final A obj = new A();

    Thread t1 = new Thread(new A());

    t1.start();
}
}

```

3. Implement the java.util.concurrent.Callable interface

```

class Counter implements Callable {

    private static final int THREAD_POOL_SIZE = 2;

    // method where the thread execution takes place
    public String call() {
        return Thread.currentThread().getName() + " executing ...";
    }

    public static void main(String[] args) throws InterruptedException,
        ExecutionException {
        // create a pool of 2 threads
        ExecutorService executor = Executors
            .newFixedThreadPool(THREAD_POOL_SIZE);

        Future future1 = executor.submit(new Counter());
        Future future2 = executor.submit(new Counter());

        System.out.println(Thread.currentThread().getName() + " executing ...");

        //asynchronously get from the worker threads
        System.out.println(future1.get());
        System.out.println(future2.get());
    }
}

```

Favor Callable interface with the Executor framework for thread pooling.

The Runnable or Callable interface is preferred over extending the Thread class

19) which one better implement the interface or extend thread.

Ans- Threading is a facility to allow multiple tasks to run concurrently within a single process. Threads are independent, concurrent execution through a program, and each thread has its own stack.

In Java threads can be implemented in two ways.

1)by Extending Thread Class

e.g.

```
class ThreadDemo extends Thread{  
    public void run(){  
        System.out.println("thread is running...");  
    }  
    public static void main(String args[]){  
        ThreadDemo t1=new ThreadDemo();  
        t1.start();  
    }  
}
```

Output:thread is running...

2)by Implementing Runnable Interface

e.g.

```
class RunnableDemo implements Runnable{  
    public void run(){  
        System.out.println("thread is running...");  
    }  
    public static void main(String args[]){  
        RunnableDemo rd=new RunnableDemo();  
        Thread t1 =new Thread(rd);  
        t1.start();  
    }  
}
```

Output:thread is running...

Creating a thread in Java

difference between extends thread and implements runnable

- The limitation with "extends Thread" approach is that if you extend Thread, you can not extend other class . Java does not support multiple inheritance.

While with implement runnable we can extend other class also.

- In "implements Runnable" , we are creating a different Runnable class for a specific task. It gives us the freedom to reuse the specific behavior task whenever required.

"extends Thread" contains both thread and job specific behavior code. Hence once thread completes execution , it can not be restart again.

- "implements Runnable" makes the code loosely-coupled and easier to read .Because the code is split into two classes . Thread class for the thread specific code and your Runnable implementation class for your task that should be run by a thread code.

"extends Thread" makes the code tightly coupled . Single class contains the thread code as well as the task that needs to be done by the thread.

there are many other difference also.

As per he above difference second approach has many advantages over the first approach so second approach is preferable.



20) how to connect dispatcher servlet to ViewandModel.

Ans- <https://dzone.com/articles/how-spring-mvc-really-works>

21) Explain Ur project Flow.

Ans- Project Architecture Means what are the layers in our project with flow diagram. Here tier can be referred as a 'layer'. suppose our project contains 3 layer like client layer, business layer and data layer then we draw the all layers flow.

In IT industry N-Tier architecture is an industry-proven software architecture model. It is suitable to support enterprise level client-server applications by providing solutions on scalability, security, fault tolerance, re-usability and maintainability. It helps developers to create flexible and reusable applications.

The Three-tier Architecture is divided into 3 layers:

Presentation layer(Client or Browser)
Business layer(Struts or Spring) we used Server(either Tomcat (or) weblogic)
Database layer(Oracle (or) MySQL)

presentation layer:

It is also known as Client layer. Top most layer of an application. The main functionality of this layer is to communicate with Application layer. For example, login page of Facebook where an end user could see text boxes and buttons to enter user id, password and to click on sign-in. This is also called view of the application.

Example: JSP

Business layer:

It is also called business logic layer or logical layer. As per the Facebook login page example, once user clicks on the login button, business layer interacts with Database layer and sends required information to the Presentation layer. It controls an application's functionality by performing detailed processing. This layer acts as a mediator between the Presentation and the Database layer. Complete business logic will be written in this layer.

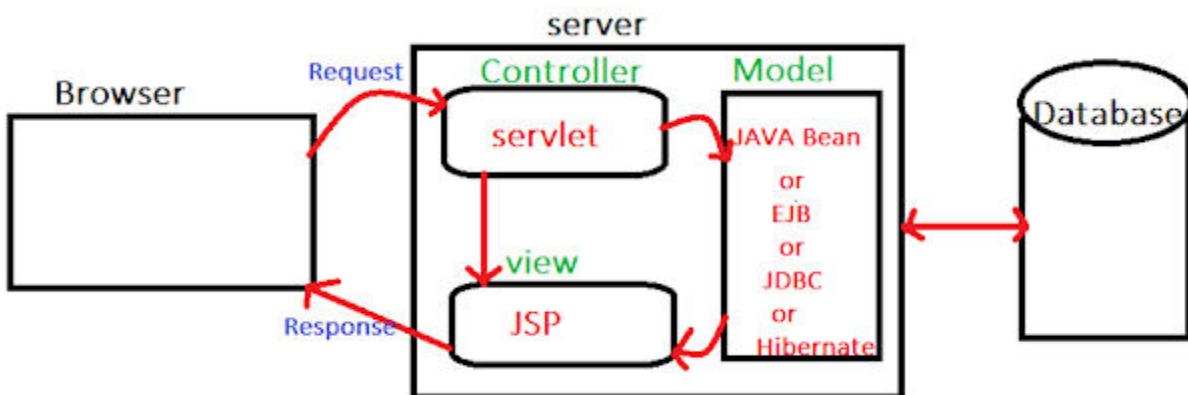
Example: STRUTS, SPRING

Data

layer:

The data is stored in this layer. Business layer communicates with Database layer to retrieve the data. It contains methods that connects the database and performs required action e.g.: insert, update, delete etc.

Example: oracle, MySQL



Step 1: When ever we submitting the request from client(Browser) to Server component then

controller receives the request from Browser. So you must remember every request will reach to controller component.

Step 2: The controller receives the request from Browser then Controller has to identify a particular model. Then it will execute business logic. Here model is might be java class or business logic's available. Here controller acts as mediator between view and the model. Finally, after performing some action on server side then controller identify respective view component. That means JSP page has view part.

Step 3: After executing model part if it is required to interact with database then model part interact with database.

Step 4: Then view layer displays of the model to the end user. Then generate the require dynamic web page return to Browser. Finally view part send to the response to the client(Browser)

22) what is iterator

Ans- Iterators are used in [Collection framework](#) in Java to retrieve elements one by one. There are three iterators.

Enumeration :

It is a interface used to get elements of legacy collections(Vector, Hashtable). Enumeration is the first iterator present from JDK 1.0, rests are included in JDK 1.2 with more functionality.

Enumerations are also used to specify the input streams to a *SequenceInputStream*. We can create Enumeration object by calling *elements()* method of vector class on any vector object

```
// Here "v" is an Vector class object. e is of
```

```
// type Enumeration interface and refers to "v"
```

```
Enumeration e = v.elements();
```

There are **two** methods in Enumeration interface namely :

```
// Tests if this enumeration contains more elements  
public boolean hasMoreElements();
```

```
// Returns the next element of this enumeration
// It throws NoSuchElementException
// if no more element present
public Object nextElement();
filter_none

edit
play_arrow
brightness_4
// Java program to demonstrate Enumeration

import java.util.Enumeration;
import java.util.Vector;

public class Test
{
    public static void main(String[] args)
    {
        // Create a vector and print its contents
        Vector v = new Vector();
        for (int i = 0; i < 10; i++)
            v.addElement(i);
        System.out.println(v);

        // At beginning e(cursor) will point to
        // index just before the first element in v
        Enumeration e = v.elements();

        // Checking the next element availability
        while (e.hasMoreElements())
        {
            // moving cursor to next element
            int i = (Integer)e.nextElement();
        }
    }
}
```

```

        System.out.print(i + " ");
    }
}

}

```

Output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
0 1 2 3 4 5 6 7 8 9
```

Limitations of Enumeration :

- Enumeration is for **legacy** classes(Vector, Hashtable) only. Hence it is not a universal iterator.
- Remove operations can't be performed using Enumeration.
- Only forward direction iterating is possible.

Iterator:

It is a **universal** iterator as we can apply it to any Collection object. By using Iterator, we can perform both read and remove operations. It is improved version of Enumeration with additional functionality of remove-ability of a element.

Iterator must be used whenever we want to enumerate elements in all Collection framework implemented interfaces like Set, List, Queue, Deque and also in all implemented classes of Map interface. Iterator is the **only** cursor available for entire collection framework.

Iterator object can be created by calling *iterator()* method present in Collection interface.

```
// Here "c" is any Collection object. itr is of
```

```
// type Iterator interface and refers to "c"
```

```
Iterator itr = c.iterator();
```

Iterator interface defines **three** methods:

```
// Returns true if the iteration has more elements
```

```
public boolean hasNext();
```

```
// Returns the next element in the iteration
```

```
// It throws NoSuchElementException if no more
```

```
// element present
```

```
public Object next();
```

```
// Remove the next element in the iteration
```

```
// This method can be called only once per call
```

```
// to next()
```

```
public void remove();
```

remove() method can throw two exceptions

- *UnsupportedOperationException* : If the remove operation is not supported by this iterator

- *IllegalStateException* : If the next method has not yet been called, or the remove method has already been called after the last call to the next method

filter_none

edit

play_arrow

brightness_4

```
// Java program to demonstrate Iterator

import java.util.ArrayList;
import java.util.Iterator;

public class Test
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();

        for (int i = 0; i < 10; i++)
            al.add(i);

        System.out.println(al);

        // at beginning itr(cursor) will point to
        // index just before the first element in al
        Iterator itr = al.iterator();

        // checking the next element availability
        while (itr.hasNext())
        {
            // moving cursor to next element
            int i = (Integer)itr.next();
```

```

        // getting even elements one by one
        System.out.print(i + " ");

        // Removing odd elements
        if (i % 2 != 0)
            itr.remove();
    }

    System.out.println();
    System.out.println(al);
}

}

```

Output:

```

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
0 1 2 3 4 5 6 7 8 9
[0, 2, 4, 6, 8]

```

Limitations of Iterator :

- Only forward direction iterating is possible.
- Replacement and addition of new element is not supported by Iterator.

ListIterator:

It is only applicable for List collection implemented classes like arraylist, linkedlist etc. It provides bi-directional iteration.

ListIterator must be used when we want to enumerate elements of List. This cursor has more functionality(methods) than iterator.

ListIterator object can be created by calling *listIterator()* method present in List interface.

```
// Here "l" is any List object, ltr is of type
```

```
// ListIterator interface and refers to "l"
```

```
ListIterator ltr = l.listIterator();
```

ListIterator interface extends Iterator interface. So all three methods of Iterator interface are available for ListIterator. In addition there are six more methods.

```
// Forward direction
```

```

// Returns true if the iteration has more elements
public boolean hasNext();

// same as next() method of Iterator
public Object next();

// Returns the next element index
// or list size if the list iterator
// is at the end of the list
public int nextIndex();

// Backward direction

// Returns true if the iteration has more elements
// while traversing backward
public boolean hasPrevious();

// Returns the previous element in the iteration
// and can throws NoSuchElementException
// if no more element present
public Object previous();

// Returns the previous element index
// or -1 if the list iterator is at the
// beginning of the list
public int previousIndex();

// Other Methods

// same as remove() method of Iterator
public void remove();

// Replaces the last element returned by
// next() or previous() with the specified element
public void set(Object obj);

// Inserts the specified element into the list at
// position before the element that would be returned
// by next(),
public void add(Object obj);

```

Clearly the three methods that *ListIterator* inherits from *Iterator* (*hasNext()*, *next()*, and *remove()*) do exactly the same thing in both interfaces. The *hasPrevious()* and the previous operations are exact analogues of *hasNext()* and *next()*. The former operations refer to the element before the (implicit) cursor, whereas the latter refer to the element after the cursor. The previous operation moves the cursor backward, whereas next moves it forward.

ListIterator has no current element; its cursor position always lies between the element that would be returned by a call to *previous()* and the element that would be returned by a call to *next()*

set() method can throw four exceptions

- *UnsupportedOperationException* – if the set operation is not supported by this list iterator
- *ClassCastException* : If the class of the specified element prevents it from being added to this list
- *IllegalArgumentException* : If some aspect of the specified element prevents it from being added to this list
- *IllegalStateException* : If neither next nor previous have been called, or remove or add have been called after the last call to next or previous

add() method can throw three exceptions

- *UnsupportedOperationException* : If the add method is not supported by this list iterator
- *ClassCastException* : If the class of the specified element prevents it from being added to this list
- *IllegalArgumentException* : If some aspect of this element prevents it from being added to this list

filter_none

edit

play_arrow

brightness_4

```
// Java program to demonstrate ListIterator

import java.util.ArrayList;
import java.util.ListIterator;

public class Test
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        for (int i = 0; i < 10; i++)
            al.add(i);

        System.out.println(al);

        // at beginning ltr(cursor) will point to
        // index just before the first element in al
        ListIterator ltr = al.listIterator();
```

```

// checking the next element availability
while (ltr.hasNext())
{
    // moving cursor to next element
    int i = (Integer) ltr.next();

    // getting even elements one by one
    System.out.print(i + " ");

    // Changing even numbers to odd and
    // adding modified number again in
    // iterator
    if (i%2==0)
    {
        i++; // Change to odd
        ltr.set(i); // set method to change value
        ltr.add(i); // to add
    }
    System.out.println();
    System.out.println(al);
}
}

```

Output:

```

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
0 1 2 3 4 5 6 7 8 9
[1, 1, 1, 3, 3, 3, 5, 5, 7, 7, 7, 9, 9, 9]

```

Limitations of ListIterator : It is the most powerful iterator but it is only applicable for List implemented classes, so it is not a universal iterator.

Important Common Points

1 : Please note that initially any iterator reference will point to the index just before the index of first element in a collection.

2 : We don't create objects of Enumeration, Iterator, ListIterator because they are interfaces. We use methods like elements(), iterator(), listIterator() to create objects. These methods have anonymous [Inner classes](#) that extends respective interfaces and return this class object. This can be verified by below code. For more on inner class refer

filter_none

edit

play_arrow

brightness_4

```
// Java program to demonstrate iterators references

import java.util.Enumeration;
import java.util.Iterator;
import java.util.ListIterator;
import java.util.Vector;

public class Test
{
    public static void main(String[] args)
    {
        Vector v = new Vector();

        // Create three iterators
        Enumeration e = v.elements();
        Iterator itr = v.iterator();
        ListIterator ltr = v.listIterator();

        // Print class names of iterators
        System.out.println(e.getClass().getName());
        System.out.println(itr.getClass().getName());
        System.out.println(ltr.getClass().getName());
    }
}
```

```
    }  
}
```

Output:

```
java.util.Vector$1  
java.util.Vector$Itr  
java.util.Vector$ListItr
```

24) explain annotations in spring

Ans - Spring Annotations

- [Spring framework](#) implements and promotes the principle of control inversion (IOC) or dependency injection (DI) and is in fact an IOC container.
- Traditionally, Spring allows a developer to manage bean dependencies by using XML-based configuration.
- There is an alternative way to define beans and their dependencies. This method is a Java-based configuration.
- Unlike the XML approach, Java-based configuration allows you to manage bean components programmatically. That's why Spring annotations were introduced.

In this article we will explore most commonly used Spring Annotations and also look at some example program.

Spring Annotations List

Some of the spring core framework [annotations](#) are:

1. **@Configuration**: Used to indicate that a class declares one or more **@Bean** methods. These classes are processed by the Spring container to generate bean definitions and service requests for those beans at runtime.
2. **@Bean**: Indicates that a method produces a bean to be managed by the Spring container. This is one of the most used and important spring annotation. **@Bean** annotation also can be used with parameters like name, initMethod and destroyMethod.
 - name – allows you give name for bean
 - initMethod – allows you to choose method which will be invoked on context register
 - destroyMethod – allows you to choose method which will be invoked on context shutdown

For example:

```
@Configuration  
public class AppConfig {
```

```

@Bean(name = "comp", initMethod = "turnOn",
destroyMethod = "turnOff")
Computer computer() {
    return new Computer();
}
}

public class Computer {

    public void turnOn() {
        System.out.println("Load operating system");
    }
    public void turnOff() {
        System.out.println("Close all programs");
    }
}

```

3. `@PreDestroy` and `@PostConstruct` are alternative way for bean initMethod and destroyMethod. It can be used when the bean class is defined by us. For example;

```

public class Computer {

    @PostConstruct
    public void turnOn() {
        System.out.println("Load operating system");
    }

    @PreDestroy
    public void turnOff() {
        System.out.println("Close all programs");
    }
}

```

17. `@ComponentScan`: Configures component scanning directives for use with `@Configuration` classes. Here we can specify the base packages to scan for spring components.
18. `@Component`: Indicates that an annotated class is a “component”. Such classes are considered as candidates for auto-detection when using annotation-based configuration and classpath scanning.
19. `@PropertySource`: provides a simple declarative mechanism for adding a property source to Spring’s Environment. There is a similar annotation for adding an array of property source files i.e `@PropertySources`.
20. `@Service`: Indicates that an annotated class is a “Service”. This annotation serves as a specialization of `@Component`, allowing for implementation classes to be autodetected through classpath scanning.
21. `@Repository`: Indicates that an annotated class is a “Repository”. This annotation serves as a specialization of `@Component` and advisable to use with `DAO` classes.
22. `@Autowired`: [Spring @Autowired annotation](#) is used for automatic injection of beans. Spring `@Qualifier` annotation is used in conjunction with Autowired to avoid confusion when we have two or more bean configured for same type.

Spring MVC Annotations

Some of the important Spring MVC annotations are:

1. `@Controller`
2. `@RequestMapping`
3. `@PathVariable`
4. `@RequestParam`
5. `@ModelAttribute`
6. `@RequestBody` and `@ResponseBody`
7. `@RequestHeader` and `@ResponseHeader`

You can learn more about them at [Spring MVC Tutorial](#).

Spring Transaction Management Annotations

`@Transactional` is the spring declarative transaction management annotation, read more at [Spring MVC Hibernate](#).

Spring Security Annotations

`@EnableWebSecurity` is used with `@Configuration` class to have the Spring Security configuration defined, read more at [Spring Security Example](#).

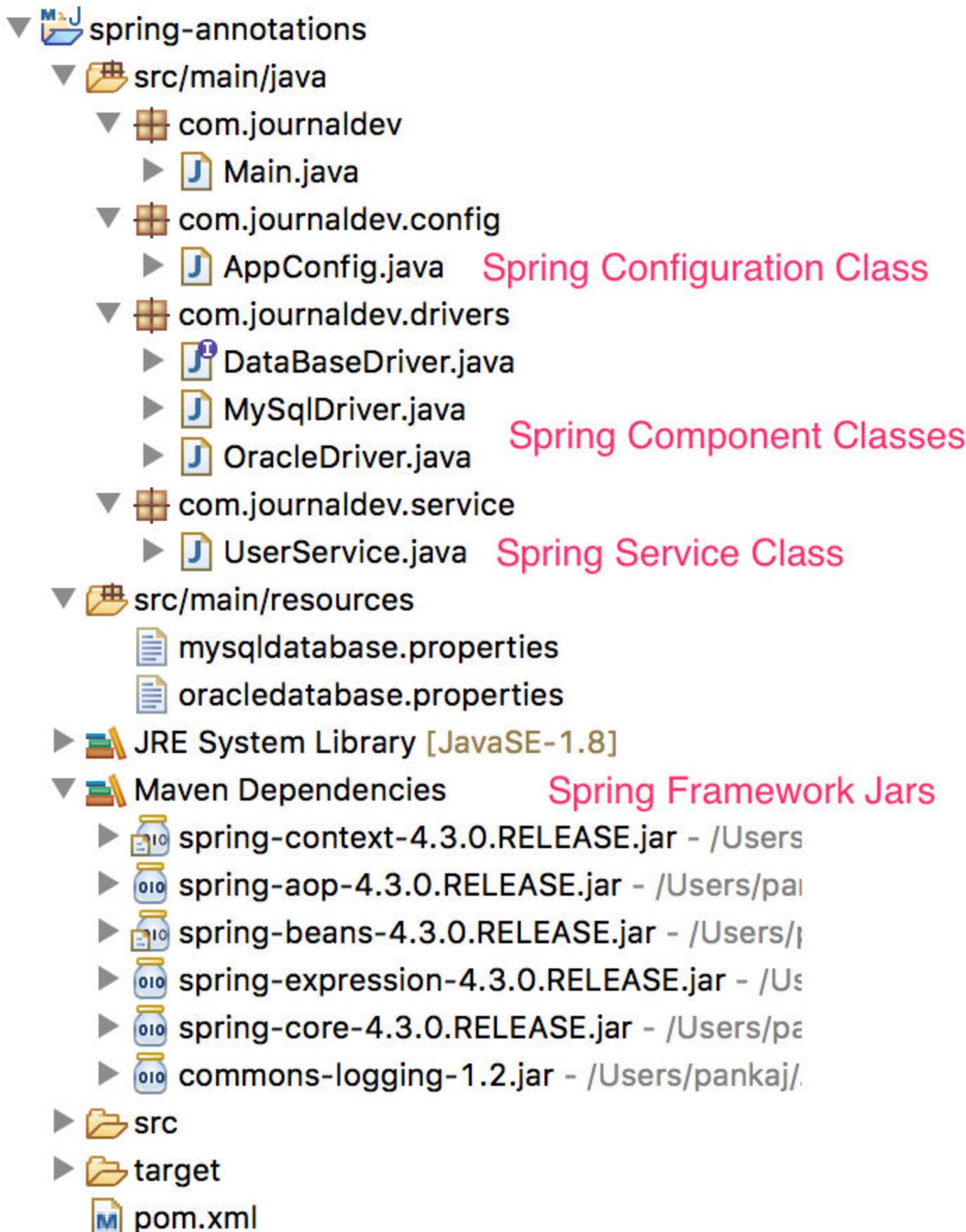
Spring Boot Annotations

1. `@SpringBootApplication`
2. `@EnableAutoConfiguration`

Read more at [Spring Boot Example](#).

Spring Annotations Example

Let's look at a simple example where we will use Spring annotations in our application. Below image illustrates my Spring Annotations Example project.



Spring Framework Dependencies

I have created the maven project and added Spring Core Framework dependencies.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.journaldev.spring</groupId>
  <artifactId>spring-annotations</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>Spring Annotations</name>

  <properties>
    <spring.framework>4.3.0.RELEASE</spring.framework>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>${spring.framework}</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>${spring.framework}</version>
    </dependency>
  </dependencies>

</project>
```

This will pull up all the spring core jars for our project.

Component Classes

Next step is to create component classes. Here I am imitating multiple database components, one for MySQL and another for Oracle.

```
package com.journaldev.drivers;

public interface DataBaseDriver {
```

```
    public String getInfo();  
}
```

DataBaseDriver is the base **interface** that we will implement.

```
package com.journaldev.drivers;  
  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.context.annotation.PropertySource;  
import org.springframework.stereotype.Component;  
  
@Component  
@PropertySource("classpath:mysqldatabase.properties")  
public class MySqlDriver implements DataBaseDriver {  
  
    @Value("${databaseName}")  
    private String databaseName;  
    @Value("${disableStatementPooling}")  
    private String disableStatementPooling;  
  
    public String getInfo() {  
        return "[ Driver: mySql" +  
                ", databaseName: " + databaseName +  
                ", disableStatementPooling: " +  
                disableStatementPooling +  
                " ]";  
    }  
}
```

Notice the use of **@Component** annotation to indicate spring framework to treat this class as a Component. We are also using **@PropertySource** and **@Value** annotations, Spring will use these at runtime to inject and set these variable values from specified property file. Below is the properties declared in **mysqldatabase.properties** file.

```
databaseName=school  
disableStatementPooling=true
```

```
package com.journaldev.drivers;  
  
public class OracleDriver implements DataBaseDriver {  
  
    protected String url;  
    protected String user;  
    protected String password;  
    protected String driver;  
    protected Integer port;
```

```
    public String getUrl() {
        return url;
    }

    public String getUser() {
        return user;
    }

    public void setUser(String user) {
        this.user = user;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getDriver() {
        return driver;
    }

    public void setDriver(String driver) {
        this.driver = driver;
    }

    public Integer getPort() {
        return port;
    }

    public void setPort(Integer port) {
        this.port = port;
    }

    public void setUrl(String url) {
        this.url = url;
    }

    public String getInfo() {
        return "[ Driver: Oracle" +
               ", url: " + url +
               ", port; " + port +
               ", user: " + user +
```

```

        ", password: " + password +
        ", driver: " + driver +
        " ] ";
    }
}

```

`oracleDriver` is a simple bean, we will use service class to inject properties to this bean.

Spring Service Class

```

package com.journaldev.service;

import com.journaldev.drivers.DataBaseDriver;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Service;

@Service
public class UserService {

    @Autowired
    @Qualifier("oracleDriver")
    private DataBaseDriver DataBaseDriver;

    public String getDriverInfo() {
        return DataBaseDriver.getInfo();
    }
}

```

Here we are using `@Service` annotation to indicate Spring framework to treat this as a Service class. Then we are using `@Autowired` and `@Qualifier("oracleDriver")` annotations to tell spring framework to inject bean named `oracleDriver` to class property `DataBaseDriver`. Note that we haven't yet created this spring bean.

Spring Beans

Final step is to create our spring beans and configuration classes to glue everything together.

```

package com.journaldev.config;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;

```

```

import org.springframework.core.env.Environment;

import com.journaldev.drivers.DataBaseDriver;
import com.journaldev.drivers.MySqlDriver;
import com.journaldev.drivers.OracleDriver;

@Configuration
@ComponentScan("com.journaldev")
@PropertySource("classpath:oracledatabase.properties")
public class AppConfig {

    @Autowired
    Environment environment;

    @Bean
    DataBaseDriver oracleDriver() {
        OracleDriver oracleDriver = new OracleDriver();
        oracleDriver.setDriver(environment.getProperty("db.driver"));
        oracleDriver.setUrl(environment.getProperty("db.url"));

        oracleDriver.setPort(Integer.parseInt(environment.getProperty("db.port")));
        oracleDriver.setUser(environment.getProperty("db.user"));
        oracleDriver.setPassword(environment.getProperty("db.password"));

        return oracleDriver;
    }

    @Bean
    DataBaseDriver mysqlDriver() {
        return new MySqlDriver();
    }
}

```

Notice the bean definition for `oracleDriver`. In this method, we are reading properties from `oracledatabase.properties` file that is being set to `environment` variable by Spring framework.

Here is the properties defined in `oracledatabase.properties` file.

```
db.url=localhost
```

```
db.port=4444
db.user=vasiliy
db.password=yilisav
db.driver=driver name
```

Our spring annotations example project is ready to test. As a summary, we performed following steps:

1. Created maven project and added required spring dependencies.
2. Created component classes and inject properties from a resource file into it's variable.
3. If we have a third party component, we can use Service class to inject dependencies into it. Just like we did for OracleDriver through UserService class.
4. Finally, we created Configuration class to define [spring beans](#) and set the base package to scan for spring component classes and configure them.

Spring Annotations Example Testing

Here is our main class to test our Spring annotations example project.

```
package com.journaldev;

import
org.springframework.context.annotation.AnnotationConfigApplicationContext;
import
org.springframework.context.support.AbstractApplicationContext;

import com.journaldev.config.AppConfig;
import com.journaldev.drivers.DataBaseDriver;
import com.journaldev.service.UserService;

public class Main {
    public static void main(String[] args) {
        AbstractApplicationContext appContext = new
AnnotationConfigApplicationContext(AppConfig.class);

        DataBaseDriver oracle = appContext.getBean("oracleDriver",
DataBaseDriver.class);
        DataBaseDriver mysql = appContext.getBean("mysqlDriver",
DataBaseDriver.class);

        System.out.println("Oracle driver info:");
        System.out.println(oracle.getInfo());

        System.out.println("MySQL driver info:");
        System.out.println(mysql.getInfo());
    }
}
```

```
        System.out.println("UserService Information");
        UserService userService =
appContext.getBean(UserService.class);
        System.out.println(userService.getDriverInfo());

        appContext.close();
    }
}
```

Below image shows the output produced. Notice that we haven't configured any logging framework, so all the spring framework logging is getting printed into console in red color.

```
J Main.java ✘ J AppConfig.java J UserService.java J MySqlDriver.java J OracleDriver.java » [  
10 public class Main {  
11     public static void main(String[] args) {  
12         AbstractApplicationContext appContext = new AnnotationConfigApplicationContext(AppConfig.class);  
13  
14         DataBaseDriver oracle = appContext.getBean("oracleDriver", DataBaseDriver.class);  
15         DataBaseDriver mysql = appContext.getBean("mysqlDriver", DataBaseDriver.class);  
16  
17         System.out.println("Oracle driver info:");  
18         System.out.println(oracle.getInfo());  
19  
20         System.out.println("MySQL driver info:");  
21         System.out.println(mysql.getInfo());  
22  
23         System.out.println("UserService Information");  
24         UserService userService = appContext.getBean(UserService.class);  
25         System.out.println(userService.getDriverInfo());  
26  
27         appContext.close();  
28     }  
29 }  
30
```

```
Problems @ Javadoc Declaration Console ✘ Progress Call Hierarchy Error Log  
<terminated> Main [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/Home/bin/java (16-Nov-2017, 12:23:55 PM)  
Nov 16, 2017 12:23:55 PM org.springframework.context.annotation.AnnotationConfigApplicationContext prepareRefresh  
INFO: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@6576fe71: startup date  
Oracle driver info:  
[ Driver: Oracle, url: localhost, port: 4444, user: vasiliy, password: yilisav, driver: driver_name ]  
MySQL driver info:  
[ Driver: mySql, databaseName: school, disableStatementPooling: true ]  
UserService Information  
[ Driver: Oracle, url: localhost, port: 4444, user: vasiliy, password: yilisav, driver: driver_name ]  
Nov 16, 2017 12:23:55 PM org.springframework.context.annotation.AnnotationConfigApplicationContext doClose  
INFO: Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@6576fe71: startup date [T
```

25) Difference between SessionFactory and session.

Ans - Do we have any other differences other than the below? Also please validate whether the below are correct

1. `SessionFactory` objects are one per application and `Session` objects are one per client.
2. `SessionFactory` is to create and manage `Sessions`. `Session` is to provide a CRUD interface for mapped classes, and also access to the more versatile `Criteria API`.
3. `SessionFactory` is thread safe whereas `Session` is not thread safe

26) Difference between save and persist

Ans - `persist()` is well defined. It makes a transient instance persistent. However, it doesn't guarantee that the identifier value will be assigned to the persistent instance immediately, the assignment might happen at flush time. The spec doesn't say that, which is the problem I have with `persist()`.

`persist()` also guarantees that it will not execute an `INSERT` statement if it is called outside of transaction boundaries. This is useful in long-running conversations with an extended `Session/persistence context`.

A method like `persist()` is required.

`save()` does not guarantee the same, it returns an identifier, and if an `INSERT` has to be executed to get the identifier (e.g. "identity" generator, not "sequence"), this `INSERT` happens immediately, no matter if you are inside or outside of a transaction. This is not good in a long-running conversation with an extended `Session/persistence context`.

27) write a program how many words are repeated in your name.

Ans -

```
import java.io.*;
import java.util.*;

class Test {

    static int findRepeatFirstN2(String s)
    {

        // this is O(N^2) method
        int p = -1, i, j;
        for (i = 0; i < s.length(); i++)
        {
            for (j = i + 1; j < s.length(); j++)
            {
                if (s.charAt(i) == s.charAt(j))
                {
                    p = i;
                    break;
                }
            }
            if (p != -1)
                break;
        }
    }
}
```

```
        return p;
    }

// Driver code
static public void main (String[] args)
{
    String str = "geeksforgeeks";
    int pos = findRepeatFirstN2(str);

    if (pos == -1)
        System.out.println("Not found");
    else
        System.out.println( str.charAt(pos));
}
}

// This code is contributed by anuj_67.
```

Output:

Interview Questions at Genpact Headstrong for Java

Q. What will happen if you call return statement or System.exit on try or catch block? will finally block execute?

finally block will execute even if you put return statement in try block or catch block

finally block won't run if you call System.exit from try or catch.

Q. Can you override private or static method in Java?

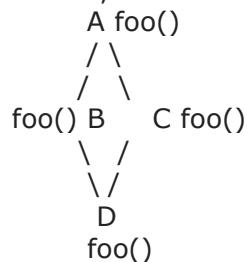
No. We cannot override private or static method in Java, if you create similar method with same return type and same method arguments that's called method hiding. (Parent and Child class concept).

Q. JSP Implicit Objects or Objects (Written Inside Scriptlets)?

- . Implicit object are the object that are created by web container provides to a developer to access them in their program using JavaBeans and Servlets.
 - . These objects are called implicit objects because they are automatically instantiated.
 - . They are by default available in JSP page.
1. Application
 2. config
 3. Exception
 4. out
 5. page
 6. PageContext
 7. request
 8. response
 9. session.

Q. Why Java doesn't support multiple inheritance?

1) First reason is ambiguity (not clear or decided) around Diamond problem, consider a class A has foo() method and then B and C derived from A and has their own foo() implementation and now class D derive from B and C using multiple inheritance and if we refer just foo() compiler will not be able to decide which foo() it should invoke. This is also called Diamond problem because structure on this inheritance scenario is similar to 4 edge diamond, see below



2). Second and more convincing reason to me is that multiple inheritances does complicate the design and creates problem during casting, constructor chaining etc. and given that there are not many scenario on which you need multiple inheritance its wise decision to omit it for the sake of simplicity. Also java avoids this ambiguity by supporting single inheritance with interfaces. Since interface only have method declaration and doesn't provide any implementation there will only be just one implementation of specific method

hence there would not be any ambiguity.

Q. Difference between == and equals?

- '==' check's if references are equal means if both have same address then it return's true.
- 'equals' check's value irrespective of address.
- Default implementation of equals() class provided by java.lang.Object compares memory location and only return true if two reference variable are pointing to same memory location i.e. essentially they are same object.

- 1) If two objects are equal by equals() method then there hashcode must be same.
- 2) If two objects are not equal by equals() method then there hashcode could be same or different.

Main difference between == and equals in Java is that "==" is used to compare primitives while equals() method is recommended to check equality of objects.

Another difference between them is that, If both "==" and equals() is used to compare objects than == returns true only if both references points to same object while equals() can return true or false based on its overridden implementation.

Ex. String str="abc";
String str1="abc";

Both case will return true.

Default Implementation of Equals

```
public boolean equals(Object obj) {  
    return (this == obj);  
}
```

Q. Why Synchronization in Java?

- Because Java is a multi -threaded language where multiple threads run in parallel to complete program execution.
- In multi-threaded environment synchronization of java object or synchronization of java class becomes extremely important.
- Synchronization in Java is possible by using java keyword "synchronized" and "volatile".
- Prevents access of multiple Threads to shared resource at a time.
- Synchronized is a modifier.

JVM guarantees that Java synchronized code will only be executed by one thread at a time.

- Volatile can be used only with variables.
- Synchronized only when applying block level and method level locks.
- With Block level, we need to pass Class as an agreement:

```
if (obj == null) {  
    synchronized (ClassName.class) //Mandatory{  
        obj = new Hello();  
    }  
}
```

Q. this keyword?

- Can be used to refer current class instance variable.
- `this()` can be used to invoke current class Constructor. And it should be the first statement. If we are calling in class constructor using first line, it is giving compilation error è recursive calling of constructor.

```
public class Test {  
    public Test() {  
        this(); // Recursive Constructor invocation Test()  
    } }  
· this can be used to invoke current class method.  
· this can be passed as an argument in method call.  
· this can be passed as an argument in Constructor call.  
· this() can be used to return current class instance.  
· Cannot be used in static context.
```

```
public class Test {  
    public Test() {  
  
    }  
  
    private void test(String name) {  
        this(); // Compile time error - Constructor call must be the first statement in  
constructor.  
    }  
  
}
```

Q. What are the different ways to create singleton ?

Answer

Q. super keyword in Java?

Java keyword that refers to the immediate parent class object.

Q : What are basic features of java ?

Abstraction
Polymorphism
Encapsulation

Q : Difference between JDK, JVM and JRE ?

Answer

Q : What is Garbage Collection. ?

Garbage Collection inside view

Q : Can System.gc() start the java garbage collection process?

Whether it will run the Garbage collector or not, cannot be confirmed. It is not guaranteed that it will start the garbage collection.

Data Structures and Collections

Q: How Hashmap works internally ?

[Reference 1](#)

[Reference 2](#)

Q : What is an iterator ?

[Java Docs](#)

Q : What is the difference between Iterator and List Iterator ?

[List Iterator](#)

[Iterator](#)

Q : How hashMap works internally in Java ? Can we safely use hashMap in multithreading environment ?

[Reference 1](#)

No we cannot use HashMap in multithreading environment.

Q : What will happen if we override only hashCode method but not equals method while creating my custom class (say Employee) whose instances we want to store as key in hashmap.

Overriding only hashCode method and not the equals method will not give any compile time or runtime error but it is a logical error. For example if you do not override equals method, then different Employee class instances will be compared by using equals method present in Object class and it will mark the objects as duplicate only when two objects are pointing to one instance of Employee class. However the comparison between two Employee objects should be based on custom parameters like employee id, name, designation etc. Comparison based on custom parameters is done by overriding the equals method along with hashCode method.

1. Implement equals and hashCode methods for demo class

. Class Demo { private String key; private int value; } 2. Interface Demo{ m1(); m2(); } Class A implements Demo{ m1(); m2(); m3(); } main(){ Demo object = new A(); object.m3(); } 3. What happens when Immutable class contains mutable reference as instance variable. How to avoid the issues with it. 4. Hashmap working , collision. 5.spring bean definitions 6. Caching in hibernate. 2nd round Types of class loaders. Rest and soap difference Jre and JVM How will be the out of memory will be tackled after reaching max size available Difference between logical and physical database Which design patterns are been used in your project What is build design pattern For implementing caching in application, which kind of collection do you use Difference between list and set Implementation of custom set in java What all required changes are done on class if it is being used as key in collection What need to be considered inside implementation of equals and hashCode Contact equals and hashCode. Explain executor framework. Difference between interface and abstract class. Why we explicitly give public static final for constants if JVM internally treats this as same in interface. What does final keyword actually does to an object - internal working How do you display duplicates with their count using collection.

Synchronous and Asynchronous Web Services

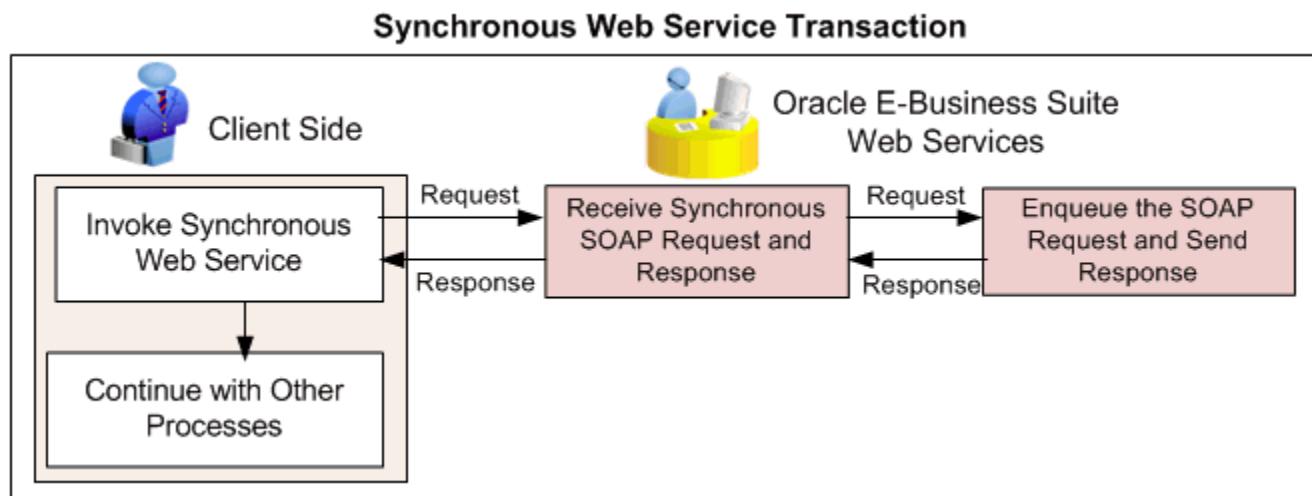
Synchronous and Asynchronous Web Services

Oracle E-Business Suite Integrated SOA Gateway supports both synchronous and asynchronous service processing and execution for SOAP-based services.

Interfaces exposed as REST services can be generated with the support for synchronous interaction pattern only. Asynchronous pattern for REST services is not supported in this release.

- **Synchronous Web Services**

This type of service execution provides an immediate response to a query. In this situation, the client will wait until the server sends back the response message. The advantage of using the synchronous service is that the client application knows the status of the Web service operation in a very short time.



When a Web service client sends a synchronous SOAP request to Oracle E-Business Suite service, the SOAP response will be sent back to the client as soon as the process completes.

- **Asynchronous Web Services (SOAP Web Services Only)**

This type of service execution may require a significant amount of time to process a request. However, the client that invoked the Oracle E-Business Suite

Web service can continue with other processing in the meantime rather than wait for the response.

Asynchronous operation is extremely useful for environments in which a service, such as a loan processor, can take a long time to process a client request.

In this release, asynchronous operation pattern is supported for SOAP-based Web services only.

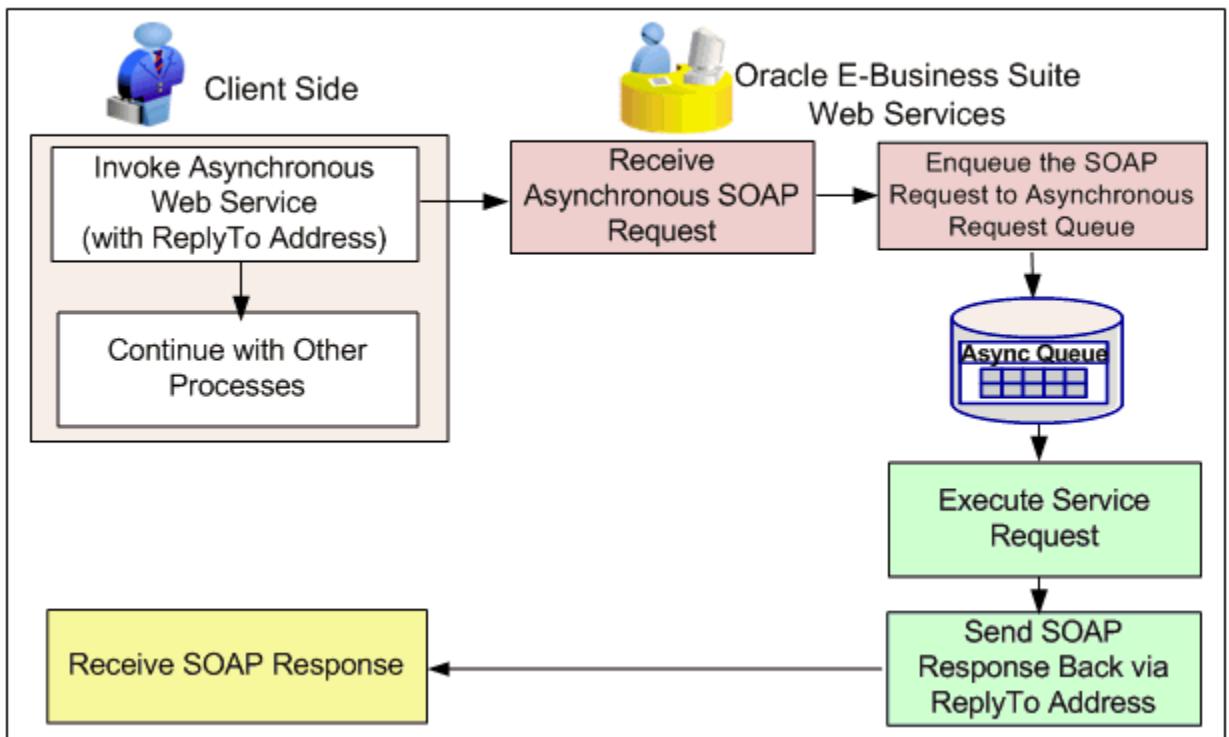
Callback without Acknowledgement

Callback pattern is a very important communication method in asynchronous services - a request is made to the service provider and a response is sent back to the requester when it is ready. This pattern can be used in conjunction with acknowledgement to recognize the receipt of a request sent by a requester. Only *callback without acknowledgement* pattern is supported in this release.

In callback without acknowledgment pattern, a SOAP Callback header becomes necessary when the Web service is asynchronous and the client contact information is unknown at deployment time. Callback header allows the client to specify how to contact the client (`ReplyTo` address) in the request for service. Therefore, client must publish a listener or a receive service. In other words, the structure of the WSDL dictates how the client will receive the response.

A Web service client must provide `MessageID`, and an appropriate callback endpoint address (`ReplyTo` and `FaultTo`) using WS-Addressing in SOA Headers for the asynchronous request callback pattern.

Asynchronous Web Service Transaction



When a Web service client sends a SOAP request to Oracle E-Business Suite service, on completion of service execution, the SOAP response (service response payload) is sent to ReplyTo address of the client. This pattern does not expect acknowledgment from client as it is a fire-and-forget message exchange pattern for callback.

SOAP services, depending on specified interaction patterns, can be generated synchronously, asynchronously, or both synchronously and asynchronously to meet your business needs. REST services can be generated with synchronous operation only.

Once a SOAP service has been generated and deployed to an Oracle SOA Suite WebLogic managed server, service consumers or Web service clients can send request messages through Oracle SOA Suite. After security checks on the inbound requests, Oracle E-Business Suite Web services can be invoked synchronously or asynchronously.

OR

Synchronization

Overview

In general, mobile applications need to be able to operate without constant network connectivity. This means the client app must synchronize data with the server application after a disconnected period. To do this:

- The client (browser) app replicates changes made in the server application.
- The server application replicates changes made in the client (browser) app.

This process is called *synchronization* (abbreviated as *sync*). Sync replicates data from the *source* to the *target*, and the target calls the LoopBack replication API.

Note: The LoopBack replication API is a JavaScript API, and thus (currently, at least) works only with a JavaScript client.

Replication means intelligently copying data from one location to another. LoopBack copies data that has changed from source to target, but does not overwrite data that was modified on the target since the last replication. So, sync is just bi-directional replication.

In general there may be conflicts when performing replication. So, for example, while disconnected, a user may make changes on the client that conflict with changes made on the server. What happens when an object or field is modified both locally and remotely? LoopBack handles conflict resolution for you, and enables you to easily present a user interface to allow the end user to make informed decisions to resolve conflicts when they occur. See [Resolving conflicts](#) below.

Note:

Currently synchronization is built-in to LoopBack, but will be refactored into a component in the future.

LoopBack in the browser

LoopBack implements synchronization using the LoopBack browser API, that provides the same client JavaScript API as for Node. Thus, LoopBack in the browser is sometimes referred to as *isomorphic*, because you can call exactly the same APIs on client and server.

LoopBack in the browser uses [Browserify](#) to handle dependencies. If you wish, you can use build tools such as [Gulp](#) or Grunt to generate the client API based on the back-end models and REST API. For example, [loopback-example-full-stack](#) uses Grunt.

Synchronization as described above to handle offline operation is called *offline sync*. LoopBack also provides the ability to consolidate (or “batch”) data changes the user makes on the device and send them to the server in a single HTTP request. This is called *online sync*.

Terminology

In addition to [standard terminology](#), conflict resolution uses a number of specific terms.

Change list

A list of the current and previous revisions of all models. Each data source has a unique change list.

Checkpoint

An orderable identifier for tracking the last time a source completed replication. Used for filtering the change list during replication.

Checkpoint list

An ordered list of replication checkpoints used by clients to filter out old changes.

Conflict

When replicating a change made to a source model, a conflict occurs when the source's previous revision differs from the target's current revision.

Rebasing

Conflicts can only be resolved by changing the revision they are based on. Once a source model is "rebased" on the current target version of a model, it is no longer a conflict and can be replicated normally.

Revision

A string that uniquely identifies the state of a model.

Setup

Setup involves three steps:

1. Enable change tracking in the LoopBack app.
2. Create a client app that uses the LoopBack API.
3. Run the client app in the browser

Enable change tracking

You must enable change tracking for each model that you want to be able to access offline. Make the following change to the [Model definition JSON file](#):

- Set `trackChanges` to `true`.
- Change the `id` property to an auto-generated GUID; for information on GUIDs, see [Model definition JSON file reference](#).
- Set `strict` property to `validate`.
- Set the `persistUndefinedAsNull` property to `true`.

For example:

common/models/todo.json

```
{  
  
  "name": "Todo",  
  
  "base": "PersistedModel",  
  
  "strict": "validate",  
  
  "trackChanges": true,  
  
  "persistUndefinedAsNull": true,  
  
  "properties" : {  
  
    "id": {  
  
      "id": true,  
  
      "type": "string",  
  
      "defaultFn": "guid"  
  
    },  
  
    "title": {  
  
      "type": "string",  
  
      "required": true  
  
    },  
  
    "description": {  
  
      "type": "string"  
  
    }  
  }  
}
```

```
}
```

For each change-tracked model, a new model (database table) is created to contain change-tracking records. In the example above, a `Todo-Change` model will be created. The change model is attached to the same data source as the model being tracked. Therefore, you will need to migrate your database schema after you have enabled change tracking.

The change-tracking records are updated in background. Any errors are reported via the static model method `handleChangeError`. It is recommended to provide a custom error handler in your models, as the default behavior is to throw an error.

common/models/todo.js

```
module.exports = function(Todo) {  
  
  Todo.handleChangeError = function(err) {  
  
    console.warn('Cannot update change records for Todo:', err);  
  
  };  
  
}
```

Create a client app

The next step is to create client-side LoopBack app. For each replicated model, create two new client-only subclasses:

- A local model that will use local storage to persist the changes offline
- A remote model that will be connected to the server and used as a target for replication. This model will have change tracking disabled (because the server is already handling it) and enable only the replication REST API.

For example, for the To Do example, here is the JSON file that defines the client local model:

client/models/local-todo.json

```
{  
  
  "name": "LocalTodo",  
  
  "base": "Todo"  
  
}
```

Here is the JSON file that defines the client remote local model:

client/models/remote-todo.json

```
{  
  
  "name": "RemoteTodo",  
  
  "base": "Todo",  
  
  "plural": "Todos",  
  
  "trackChanges": false,  
  
  "enableRemoteReplication": true  
  
}
```

And here is the client model configuration JSON file:

client/model-config.json

```
{  
  
  "_meta": {  
  
    "sources": [ "../common/models", "./models" ]  
  
  },  
  
  "RemoteTodo": {  
  
    "dataSource": "remote"  
  
  },  
  
  "LocalTodo": {  
  
    "dataSource": "local"  
  
  }  
  
}
```

Here is the JSON file that defines the client datasources:

client/datasources.json

```
{  
  
  "remote": {  
  
    "connector": "remote",  
  
    "url": "/api"  
  
  },  
  
  "local": {  
  
    "connector": "memory",  
  
    "localStorage": "todo-db"  
  
  }  
  
}
```

Now that you have all models in place, you can set up bi-directional replication between `LocalTodo` and `RemoteTodo`, for example in a client boot script:

client/boot/replication.js

```
module.exports = function(client) {  
  
  var LocalTodo = client.models.LocalTodo;  
  
  var RemoteTodo = client.models.RemoteTodo;  
  
  
  
  
  var since = { push: -1, pull: -1 };  
  
  
  
  
  
function sync() {
```

```
// It is important to push local changes first,  
  
// that way any conflicts are resolved at the client  
  
LocalTodo.replicate(  
  
    RemoteTodo,  
  
    since.push,  
  
    function pushed(err, conflicts, cps) {  
  
        // TODO: handle err  
  
        if (conflicts.length)  
  
            handleConflicts(conflicts);  
  
        since.push = cps;  
  
  
        RemoteTodo.replicate(  
  
            LocalTodo,  
  
            since.pull,  
  
            function pulled(err, conflicts, cps) {  
  
                // TODO: handle err  
  
                if (conflicts)  
  
                    handleConflicts(conflicts.map(function(c) { return c.swapParties(); }))  
;  
  
                since.pull = cps;  
  
            });
}
```

```

    });

}

LocalTodo.observe('after save', function(ctx, next) {
  next();
  sync(); // in background
});

LocalTodo.observe('after delete', function(ctx, next) {
  next();
  sync(); // in background
});

function handleConflicts(conflicts) {
  // TODO notify user about the conflicts
}

};

```

Run the client app in the browser

The loopback-boot module provides a build tool for adding all application metadata and model files to a Browserify bundle. [Browserify](#) is a tool that packages Node.js scripts into a single file that runs in a browser.

Below is a simplified example packaging the client application into a browser “module” that can be loaded via `require('lbclient')`. Consult [build.js](#) in `loopback-example-full-stack` for a full implementation that includes source-maps and error handling.

client/build.js

```
var b = browserify({ basedir: __dirname });

b.require('./client.js', { expose: 'lbclient' });

boot.compileToBrowserify({ appRootDir: __dirname }, b);

var bundlePath = path.resolve(__dirname, 'browser.bundle.js');

b.pipe(fs.createWriteStream(bundlePath));
```

Access control

Because the sync algorithm calls the REST API, it honors model access control settings.

However, when replicating changes only from the server (read-only replication), the client needs to create a new checkpoint value, which requires write permissions. The “REPLICATE” permission type supports this use case: it grants limited write access to the checkpoint-related methods only. For a certain user (a role, a group) to be able to pull changes from the server, they need both READ and REPLICATE permissions. Users with WRITE permissions are automatically granted REPLICATE permission too.

Example ACL configuration:

common/models/car.json

```
{
  "acls": [
    // disable anonymous access
    {
      "principalType": "ROLE",
      "principalId": "$everyone",
      "permission": "DENY"
    }
  ]
}
```

```
},  
  
    // allow all authenticated users to read data  
  
    {  
  
        "principalType": "ROLE",  
  
        "principalId": "$authenticated",  
  
        "permission": "ALLOW",  
  
        "accessType": "READ"  
  
    },  
  
    // allow all authenticated users to pull changes  
  
    {  
  
        "principalType": "ROLE",  
  
        "principalId": "$authenticated",  
  
        "permission": "ALLOW",  
  
        "accessType": "REPLICATE"  
  
    },  
  
    // allow the user with id 0 to perform full sync  
  
    {  
  
        "principalType": "USER",  
  
        "principalId": 0,  
  
        "permission": "ALLOW",  
  
        "accessType": "WRITE"  
    }  
}
```

```
    }

]

}
```

Understanding replication

Offline data access and synchronization has three components:

- Change tracking
- Replication of changes
- Browser version of LoopBack

Change model

As explained above, a new change model is created for each change-tracked model, e.g. Todo-Change. This model can be accessed using the method `getChangeModel`, for example, `Todo.getChangeModel()`.

The change model has several properties:

- `modelId` links a change instance (record) with a tracked model instance
- `prev` and `rev` are hash values generated from the model class the Change model is representing. The `rev` property stands for Revision, while `prev` is the hash of the previous revision. When a model instance is deleted, the value `null` is used instead of a hash.
- `checkpoint` associates a change record with a Checkpoint, more on this later.

Additionally, there is a method `type()` that can be used to determine the kind of change being made: `Change.CREATE`, `Change.UPDATE`, `Change.DELETE` or `Change.UNKNOWN`.

The current implementation of the change tracking algorithm keeps only one change record for each model instance - the last change made.

Checkpoints

A checkpoint represents a point in time that you can use to filter the changes to only those made after the checkpoint was created. A checkpoint is typically created whenever a replication is performed, this allows subsequent replication runs to ignore changes that were already replicated.

While in theory the replication algorithm should work without checkpoints, in practice it's important to use correct checkpoint values because the current implementation keeps the last change only.

If you don't pass correct values in the `since` argument of `replicate` method, then you may

- Get false conflicts if the “since” value is omitted or points to an older, already replicated checkpoint.
- Incorrectly override newer changes with old data if the “since” value points to a future checkpoint that was not replicated yet.

Replication algorithm

A single iteration of the replication algorithm consists of the following steps:

1. Create new checkpoints (both source and target)
2. Get list of changes made at the source since the given source checkpoint
3. Find out differences between source and target changes since the given target checkpoint, detect any conflicts.
4. Create a set of instructions - what to change at target
5. Perform a “bulk update” operation using these instructions
6. Return the new checkpoints to the callback

It is important to create the new checkpoints as the first step of the replication algorithm. Otherwise any changes made while the replication is in progress would be associated with the checkpoint being replicated, and thus they would not be picked up by the next replication run.

The consequence is that the “bulk update” operation will associate replicated changes with the new checkpoint, and thus these changes will be considered during the next replication run, which may cause false conflicts.

In order to prevent this problem, the method `replicate` runs several iterations of the replication algorithm, until either there is nothing left to replicate, or a maximum number of iterations is reached.

Conflict detection

Conflicts are detected in the third step. The list of source changes are sent to the target model, which compares them to changes made to target model instances. Whenever both source and target modified the same model instance (the same model id), the algorithm checks the current and previous revision of both source and target models to decide whether there is a conflict.

A conflict is reported when both of these conditions are met:

- The current revisions are different, i.e. the model instances have different property values.
- The *current* target revision is different from the *previous* source revision. In other words, if the source change is in sequence after the target change, then there is no conflict.

Conflict resolution

Conflict resolution can be complex. Fortunately, LoopBack handles the complexity for you, and provides an API to resolve conflicts intelligently.

The callback of `Model.replicate()` takes `err` and `conflict[]`. Each `conflict` represents a change that was not replicated and must be manually resolved. You can fetch the current versions of the local and remote models by calling `conflict.models()`. You can manually merge the conflict by modifying both models.

Calling `conflict.resolve()` will set the source change's previous revision to the current revision of the (conflicting) target change. Since the changes are no longer conflicting and appear as if the source change was based on the target, they will be replicated normally as part of the next `replicate()` call.

The conflict class provides methods implementing three most common resolution scenarios, consider using these methods instead of `conflict.resolve()`:

- `conflict.resolveUsingSource()`
- `conflict.resolveUsingTarget()`
- `conflict.resolveManually()`

Bulk update

The bulk update operation expects a list of instructions - changes to perform. Each instruction contains a `Change` instance describing the change, a change type, and model `data` to use.

In order to prevent race conditions when third parties are modifying the replicated instances while the replication is in progress, the `bulkUpdate` function is implementing a robust checks to ensure it modifies only those model instances that have their expected revision.

The “diff” step returns the current target revision of each model instances that needs an update, this revision is stored as the `change.rev` property.

The “bulkUpdate” method loads the model instance from the database, verifies that the current revision matched the expected revision in the instruction, and then performs a conditional update/delete specifying all model properties as the condition.

```
// Example: apply an update of an existing instance

var current = findById(data.id);

if (revisionOf(current) != expectedRev)

    return conflict();

var c = Model.updateAll(current, data);

if (c != 1) conflict();
```

Sync methods

The LoopBack Model object provides a number of methods to support sync, mixed in via the DataModel object:

- **bulkUpdate** - Apply an update list.
- **changes** - Get the changes to a model since a given checkpoint. Provide a filter object to reduce the number of results returned.
- **checkpoint** - Create a checkpoint.
- **createUpdates** - Create an update list for `Model.bulkUpdate()` from a delta list from `Change.diff()`.
- **currentCheckpoint** - Get the current checkpoint ID.
- **diff** - Get a set of deltas and conflicts since the given checkpoint.
- **enableChangeTracking** - Start tracking changes made to the model.
- **getChangeModel** - Get the `Change` model.
- **getSourceId** - Get the source identifier for this model / dataSource.
- **handleChangeError** - Handle a change error. Override this method in a subclassing model to customize change error handling.
- **rectifyChange** - Tell LoopBack that a change to the model with the given ID has occurred.
- **replicate** - Replicate changes since the given checkpoint to the given target model.
- **findLastChange** - Get the last (current) Change object for a given model instance.
- **updateLastChange** - Update the last (current) Change object associated with the given model instance.

Frequently asked questions

Does LoopBack support continuous replication?

Yes: with continuous replication, the client immediately triggers a replication when local data changes and the server pushes changes when they occur.

Here is a basic example that relies on a `socket.io` style `EventEmitter`.

```
// psuedo-server.js

MyModel.on('changed', function(obj) {

  socket.emit('changed');

});
```

```
// psuedo-client.js

socket.on('changed', function(obj) {
    LocalModel.replicate(RemoteModel);
});
```

How do you trigger immediate replication?

Call `Model.replicate()` to trigger immediate replication.

Known issues

- The size of the browser bundle is over 1.4MB, which is too large for mobile clients. See <https://github.com/strongloop/loopback/issues/989>.
- It's not possible to set a model property to `undefined` via the replication. When a property is `undefined` at the source but defined at the target, "bulk update" will not set it to `undefined` at the target. This can be mitigated by using `strict` model and enabling `persistUndefinedAsNull`.
- Browser's `localStorage` limits the size of stored data to about 5MB (depending on the browser). If your application needs to store more data in offline mode, then you need to use `IndexedDB` instead of `localStorage`. LoopBack does not provide a connector for `IndexedDB` yet. See <https://github.com/strongloop/loopback/issues/858>.
- Not all connectors were updated to report the number of rows affected by `updateAll` and `deleteAll`, which is needed by "bulkUpdate". As a result, the replication fails when the target model is persisted using one of these unsupported connectors.
- LoopBack does not fully support fine-grained access control to a selected subset of model instances, therefore it is not possible to replicate models where the user can access only a subset of instances (for example only the instances the user has created).

1. Question 1. What Is Javascript Objects?

Answer :

JavaScript is an Object Oriented Programming (OOP) language. A programming language can be called object-oriented if it provides four basic capabilities to developers –

- Encapsulation – the capability to store related information, whether data or methods, together in an object.
- Aggregation – the capability to store one object inside another object.
- Inheritance – the capability of a class to rely upon another class (or number of classes) for some of its properties and methods.
- Polymorphism – the capability to write one function or method that works in a variety of different ways.

Objects are composed of attributes. If an attribute contains a function, it is considered to be a method of the object, otherwise the attribute is considered a property.

2. Question 2. What Are Object Properties?

Answer :

Object properties can be any of the three primitive data types, or any of the abstract data types, such as another object. Object properties are usually variables that are used internally in the object's methods, but can also be globally visible variables that are used.

The syntax for adding a property to an object is:

```
objectName.objectProperty = propertyName;
```

For example – The following code gets the document title using the "title" property of the document object.

```
var str = document.title;
```

3. Question 3. What Is Object Methods?

Answer :

Methods are the functions that let the object do something or let something be done to it. There is a small difference between a function and a method – at a function is a standalone unit of statements and a method is attached to an object and can be referenced by the this keyword.

Methods are useful for everything from displaying the contents of the object to the screen to performing complex mathematical operations on a group of local properties and parameters.

For example – Following is a simple example to show how to use the write() method of document object to write any content on the document.

```
document.write("This is test");
```

4. Question 4. What Is User-defined Objects?

Answer :

User-Defined Objects: All user-defined objects and built-in objects are descendants of an object called Object.

The new Operator: The new operator is used to create an instance of an object. To create an object, the new operator is followed by the constructor method.

In the following example, the constructor methods are Object(), Array(), and Date(). These constructors are built-in JavaScript functions.

```
var employee = new Object();
var books = new Array("C++", "Perl", "Java");
var day = new Date("August 15, 1947");
```

The Object() Constructor: A constructor is a function that creates and initializes an object. JavaScript provides a special constructor function called Object() to build the object. The return value of the Object() constructor is assigned to a variable.

The variable contains a reference to the new object. The properties assigned to the object are not variables and are not defined with the var keyword.

Example 1: Try the following example; it demonstrates how to create an Object.

```
<html>
  <head>
    <title>User-defined objects</title>

    <script type="text/javascript">
      var book = new Object(); // Create the object
      book.subject = "Perl"; // Assign properties to the object
      book.author = "Mohtashim";
    </script>

  </head>

  <body>
    <script type="text/javascript">
      document.write("Book name is : " + book.subject + "<br>");
      document.write("Book author is : " + book.author + "<br>");
    </script>

  </body>
</html>
```

5. Question 5. Defining Methods For An Object?

Answer :

The previous examples demonstrate how the constructor creates the object and assigns properties. But we need to complete the definition of an object by assigning methods to it.

Example: Try the following example; it shows how to add a function along with an object.

```
<html>
  <head>
    <title>User-defined objects</title>

    <script type="text/javascript">
      // Define a function which will work as a method
```

```

function addPrice(amount){
    this.price = amount;
}

function book(title, author){
    this.title = title;
    this.author = author;
    this.addPrice = addPrice; // Assign that method as property.
}
</script>

</head>
<body>

<script type="text/javascript">
var myBook = new book("Perl", "Mohtashim");
myBook.addPrice(100);

document.write("Book title is : " + myBook.title + "<br>");
document.write("Book author is : " + myBook.author + "<br>");
document.write("Book price is : " + myBook.price + "<br>");
</script>

</body>
</html>

```

6. Question 6. What Is The 'with' Keyword?

Answer :

The ‘with’ keyword is used as a kind of shorthand for referencing an object's properties or methods.

The object specified as an argument to with becomes the default object for the duration of the block that follows. The properties and methods for the object can be used without naming the object.

Syntax: The syntax for with object is as follows --

```

with (object){
    properties used without the object name and dot
}
```

7. Question 7. What Are The Javascript Native Objects?

Answer :

JavaScript has several built-in or native objects. These objects are accessible anywhere in your program and will work the same way in any browser running in any operating system.

Here is the list of all important JavaScript Native Objects –

- JavaScript Number Object
- JavaScript Boolean Object
- JavaScript String Object

- JavaScript Array Object
 - JavaScript Date Object
 - JavaScript Math Object
 - JavaScript RegExp Object
8. **Question 8. What Is Javascript Number Object?**

Answer :

The Number object represents numerical date, either integers or floating-point numbers. In general, you do not need to worry about Number objects because the browser automatically converts number literals to instances of the number class.

Syntax: The syntax for creating a number object is as follows:

```
var val = new Number(number);
```

In the place of number, if you provide any non-number argument, then the argument cannot be converted into a number, it returns NaN (Not-a-Number).

9. **Question 9. What Are The Number Properties?**

Answer :

Here is a list of each property and their description.

- MAX_VALUE: The largest possible value a number in JavaScript can have 1.7976931348623157E+308
- MIN_VALUE: The smallest possible value a number in JavaScript can have 5E-324
- NaN: Equal to a value that is not a number.
- NEGATIVE_INFINITY: A value that is less than MIN_VALUE.
- POSITIVE_INFINITY: A value that is greater than MAX_VALUE
- prototype: A static property of the Number object. Use the prototype property to assign new properties and methods to the Number object in the current document
- constructor: Returns the function that created this object's instance. By default this is the Number object.

10. **Question 10. What Are The Number Methods?**

Answer :

The Number object contains only the default methods that are a part of every object's definition.

- toExponential(): Forces a number to display in exponential notation, even if the number is in the range in which JavaScript normally uses standard notation.
- toFixed(): Formats a number with a specific number of digits to the right of the decimal.
- toLocaleString(): Returns a string value version of the current number in a format that may vary according to a browser's local settings.
- toPrecision(): Defines how many total digits (including digits to the left and right of the decimal) to display of a number.
- toString(): Returns the string representation of the number's value.
- valueOf(): Returns the number's value.

11. **Question 11. What Is Javascript Boolean Object?**

Answer :

The Boolean object represents two values, either "true" or "false". If value parameter is omitted or is 0, -0, null, false, NaN, undefined, or the empty string (""), the object has an initial value of false.

Syntax: Use the following syntax to create a boolean object.

```
var val = new Boolean(value);
```

12. Question 12. What Are The Boolean Properties?

Answer :

Here is a list of the properties of Boolean object:

- constructor: Returns a reference to the Boolean function that created the object.
- prototype: The prototype property allows you to add properties and methods to an object.

13. Question 13. What Are Boolean Methods?

Answer :

Here is a list of the methods of Boolean object and their description.

- toSource(): Returns a string containing the source of the Boolean object; you can use this string to create an equivalent object.
- toString(): Returns a string of either "true" or "false" depending upon the value of the object.
- valueOf(): Returns the primitive value of the Boolean object.

14. Question 14. What Is Javascript Strings Object?

Answer :

The String object lets you work with a series of characters; it wraps Javascript's string primitive data type with a number of helper methods.

As JavaScript automatically converts between string primitives and String objects, you can call any of the helper methods of the String object on a string primitive.

Syntax: Use the following syntax to create a String object –

```
var val = new String(string);
```

The String parameter is a series of characters that has been properly encoded.

15. Question 15. What Are String Properties?

Answer :

Here is a list of the properties of String object and their description.

- constructor: Returns a reference to the String function that created the object.
- length: Returns the length of the string.
- prototype: The prototype property allows you to add properties and methods to an object.

16. Question 16. What Are String Methods?

Answer :

Here is a list of the methods available in String object along with their description.

- charAt(): Returns the character at the specified index.
- charCodeAt(): Returns a number indicating the Unicode value of the character at the given index.

- concat(): Combines the text of two strings and returns a new string.
- indexOf(): Returns the index within the calling String object of the first occurrence of the specified value, or -1 if not found.
- lastIndexOf(): Returns the index within the calling String object of the last occurrence of the specified value, or -1 if not found.
- localeCompare(): Returns a number indicating whether a reference string comes before or after or is the same as the given string in sort order.
- match(): Used to match a regular expression against a string.
- replace(): Used to find a match between a regular expression and a string, and to replace the matched substring with a new substring.
- search(): Executes the search for a match between a regular expression and a specified string.
- slice(): Extracts a section of a string and returns a new string.
- split(): Splits a String object into an array of strings by separating the string into substrings.
- substr(): Returns the characters in a string beginning at the specified location through the specified number of characters.
- substring(): Returns the characters in a string between two indexes into the string.
- toLocaleLowerCase(): The characters within a string are converted to lower case while respecting the current locale.
- toLocaleUpperCase(): The characters within a string are converted to upper case while respecting the current locale.
- toLowerCase(): Returns the calling string value converted to lower case.
- toString(): Returns a string representing the specified object.
- toUpperCase(): Returns the calling string value converted to uppercase.
- valueOf(): Returns the primitive value of the specified object.

17. Question 17. What Are String Html Wrappers?

Answer :

Here is a list of the methods that return a copy of the string wrapped inside an appropriate HTML tag.

- anchor(): Creates an HTML anchor that is used as a hypertext target.
- big(): Creates a string to be displayed in a big font as if it were in a tag.
- blink(): Creates a string to blink as if it were in a tag.
- bold(): Creates a string to be displayed as bold as if it were in a **tag**.
- fixed(): Causes a string to be displayed in fixed-pitch font as if it were in a tag
- fontcolor(): Causes a string to be displayed in the specified color as if it were in a tag.
- fontsize(): Causes a string to be displayed in the specified font size as if it were in a tag.
- italics(): Causes a string to be italic, as if it were in an *tag*.
- link(): Creates an HTML hypertext link that requests another URL.
- small(): Causes a string to be displayed in a small font, as if it were in a tag.
- strike(): Causes a string to be displayed as struck-out text, as if it were in a ~~tag~~.

- `sub()`: Causes a string to be displayed as a subscript, as if it were in a `sub` tag
- `sup()`: Causes a string to be displayed as a superscript, as if it were in a `sup` tag

18. Question 18. What Is Javascript Arrays Object?

Answer :

The Array object lets you store multiple values in a single variable. It stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Syntax: Use the following syntax to create an Array object:

```
var fruits = new Array( "apple", "orange", "mango" );
```

The Array parameter is a list of strings or integers. When you specify a single numeric parameter with the Array constructor, you specify the initial length of the array. The maximum length allowed for an array is 4,294,967,295.

You can create array by simply assigning values as follows:

```
var fruits = [ "apple", "orange", "mango" ];
```

You will use ordinal numbers to access and to set values inside an array as follows.

`fruits[0]` is the first element

`fruits[1]` is the second element

`fruits[2]` is the third element

19. Question 19. What Are Array Properties?

Answer :

Here is a list of the properties of the Array object along with their description.

- `constructor`: Returns a reference to the array function that created the object.
- `index`: The property represents the zero-based index of the match in the string
- `input`: This property is only present in arrays created by regular expression matches.
- `length`: Reflects the number of elements in an array.
- `prototype`: The prototype property allows you to add properties and methods to an object.

20. Question 20. What Are Array Methods?

Answer :

Here is a list of the methods of the Array object along with their description.

- `concat()`: Returns a new array comprised of this array joined with other array(s) and/or value(s).
- `every()`: Returns true if every element in this array satisfies the provided testing function.
- `filter()`: Creates a new array with all of the elements of this array for which the provided filtering function returns true.
- `forEach()`: Calls a function for each element in the array.
- `indexOf()`: Returns the first (least) index of an element within the array equal to the specified value, or -1 if none is found.
- `join()`: Joins all elements of an array into a string.

- `lastIndexOf()`: Returns the last (greatest) index of an element within the array equal to the specified value, or -1 if none is found.
- `map()`: Creates a new array with the results of calling a provided function on every element in this array.
- `pop()`: Removes the last element from an array and returns that element.
- `push()`: Adds one or more elements to the end of an array and returns the new length of the array.
- `reduce()`: Apply a function simultaneously against two values of the array (from left-to-right) as to reduce it to a single value.
- `reduceRight()`: Apply a function simultaneously against two values of the array (from right-to-left) as to reduce it to a single value.
- `reverse()`: Reverses the order of the elements of an array -- the first becomes the last, and the last becomes the first.
- `shift()`: Removes the first element from an array and returns that element.
- `slice()`: Extracts a section of an array and returns a new array.
- `some()`: Returns true if at least one element in this array satisfies the provided testing function.
- `toSource()`: Represents the source code of an object
- `sort()`: Sorts the elements of an array
- `splice()`: Adds and/or removes elements from an array.
- `toString()`: Returns a string representing the array and its elements.
- `unshift()`: Adds one or more elements to the front of an array and returns the new length of the array.

21. Question 21. What Is Javascript Date Object?

Answer :

The Date object is a datatype built into the JavaScript language. Date objects are created with the new `Date()` as shown below.

Once a Date object is created, a number of methods allow you to operate on it. Most methods simply allow you to get and set the year, month, day, hour, minute, second, and millisecond fields of the object, using either local time or UTC (universal, or GMT) time.

The ECMAScript standard requires the Date object to be able to represent any date and time, to millisecond precision, within 100 million days before or after 1/1/1970. This is a range of plus or minus 273,785 years, so JavaScript can represent date and time till the year 275755.

Syntax: You can use any of the following syntaxes to create a Date object using `Date()` constructor.

```
new Date( )
new Date(milliseconds)
new Date(datestring)
new Date(year,month,date[,hour,minute,second,millisecond ])
```

22. Question 22. What Is Date Properties?

Answer :

Here is a list of the properties of the Date object along with their description.

- `constructor`: Specifies the function that creates an object's prototype.

- prototype: The prototype property allows you to add properties and methods to an object.

23. Question 23. What Are Date Methods?

Answer :

Here is a list of the methods used with Date and their description.

- Date(): Returns today's date and time
- getDate(): Returns the day of the month for the specified date according to local time.
- getDay(): Returns the day of the week for the specified date according to local time.
- getFullYear(): Returns the year of the specified date according to local time.
- getHours(): Returns the hour in the specified date according to local time.
- getMilliseconds(): Returns the milliseconds in the specified date according to local time.
- getMinutes(): Returns the minutes in the specified date according to local time.
- getMonth(): Returns the month in the specified date according to local time.
- getSeconds(): Returns the seconds in the specified date according to local time.
- getTime(): Returns the numeric value of the specified date as the number of milliseconds since January 1, 1970, 00:00:00 UTC.
- getTimezoneOffset(): Returns the time-zone offset in minutes for the current locale.
- getUTCDate(): Returns the day (date) of the month in the specified date according to universal time.
- getUTCDate(): Returns the day of the week in the specified date according to universal time.
- getUTCFullYear(): Returns the year in the specified date according to universal time.
- getUTCHours(): Returns the hours in the specified date according to universal time.
- getUTCMilliseconds(): Returns the milliseconds in the specified date according to universal time.
- getUTCMinutes(): Returns the minutes in the specified date according to universal time.
- getUTCMonth(): Returns the month in the specified date according to universal time.
- getUTCSeconds(): Returns the seconds in the specified date according to universal time.
- getYear(): Deprecated - Returns the year in the specified date according to local time. Use getFullYear instead.
- setDate(): Sets the day of the month for a specified date according to local time.
- setFullYear(): Sets the full year for a specified date according to local time.
- setHours(): Sets the hours for a specified date according to local time.

- `setMilliseconds()`: Sets the milliseconds for a specified date according to local time.
- `setMinutes()`: Sets the minutes for a specified date according to local time.
- `setMonth()`: Sets the month for a specified date according to local time.
- `setSeconds()`: Sets the seconds for a specified date according to local time.
- `setTime()`: Sets the Date object to the time represented by a number of milliseconds since January 1, 1970, 00:00:00 UTC.
- `setUTCDate()`: Sets the day of the month for a specified date according to universal time.
- `setUTCFullYear()`: Sets the full year for a specified date according to universal time.
- `setUTCHours()`: Sets the hour for a specified date according to universal time.
- `setUTCMilliseconds()`: Sets the milliseconds for a specified date according to universal time.
- `setUTCMilliseconds()`: Sets the minutes for a specified date according to universal time.
- `setUTCMonth()`: Sets the month for a specified date according to universal time.
- `setUTCSeconds()`: Sets the seconds for a specified date according to universal time.
- `setYear()`: Deprecated - Sets the year for a specified date according to local time. Use `setFullYear` instead.
- `toDateString()`: Returns the "date" portion of the Date as a human-readable string.
- `toGMTString()`: Deprecated - Converts a date to a string, using the Internet GMT conventions. Use `toUTCString` instead.
- `toLocaleDateString()`: Returns the "date" portion of the Date as a string, using the current locale's conventions.
- `toLocaleFormat()`: Converts a date to a string, using a format string.
- `toLocaleString()`: Converts a date to a string, using the current locale's conventions.
- `toLocaleTimeString()`: Returns the "time" portion of the Date as a string, using the current locale's conventions.
- `toSource()`: Returns a string representing the source for an equivalent Date object; you can use this value to create a new object.
- `toString()`: Returns a string representing the specified Date object.
- `toTimeString()`: Returns the "time" portion of the Date as a human-readable string.
- `toUTCString()`: Converts a date to a string, using the universal time convention.
- `valueOf()`: Returns the primitive value of a Date object.

24. Question 24. What Are Date Static Methods?

Answer :

In addition to the many instance methods listed previously, the Date object also defines two static methods. These methods are invoked through the Date() constructor itself.

Date.parse(): Parses a string representation of a date and time and returns the internal millisecond representation of that date.

Date.UTC(): Returns the millisecond representation of the specified UTC date and time.

25. Question 25. What Is Javascript Math Object?

Answer :

The math object provides you properties and methods for mathematical constants and functions. Unlike other global objects, Math is not a constructor. All the properties and methods of Math are static and can be called by using Math as an object without creating it.

Thus, you refer to the constant pi as Math.PI and you call the sine function as Math.sin(x), where x is the method's argument.

Syntax: The syntax to call the properties and methods of Math are as follows

```
var pi_val = Math.PI;  
var sine_val = Math.sin(30);
```

26. Question 26. What Are The Math Properties?

Answer :

Here is a list of all the properties of Math and their description.

- E : Euler's constant and the base of natural logarithms, approximately 2.718.
- LN2: Natural logarithm of 2, approximately 0.693.
- LN10: Natural logarithm of 10, approximately 2.302.
- LOG2E: Base 2 logarithm of E, approximately 1.442.
- LOG10E: Base 10 logarithm of E, approximately 0.434.
- PI: Ratio of the circumference of a circle to its diameter, approximately 3.14159.
- SQRT1_2: Square root of 1/2; equivalently, 1 over the square root of 2, approximately 0.707.
- SQRT2: Square root of 2, approximately 1.414.

27. Question 27. What Are The Math Methods?

Answer :

Here is a list of the methods associated with Math object and their description

- abs(): Returns the absolute value of a number.
- acos(): Returns the arccosine (in radians) of a number.
- asin(): Returns the arcsine (in radians) of a number.
- atan(): Returns the arctangent (in radians) of a number.
- atan2(): Returns the arctangent of the quotient of its arguments.
- ceil(): Returns the smallest integer greater than or equal to a number.
- cos(): Returns the cosine of a number.
- exp(): Returns EN, where N is the argument, and E is Euler's constant, the base of the natural logarithm.
- floor(): Returns the largest integer less than or equal to a number.

- log(): Returns the natural logarithm (base E) of a number.
- max(): Returns the largest of zero or more numbers.
- min(): Returns the smallest of zero or more numbers.
- pow(): Returns base to the exponent power, that is, base exponent.
- random(): Returns a pseudo-random number between 0 and 1.
- round(): Returns the value of a number rounded to the nearest integer.
- sin(): Returns the sine of a number.
- sqrt(): Returns the square root of a number.
- tan(): Returns the tangent of a number.
- toSource(): Returns the string "Math".

28. Question 28. What Are Regular Expressions And Regexp Object?

Answer :

A regular expression is an object that describes a pattern of characters.

The JavaScript RegExp class represents regular expressions, and both String and RegExp define methods that use regular expressions to perform powerful pattern-matching and search-and-replace functions on text.

Syntax: A regular expression could be defined with the RegExp () constructor, as follows –

```
var pattern = new RegExp(pattern, attributes);
```

or simply

```
var pattern = /pattern/attributes;
```

Here is the description of the parameters:

- pattern – A string that specifies the pattern of the regular expression or another regular expression.
- attributes – An optional string containing any of the "g", "i", and "m" attributes that specify global, case-insensitive, and multiline matches, respectively.

29. Question 29. What Are Brackets?

Answer :

Brackets ([]) have a special meaning when used in the context of regular expressions. They are used to find a range of characters.

- [...]: Any one character between the brackets.
- [...] : Any one character not between the brackets.
- [0-9]: It matches any decimal digit from 0 through 9.
- [a-z]: It matches any character from lowercase a through lowercase z.
- [A-Z]: It matches any character from uppercase A through uppercase Z.
- [a-Z]: It matches any character from lowercase a through uppercase Z.

The ranges shown above are general; you could also use the range [0-3] to match any decimal digit ranging from 0 through 3, or the range [b-v] to match any lowercase character ranging from b through v.

30. Question 30. What Are Quantifiers?

Answer :

The frequency or position of bracketed character sequences and single characters can be denoted by a special character. Each special character has a specific connotation.

The +, *, ?, and \$ flags all follow a character sequence.

- p^+ : It matches any string containing one or more p's.
- p^* : It matches any string containing zero or more p's.
- $p^?$: It matches any string containing at most one p.
- $p^{\{N\}}$: It matches any string containing a sequence of N p's
- $p^{\{2,3\}}$: It matches any string containing a sequence of two or three p's.
- $p^{\{2,\}}$: It matches any string containing a sequence of at least two p's.
- $p\$$: It matches any string with p at the end of it.
- p : It matches any string with p at the beginning of it.

JUnit Concepts

JUnit is a program that can be used to perform unit testing of software by writing test cases in Java. A common use for JUnit is to create a set of unit tests that can be run automatically when changes are made to software; in this way, developers can ensure that changes to the software they are creating do not break things that were previously functioning. There is even a method of development known as [test-driven development](#) (TDD) that advocates writing unit tests even before writing the software under test itself. JUnit also provides a test runner that is capable of running the unit tests and reporting on the success or failure of the tests.

Some common terms that you may encounter when reading about JUnit include

- **Test method:** a method in a Java class that contains a single unit test.
- **Test class:** a Java class containing one or more test methods.
- **Assertion:** a statement that you include in a test method to check that the results of a test are as expected.
- **Test fixture:** a class that is used to set up state for multiple tests; typically used when the set up routines are “expensive” or take a long time to execute.
- **Test suite:** a grouping of test classes that are run together.

Installing the JUnit Extensions

Oracle JDeveloper 11g uses extensions to provide JUnit support. To install the extensions, perform the following steps:

1. From the Oracle JDeveloper **Help** menu, choose **Check for Updates....**

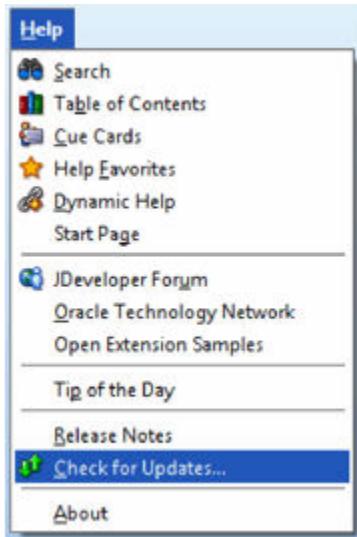


Figure 1 Checking for Updates

2. Ensure that, at a minimum, you have selected the “Official Oracle Extensions and Updates” update center, and click **Next**:

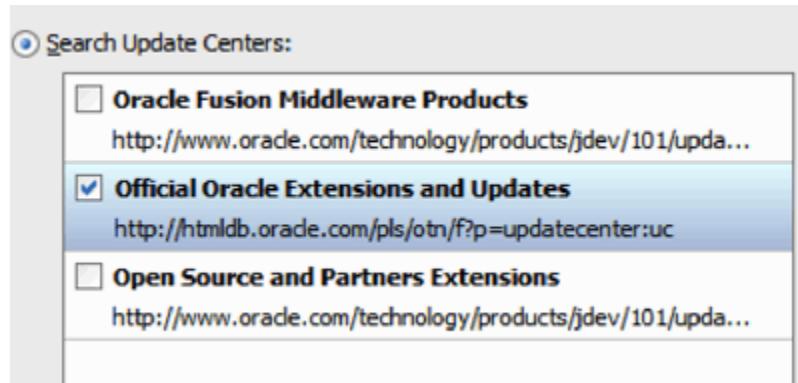


Figure 2 Update Centers

3. Type **JUnit** in the search box to search for the updates. At a minimum, include the BC4J JUnit Integration and JUnit Integration extensions, and click **Next**:

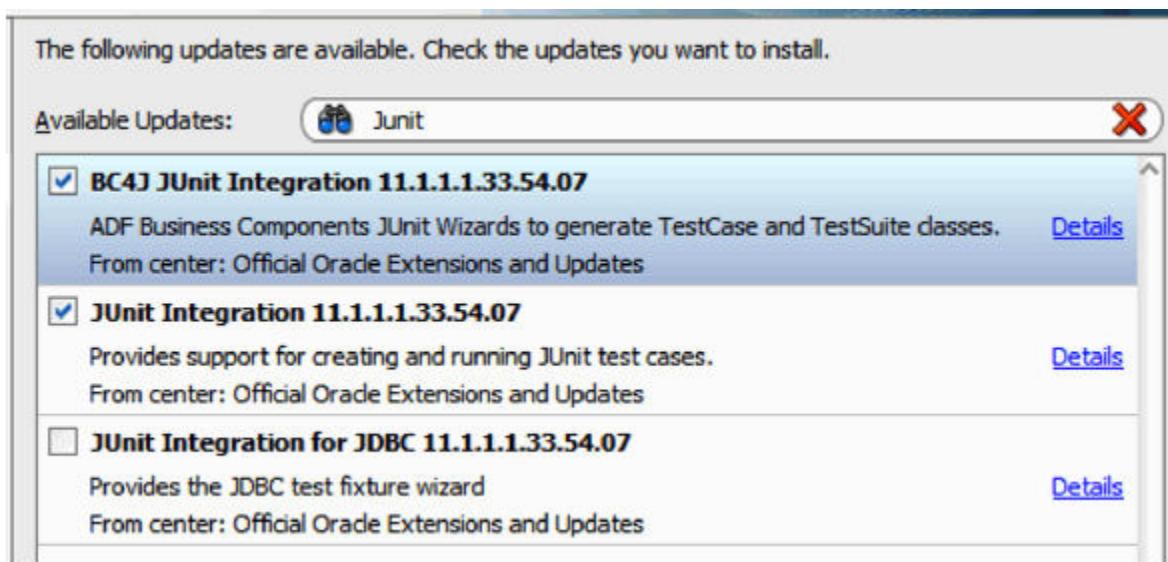


Figure 3 Selecting the JUnit Extensions

4. Agree to the license agreement and click **Next**. Provide your Oracle Technology Network (OTN) username and password if prompted. Oracle JDeveloper will download and install the extensions..
5. When prompted, click **Finish** to close the wizard and allow Oracle JDeveloper to restart in order to finish installing the extensions. Answer "No" when asked if you want to migrate settings from a previous version.

What Should I Test?

Now that the JUnit extensions are installed, you are ready to start creating unit tests. The first thing you might ask yourself is, "For what should I write a unit test?" For Web applications such as the one you have been using throughout this article series, that is often one of the more difficult questions to answer. In general, unit tests should have the following characteristics.

- They test a small bit (or "unit") of code. If the tests test "too much," then they become less useful, as it will not be clear which bit of code is causing the test to fail.
- They do not depend on external resources such as databases. The reason for this is so that the tests can be run in a variety of environments (for example, the IDE or build farms) and so that multiple concurrent executions of the unit tests do not interfere with each other.

- They should run quickly. This is to encourage the tests to be run as often as possible, even as often as every compile. If the tests take too long to execute, developers will be less likely to run them often.

For our simple application (a Web application used to maintain tables in a database), the first two of these general characteristics are somewhat difficult to achieve. In fact, the [Wikipedia article on test-drive development](#) acknowledges this: “Test-driven development is difficult to use in situations where full functional tests are required to determine success or failure. Examples of these are user interfaces, programs that work with databases....”

Although TDD advocates the use of “mock” objects to remove dependencies on external resources, a common compromise is to allow the use of a real database for unit tests. My experience (which has been mostly around writing enterprise applications; as a general rule, they have some type of database) has shown that this is a useful compromise. I try to write my unit tests so that they do not depend upon a specific database state, or if they do, the unit tests should create that expected state themselves. A later article in this series will show how to write functional (Web user interface) tests that can be automated just as JUnit tests can be.

A good starting point for writing JUnit tests in an Oracle ADF application is in the Model layer. We can write unit tests to ensure that expected View Object instances are available from the appropriate application modules and check that our data validation rules are correctly accepting valid data and rejecting invalid data.

Creating the Unit Test Project

Because keeping unit tests separate from the rest of the application code will make it easier to create deployment artifacts such as EAR files that do not include the unnecessary-for-deployment unit tests, I usually create a separate project to hold the unit tests. Because we are going to create unit tests for our Model (Business Components) project, let us create a new project called “ModelTests” to hold the unit tests. With the otnapp application open in the Oracle JDeveloper IDE, choose **New Project** from the Application Navigator shortcut menu:

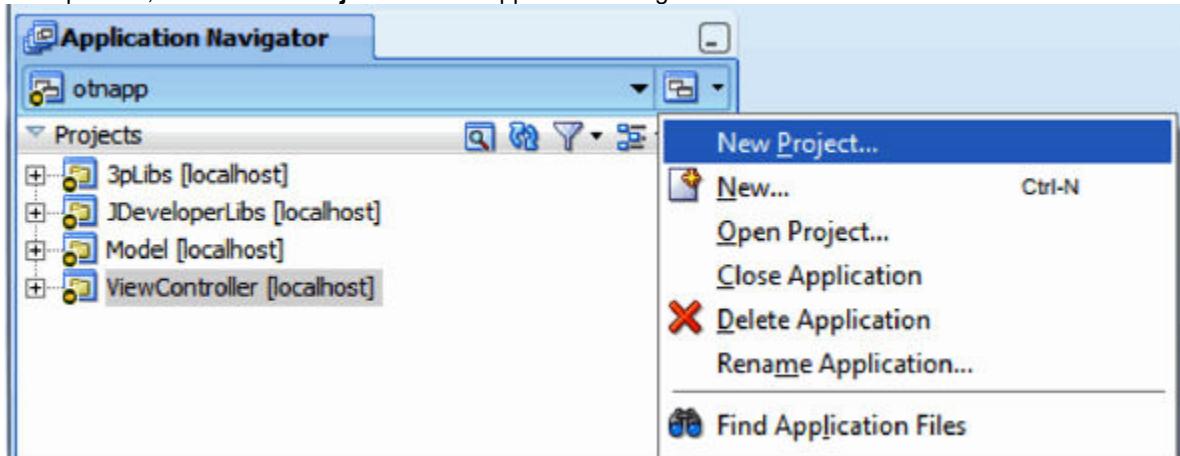


Figure 4 Creating a New Project

Leave **Generic Project** selected in the **New Gallery** dialog box and click **OK**:

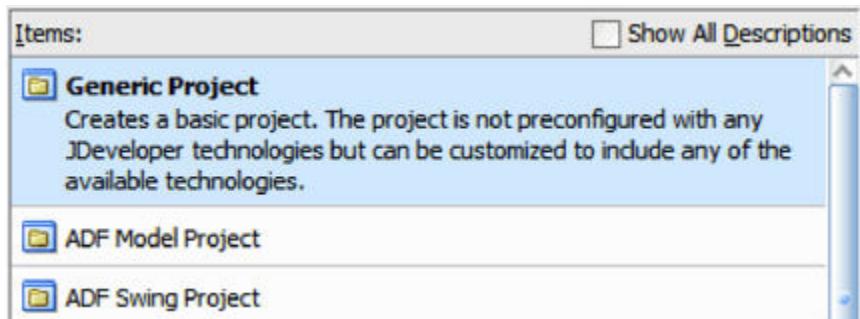


Figure 5 Selecting a Generic Project

Provide an appropriate name (such as "ModelTests") for the project and click **Finish**:

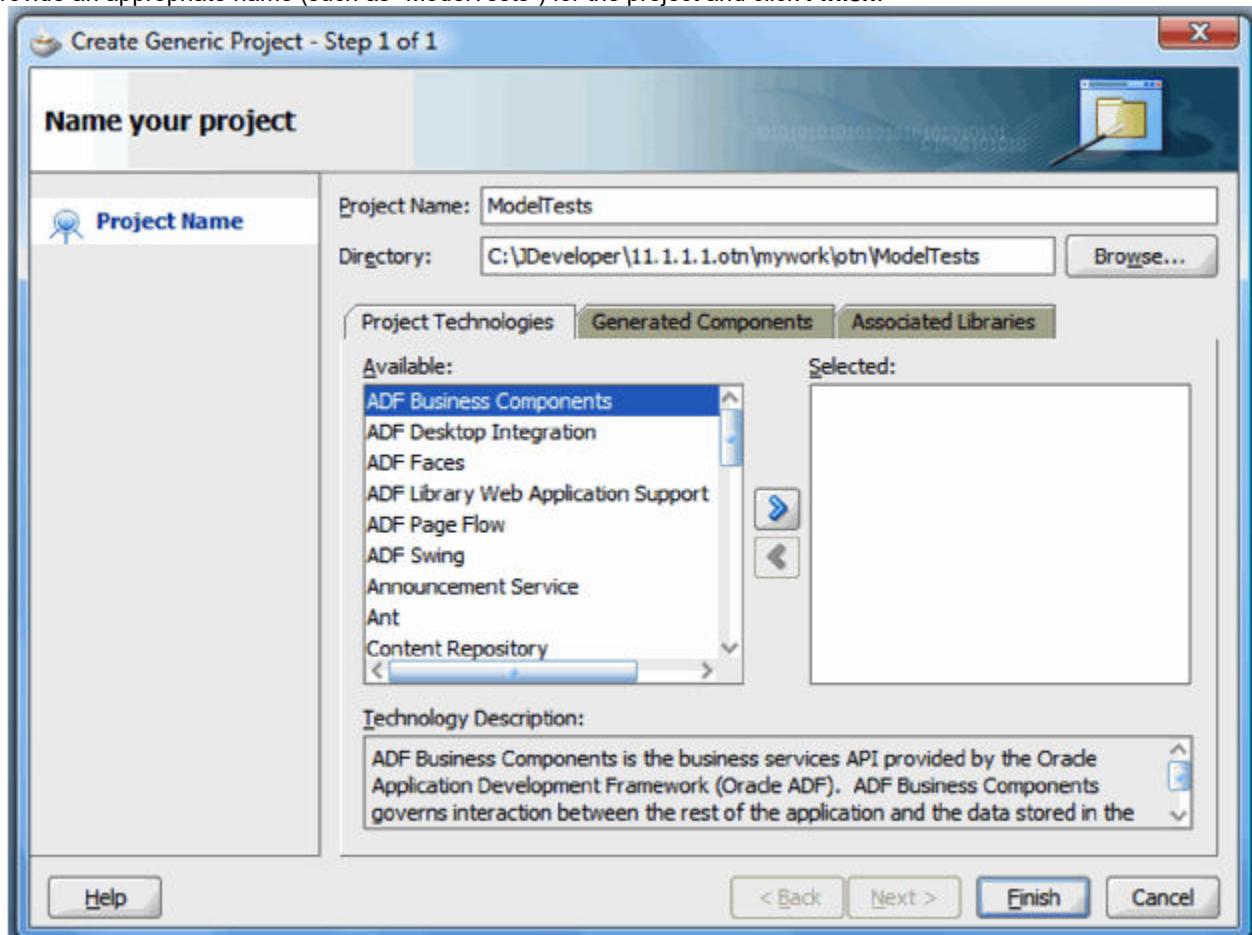


Figure 6 Naming the Tests Project

Creating Unit Tests

Now that we have a project, we can use the Business Components Test Suite Wizard to set up a basic set of JUnit tests for the Oracle ADF Business Components in our Model project, as well as a JUnit Test Fixture (for setting up the database connection) and a JUnit test suite (for running the generated tests). To do so, right-click the ModelTests project and select **New...** from the context menu:

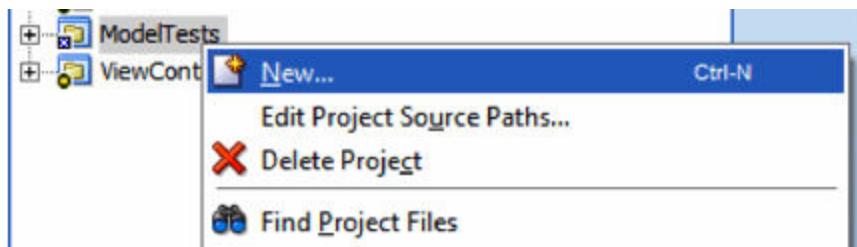


Figure 7 Invoking New from the Context Menu

When the **New Gallery** appears, select the **Unit Tests** category and **Business Components Test Suite** item and click **OK**:

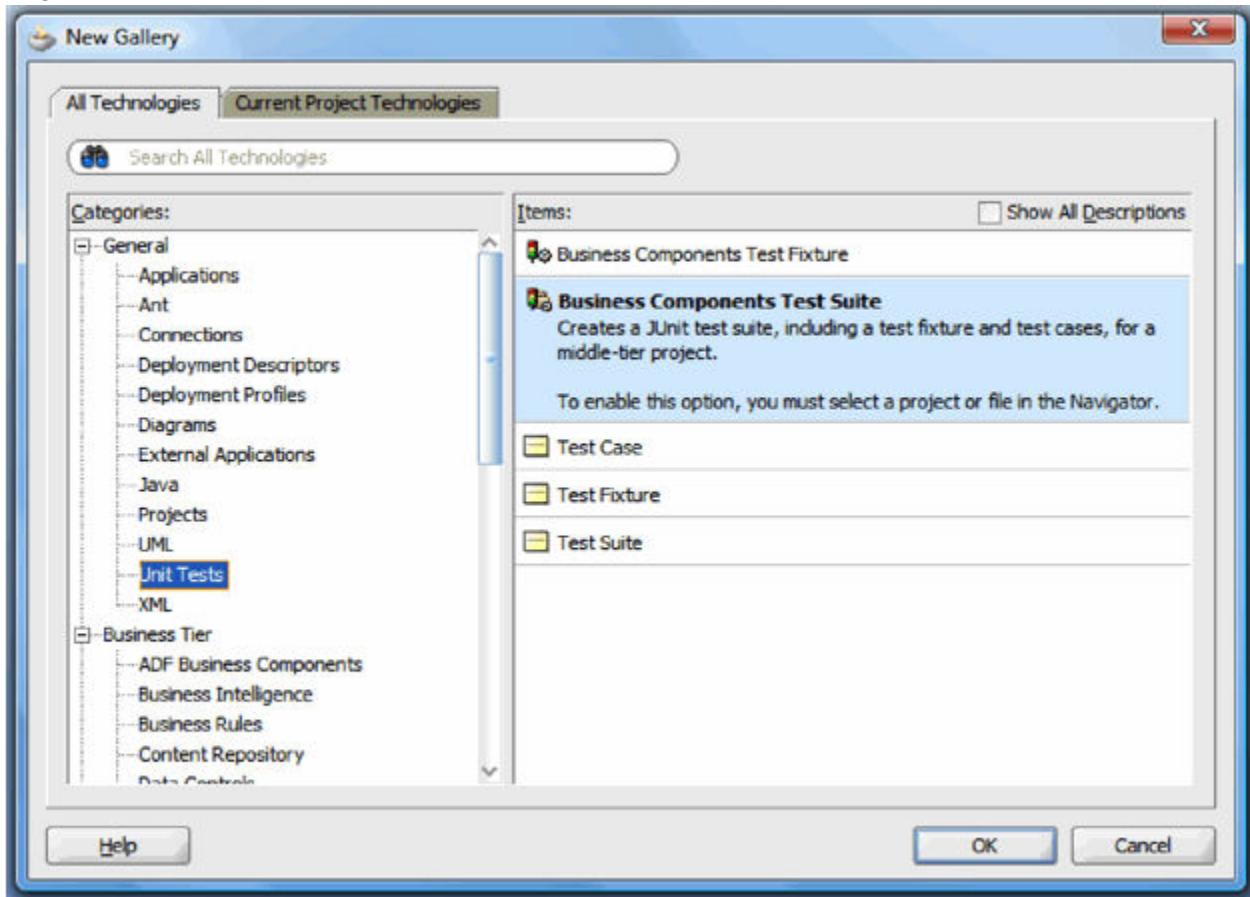


Figure 8 Invoking the Business Components Test Suite Wizard

On the first page of the **JUnit ADF Business Components Test Suite Wizard**, ensure that the Business Components Project (Model.jpr) is selected and that the proper Application Module (OTNAppModule) and Configuration (OTNAppModuleLocal) are selected, and click **Finish**:

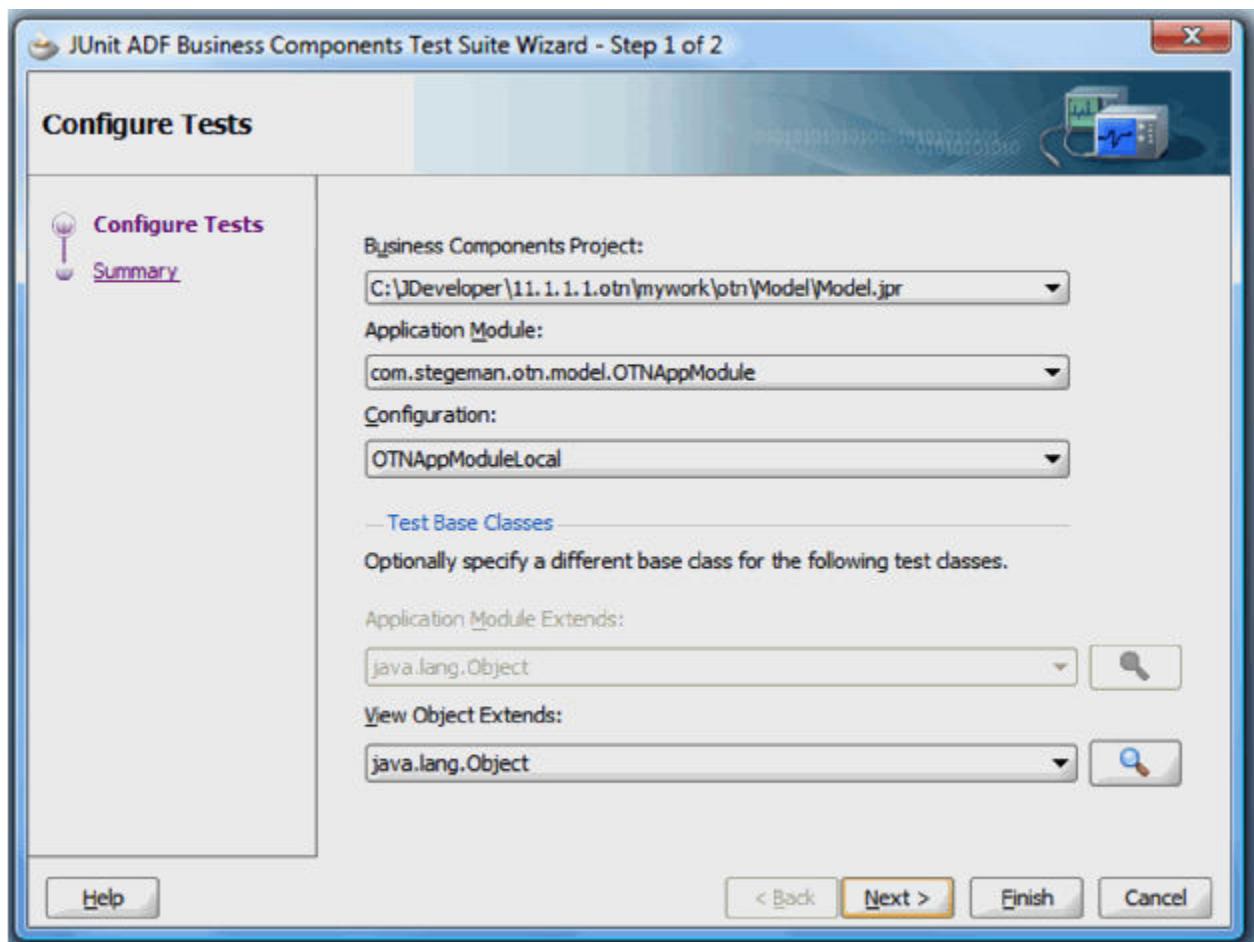


Figure 9 JUnit ADF Business Components Test Suite Wizard

Once you finish the wizard, Oracle JDeveloper will create the following files in the ModelTests project:

File	Purpose
AllOTNAppModuleTests.java	Test suite class
OTNAppModuleAMFixture.java	Test fixture class that is used by all of the tests to obtain an instance of the application module so that each test does not have to create its own instance of the application module (for performance reasons)
DepartmentsVOTest.java, DepartmentsVOTest.xml	Unit test class for the DepartmentsVO View Object
EmployeesForDepartmentVOTest.java, EmployeesForDepartmentVOTest.xml	Unit test class for the EmployeesForDepartmentVO View Object

Before we have a look at the code itself, let's run the generated test cases and see what happens. JUnit uses the concept of a test suite to group tests together, so let's run the generated test suite using Oracle JDeveloper's JUnit test runner. To do so, right-click the test suite class (`AllOTNAppModuleTests.java`) in the Application Navigator and choose **Run** from the context menu:

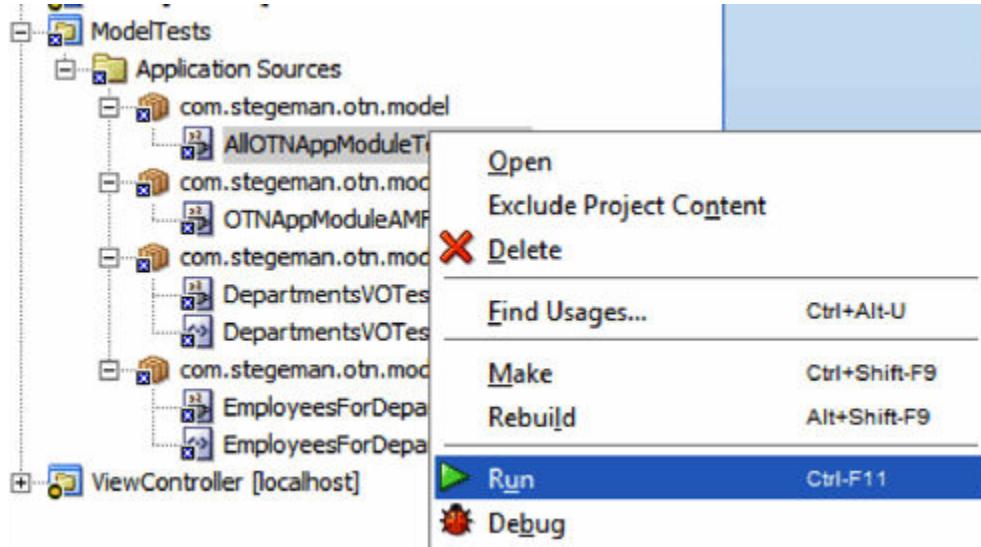


Figure 10 Running the Test Suite

Once the tests run, you will be able to see the results in the JUnit Test Runner – Log window:

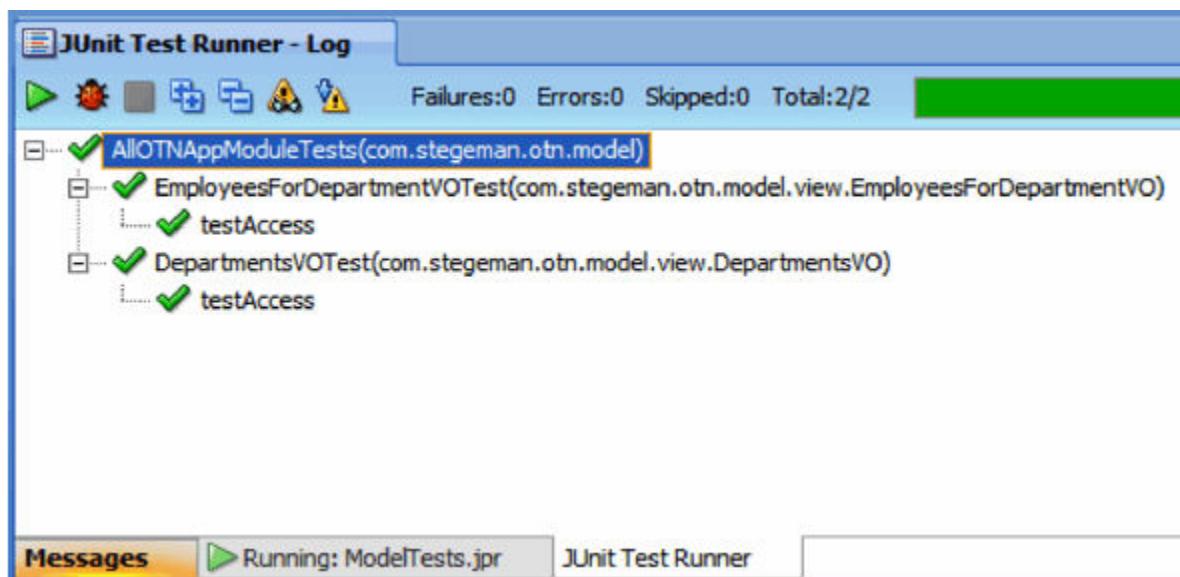


Figure 11 Test Suite Results

From this example, you can see that two unit tests were run and that both of them were successful (no failures or errors). Now, let's have a look at the actual code behind the generated unit tests and add some of our own tests. Double-click the `DepartmentsVOTest.java` file in the Application Navigator to open the source code for the unit tests of the `DepartmentsVO` View Object:

```

@Test
public void testAccess()
{
    ViewObject view = fixture1.getApplicationModule().findViewObject("Departments");
    assertNotNull(view);
}

@Before
public void setUp()
{
}

@After
public void tearDown()
{
}

```

Figure 12 Source Code for DepartmentsVO Unit Tests

The first thing you will notice is the use of Java 5 annotations in the code; these annotations are used to indicate to JUnit which methods are unit tests (with the @Test annotation) and which methods are run @Before or @After each unit test in the class. Unlike in previous versions of JUnit, the methods do not have to follow a specific naming convention for @Test, @Before, or @After methods. One thing to note is that JUnit does not guarantee the order of calling the @Before methods, the @Test methods, or the @After methods; the only guarantee is that all @Before methods are called before each @Test method, and all @After methods are called after each @Test method.

The Oracle JDeveloper –generated test class has one test method called testAccess that attempts to get a View Object instance from an application module (the application module was created by a test fixture— more on that later) and uses a JUnit *assertion* to ensure that the obtained View Object is not null. This is a common pattern for a unit test within JUnit: perform a test and then use one of the assert methods to verify the result. If the assertion is not true, then JUnit will flag the particular unit test as failed.

As you can see, the basic unit test generated by the wizard is quite simplistic. Let's write our own unit test for this View Object to test that the Department ID attribute is being enforced as required. We can write such a unit test by creating a row, not setting the Department ID, validating the row, and seeing that the proper exception is thrown. But, how do we tell JUnit that we expect to see an exception (and furthermore that the test should fail if the exception is not thrown)? We do that by using the *expected* attribute of the @Test annotation to tell JUnit that we expect to see a certain exception. Oracle ADF should throw an oracle.jbo.AttrValException in the case we don't supply a required attribute, so we can code our unit test like this:

```

@Test (expected=AttrValException.class)
public void testDepartmentIdRequired()
{
    ViewObject view = fixture1.getApplicationModule().findViewObject("Departments");

    oracle.jbo.Row r = view.createRow();

    // Set all required attributes except Department ID, which we are testing

    r.setAttribute("DepartmentName", "test");

    r.validate();

}

```

Figure 13 Unit Test to Ensure Department ID is Required

If you run the test suite class again, you should see the additional test run successfully:

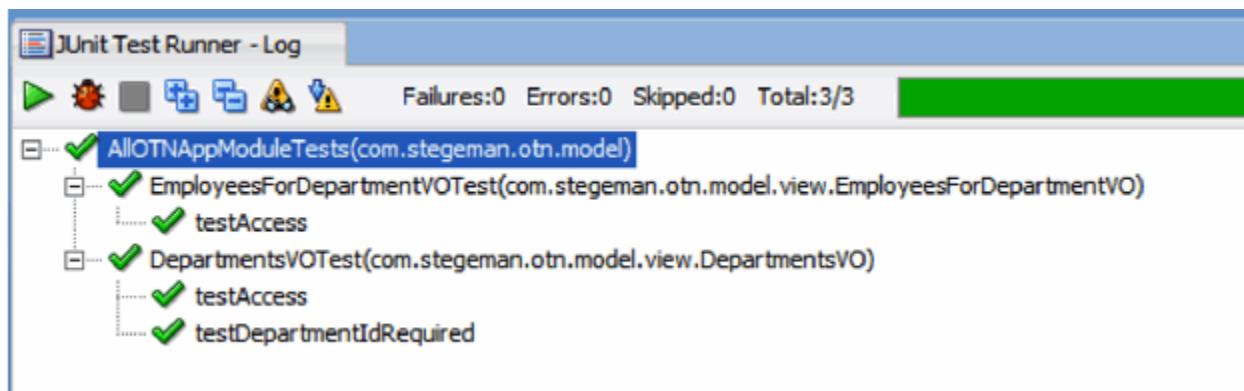


Figure 14 Running with an Additional Unit Test

So far, all of our tests have succeeded. Let's do a bit of TDD now, based upon a new requirement just in from the user community: the Department Name must be at least 4 characters long. In the spirit of TDD, before we write a single line of code to implement this requirement, we must write the test for it and observe it failing. We can write a unit test for this requirement by trying to set the Department Name to a three-character string (with an expected exception) and running the test suite. Here is the test method:

```

@Test (expected=AttrValException.class)
public void testDepartmentNameAtLeastFourCharactersEnforced()
{
    ViewObject view = fixture1.getApplicationModule().findViewObject("Departments");

    oracle.jbo.Row r = view.createRow();

    // Set all required attributes

    r.setAttribute("DepartmentId", 1111);

    // Set Department Name to a 3-character string

    r.setAttribute("DepartmentName", "ABC");

    // validate should throw an AttrValException because it's less than the minimum required

    r.validate();

}

```

Figure 15 Unit Test for Department Name 4-character Minimum

Here is what happens when we run the test suite:

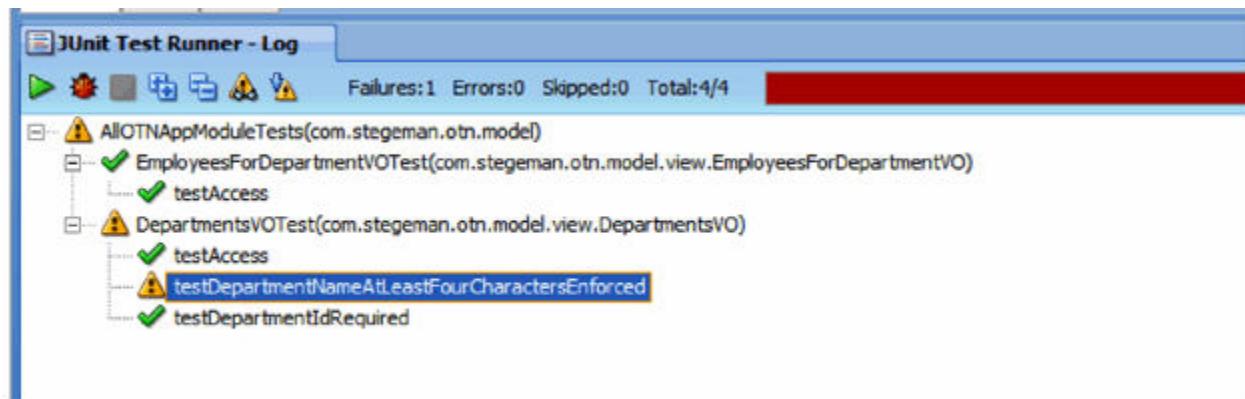


Figure 16 Test Failure for New Requirement

If you click on the failed test, you can even see why the test failed:

```
java.lang.AssertionError: Expected exception: oracle.jbo.AttrValException
    org.junit.internal.runners.MethodRoadie.runTestMethod(MethodRoadie.java:100)
    org.junit.internal.runners.MethodRoadie$2.run(MethodRoadie.java:79)
    org.junit.internal.runners.MethodRoadie.runBeforesThenTestThenAfters(MethodRoadie.java:87)
    org.junit.internal.runners.MethodRoadie.runTest(MethodRoadie.java:77)
    org.junit.internal.runners.MethodRoadie.run(MethodRoadie.java:42)
    org.junit.internal.runners.JUnit4ClassRunner.invokeTestMethod(JUnit4ClassRunner.java:88)
    org.junit.internal.runners.JUnit4ClassRunner.runMethods(JUnit4ClassRunner.java:51)
    org.junit.internal.runners.JUnit4ClassRunner$1.run(JUnit4ClassRunner.java:44)
    org.junit.internal.runners.ClassRoadie.runUnprotected(ClassRoadie.java:27)
    org.junit.internal.runners.ClassRoadie.runProtected(ClassRoadie.java:37)
```

Figure 17 Reason for Test Failure

We now have done the first step of TDD, namely to write a unit test for the new requirement and observe it failing (because we haven't implemented the requirement yet). The next step is to implement the requirement. (I won't show the steps here; simply add a length validation rule to the Departments Entity Object to enforce a minimum length of four characters.) Once we have done that, we can rerun the test suite and observe all tests succeeding:

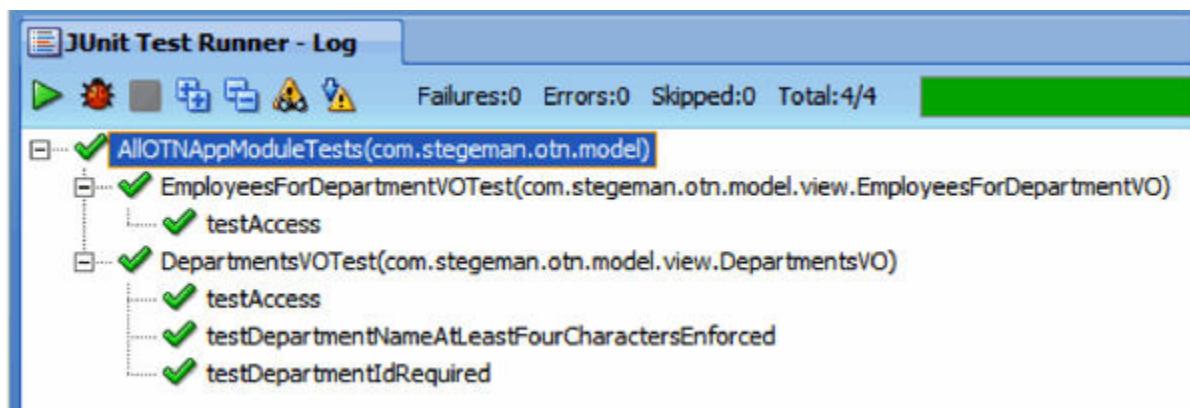


Figure 18 All Tests Succeeding Again
Test Fixtures and Test Suites

The concept of test fixtures and test suites has evolved somewhat from earlier JUnit releases; however, the basic ideas have remained the same. A test fixture is designed to be used to perform any expensive set up or initialization for a group of related tests. The ADF Business Component JUnit Wizard generates a test fixture for you that does the expensive operation of creating an Oracle ADF Application Module instance for use by the rest of the unit tests.

Test suites are simply a logical grouping of test classes that can be run together as a group. The ADF Business Component JUnit Wizard generates a test suite for you that initializes the fixture and specifies, via the `@Suite.SuiteClasses` annotation, which test classes are part of the suite:

```
@Suite.SuiteClasses(
{ EmployeesForDepartmentVOTest.class, DepartmentsVOTest.class })
```

Figure 19 SuiteClasses Annotation in the Test Suite

As you add more test classes to your project, you can simply update the test suite's @Suite.SuiteClasses annotation to include your newly created test classes. You can also create additional test suites simply by creating a new class and adding the annotations manually, or by using the Oracle JDeveloper JUnit Test Suite Wizard to create the class for you. To invoke the wizard, you can right-click your test project in the Application Navigator and click **New...**, and in the **New Gallery** that appears, select the **Unit Tests** category and **Test Suite** item (you may need to select the **All Technologies** tab to see the **Unit Tests** category):

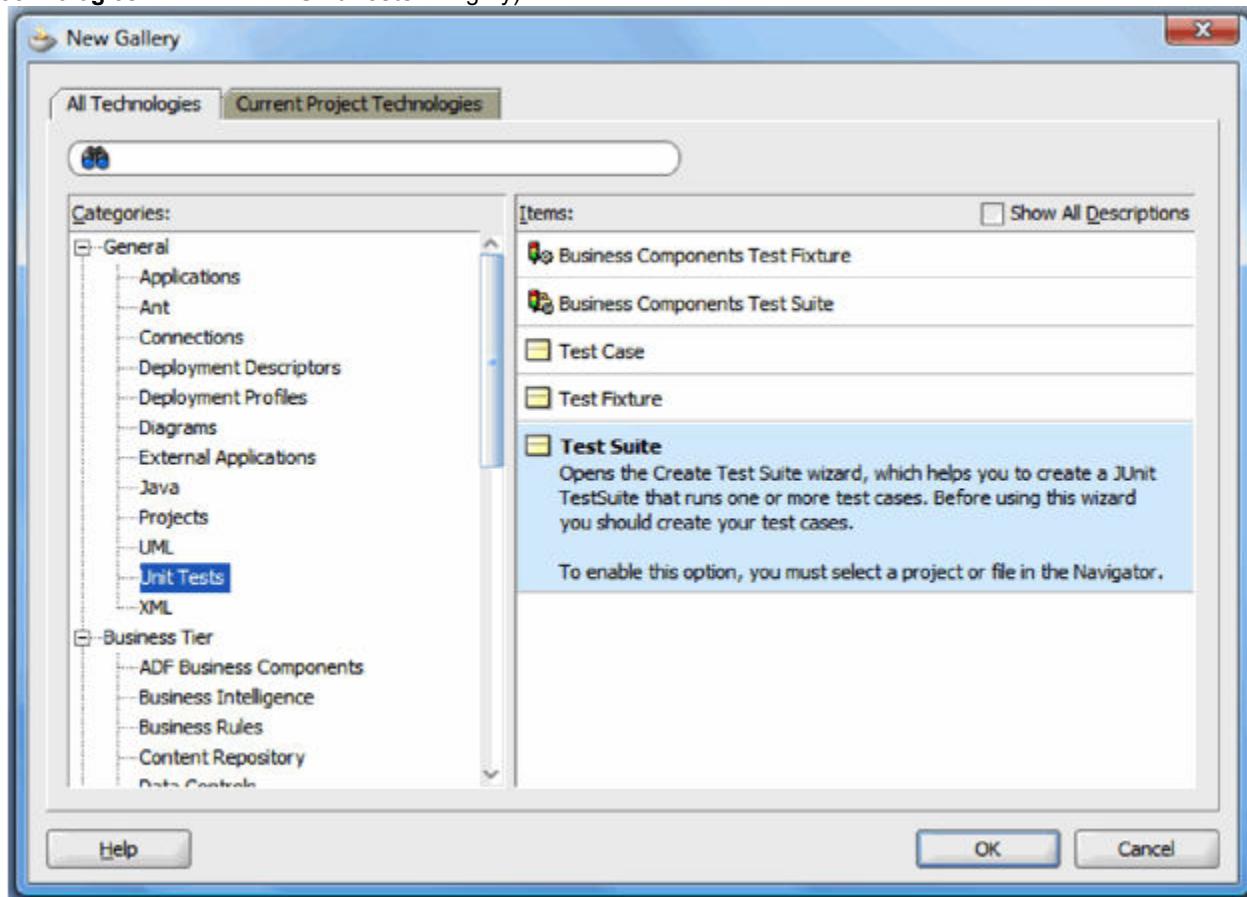


Figure 20 Creating a New Test Suite

By creating additional test suites, you can group your unit tests into logical groupings, which allows you to run a subset of your unit tests at one time.

Using Ant to Automate JUnit

In order to fully leverage JUnit in a TDD approach, your unit tests should be run as often as possible. A common approach is to use a continuous integration (CI) server that builds your code and runs unit tests each time the code is committed to the version control system. A later article in this series describes how to set up such a continuous integration process. A common way of integrating JUnit tests with such CI servers is to add Anttargets to the build process for executing the tests; the CI server can then be set up to run the unit tests as part of the build process (and usually will "fail the build" if any JUnit test fails). In order to do this, we can follow the steps in the Ant article of this

series to create an Ant build file for our ModelTests project. Assuming that you have followed the series up to this point, the steps (in short) are

1. Copy the junit-4.4.jar file from the Oracle JDeveloper installation directory to the corresponding location in the JDeveloperLibs project directory.
2. Add a path reference in jdev-libs.xml to refer to the JUnit 4.4 library:

```
<path id="JDeveloperLibs.library.JUnit.4.Runtime">
  <pathelement location="../JDeveloperLibs/jdeveloper/jdev/extensions/oracle.jdeveloper.junit/junit4.4/junit-4.4.jar"/>
</path>
```

Figure 21 JUnit 4.4 Library Reference

3. Add “Ant” to the technology scope for the ModelTests project and create a build file for that project.
4. Follow the steps in the Ant article to clean up the generated build.properties and build.xml files.
5. Remove the “copy” and “all” targets from the build.xml file.
6. Rename the existing targets to include a “ModelTests” prefix to avoid any name clashes.

Once you have performed these steps, your ANT build.xml file should look something like this:

```

<?xml version="1.0" encoding="windows-1252" ?>
<!--Ant buildfile generated by Oracle JDeveloper-->
<!--Generated 27-Oct-2009 12:15:56-->
<project name="ModelTests" default="all" basedir=".">
  <property file="build.properties"/>
  <import file="../JDeveloperLibs/jdev-libs.xml"/>

  <path id="ModelTests.classpath">
    <path refid="JDeveloperLibs.library.JUnit.4.Runtime"/>
    <path refid="JDeveloperLibs.library.ADF.Model.Runtime"/>
    <path refid="JDeveloperLibs.library.Oracle.XML.Parser.v2"/>
    <path refid="JDeveloperLibs.library.JDeveloper.Runtime"/>
    <path refid="JDeveloperLibs.library.BC4J.Runtime"/>
    <path refid="JDeveloperLibs.library.SQLJ.Runtime"/>
    <path refid="JDeveloperLibs.library.Oracle.JDBC"/>
    <path refid="JDeveloperLibs.library.Connection.Manager"/>
    <path refid="JDeveloperLibs.library.BC4J.Oracle.Domains"/>
    <path refid="JDeveloperLibs.library.MDS.Runtime"/>
    <path refid="JDeveloperLibs.library.MDS.Runtime.Dependencies"/>
    <path refid="JDeveloperLibs.library.BC4J.Security"/>
  </path>

  <target name="ModelTests.init">
    <tstamp/>
    <mkdir dir="${modeltests.output.dir}" />
  </target>

  <target name="ModelTests.clean" description="Clean the project">
    <delete includeemptydirs="true" quiet="true">
      <fileset dir="${modeltests.output.dir}" includes="**/**" />
    </delete>
  </target>

  <target name="ModelTests.compile" description="Compile Java source files" depends="ModelTests.init">
    <javac destdir="${modeltests.output.dir}" classpathref="ModelTests.classpath"
      debug="${javac.debug}" nowarn="${javac.nowarn}"
      deprecation="${javac.deprecation}" encoding="Cp1252" source="1.6"
      target="1.6">
      <src path="src" />
    </javac>
  </target>

```

</project>

Figure 22 Initial build.xml File

The next step is to create an ANT target for running the JUnit test suite. Fortunately, ANT includes a plug-in for running JUnit test suites and generating reports. Before we create the task, we will need to import the build file for the Model project so that we can add a dependency on compiling the Model classes before we test them. Add the following line to the ModelTest project's build.xml file:

```
<import file="../Model/build.xml"/>
```

Figure 23 Importing the Model Project's build.xml File

Now we can create an Ant task to execute our test suite; the JUnit plug-in for Ant will generate a test report, so we first need to add a line to our ModelTest project's build.properties file to specify where the reports go:

```
modeltests.testresults.dir=testResults
```

Figure 24 Defining the Test Report Output Directory

Then, we update the ModelTests.clean and ModelTests.init targets to clean and create the test report directory:

```
<target name="ModelTests.init">
  <tstamp/>
  <mkdir dir="${modeltests.output.dir}" />
  <mkdir dir="${modeltests.testresults.dir}" />
</target>

<target name="ModelTests.clean" description="Clean the project">
  <delete includeemptydirs="true" quiet="true">
    <fileset dir="${modeltests.output.dir}" includes="**/**"/>
    <fileset dir="${modeltests.testresults.dir}" includes="**/**"/>
  </delete>
</target>
```

Figure 25 Initializing and Cleaning the Test Report Directory

Next, we need to define a classpath for executing the Tests. The classpath will need to include the ModelTest project's classpath, the Model project's classpath, the output from both of those projects, and the .adf and src directories for the application (to get connection information). The resulting classpath in build.xml looks like this:

```
<path id="ModelTests.test.classpath">
  <path refid="ModelTests.classpath"/>
  <path refid="model.classpath"/>
  <pathelement location="../Model/classes"/>
  <pathelement location="../ModelTests/classes"/>
  <pathelement location="../.adf"/>
  <pathelement location="../src"/>
</path>
```

Figure 26 Test Execution Classpath

Finally, we can create the Ant task to actually execute our test suite. In the task, we specify the classpath, the test suite(s) to run, and the format to use for the test reports . XML is the best format to use with a CI server, so we'll use it here:

```

<target name="ModelTests.allTests" depends="model.compile, model.copy, ModelTests.compile">
    <junit printsummary="yes" haltonfailure="yes">
        <classpath refid="ModelTests.test.classpath"/>
        <formatter type="xml"/>
        <test name="com.stegeman.otn.model.AllOTNAppModuleTests" todir="testResults"/>
    </junit>
</target>

```

Figure 27 Ant Task for Running the Test Suite

Now, we should be able to run our test suite by executing the appropriate Ant task. One option is to run it from within Oracle Jdeveloper:

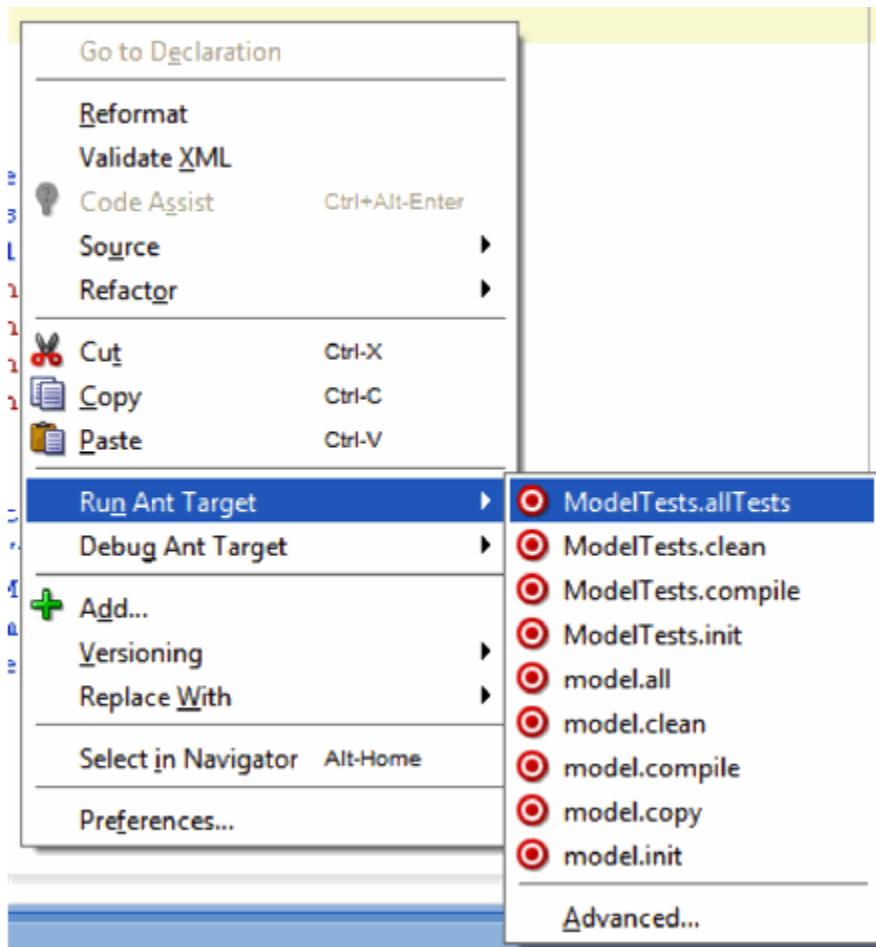


Figure 28 Running the Test Suite via the Context Menu

The other option is to run the test suite from the command line (you may need to copy junit.jar and ant-weblogic.jar from the Oracle JDeveloper Ant installation into your standalone Ant installation for this to work):

```
c:\jDeveloper\11.1.1.1.0tn\mywork\0tn\ModelTests>set ANT_HOME=c:\java\apache-ant-1.7.1  
c:\jDeveloper\11.1.1.1.0tn\mywork\0tn\ModelTests>set JAVA_HOME=c:\java\jdk1.6.0_16  
c:\jDeveloper\11.1.1.1.0tn\mywork\0tn\ModelTests>set PATH=%ANT_HOME%\bin;%JAVA_HOME%\bin;%PATH%  
c:\jDeveloper\11.1.1.1.0tn\mywork\0tn\ModelTests>ant ModelTests.allTests  
Buildfile: build.xml  
  
model.init:  
  
model.compile:  
  
ModelTests.init:  
  
ModelTests.compile:  
  
ModelTests.allTests:  
  [junit] Running com.stegeman.0tn.model.AllOTNAppModuleTests  
  [junit] Tests run: 4, Failures: 0, Errors: 0, Time elapsed: 0.093 sec  
  
BUILD SUCCESSFUL  
Total time: 4 seconds
```

Figure 29 Running the Test Suite from the Command Line

Or

JUnit Test Cases @Before @BeforeClass Annotation

JUnit is the most popular unit [Testing](#) framework in Java. It is explicitly recommended for [Unit Testing](#). JUnit does not require server for testing web application, which makes the testing process fast.

JUnit framework also allows quick and easy generation of test cases and test data. The **org.junit** package consist of many interfaces and classes for JUnit Testing such as Test, Assert, After, Before, etc.

What is Test fixture

Before we understand what a test fixture is, let's study the code below

This code is designed to execute two test cases on a simple file.

```
public class OutputFileTest {  
    private File output;  
    output = new File(...);  
    output.delete();  
public void testFile1(){  
    //Code to verify Test Case 1  
}  
    output.delete();  
    output = new File(...);  
public void testFile2(){  
    //Code to verify Test Case 2  
}  
    output.delete();  
}
```

Few issues here

- The code is not readable
- The code is not easy to maintain.
- When the test suite is complex the code could contain logical issues.

Compare the same code using JUnit

```
public class OutputFileTest  
{  
    private File output;  
    @Before public void createOutputFile()  
    {  
        output = new File(...);  
    }  
  
    @After public void deleteOutputFile()  
    {  
        output.delete();  
    }  
}
```

```
@Test public void testFile1()
{
    // code for test case objective
}
@Test public void testFile2()
{
    // code for test case objective
}
```

The code far more readable and maintainable. The above code structure is a **Text fixture**.

A test fixture is a context where a [Test Case](#) runs. Typically, test fixtures include:

- Objects or resources that are available for any test case.
- Activities required that makes these objects/resources available.
- These activities are
 1. allocation (**setup**)
 2. de-allocation (**teardown**).

Setup and Teardown

- Usually, there are some repeated tasks that must be done prior to each test case. **Example:** create a database connection.
- Likewise, at the end of each test case, there may be some repeated tasks. **Example:** to clean up once test execution is over.
- JUnit provides annotations that help in setup and teardown. It ensures that resources are released, and the test system is in a ready state for next test case.

These annotations are discussed below-

Setup

@Before annotation is used on a method containing [Java](#) code to run before each test case. i.e it runs before each test execution.

Teardown (regardless of the verdict)

@After annotation is used on a method containing java code to run after each test case. These methods will run even if any exceptions are thrown in the test case or in the case of assertion failures.

Note:

- It is allowed to have any number of annotations listed above.
- All the methods annotated with **@Before** will run before each test case, but they may run in any order.
- You can inherit **@Before** and **@After** methods from a super class, Execution is as follows: It is a standard execution process in JUnit.
 1. Execute the **@Before** methods in the superclass
 2. Execute the **@Before** methods in this class
 3. Execute a **@Test** method in this class
 4. Execute the **@After** methods in this class
 5. Execute the **@After** methods in the superclass

Example: Creating a class with file as a text fixture

```
public class OutputFileTest
{
    private File output;
    @Before    public void create outputFile()
    {
        output = new File(...);
    }

    @After public void delete outputFile()
    {
        output.delete();
    }

    @Test public void testFile1()
    {
        // code for test case objective
    }
    @Test public void testFile2()
    {
        // code for test case objective
    }
}
```

In the above example the chain of execution will be as follows-



1. createOutputFile()
2. testFile1()
3. deleteOutputFile()
4. createOutputFile()
5. testFile2()
6. deleteOutputFile()

Assumption: testFile1() runs before testFile2() – which is not guaranteed.

Once-only setup

- It is possible to run a method only once for the entire test class before any of the tests are executed, and prior to any **@Before** method(s).
- "Once only setup" are useful for starting servers, opening communications, etc. It's time-consuming to close and re-open resources for each test.
- This can be done using the **@BeforeClass** annotation

```
@BeforeClass public static void Method_Name() {  
    // class setup code here  
}
```

Once-only tear down

- Similar to once only setup , a once-only cleanup method is also available. It runs after all test case methods and **@After** annotations have been executed.
- It is useful for stopping servers, closing communication links, etc.
- This can be done using the **@AfterClass** annotation

```
@AfterClass public static void Method_Name()
{
    // class cleanup code here
}
```

JUnit Test Suites

If we want to execute multiple tests in a specified order, it can be done by combining all the tests in one place. This place is called as the test suites. More details on how to execute test suites and how it is used in JUnit will be covered in this [tutorial](#).

Junit Test Runner

JUnit provides a tool for execution of your test cases.

- **JUnitCore** class is used to execute these tests.
- A method called **runClasses** provided by **org.junit.runner.JUnitCore**, is used to run one or several test classes.
- Return type of this method is the **Result** object (**org.junit.runner.Result**), which is used to access information about the tests. See following code example for more clarity.

```
public class Test {
    public static void main(String[] args) {
        Result result = JUnitCore.runClasses(CreateAndSetName.class);
        for (Failure failure : result.getFailures()) {
            System.out.println(failure.toString());
        }
        System.out.println(result.wasSuccessful());
    }
}
```

In above code "result" object is processed to get failures and successful outcomes of test cases we are executing.

First JUnit program

Fair knowledge of SDLC, java programming, and basics of software testing process helps in understanding JUnit program.

Let's understand Unit Testing using a live example. We need to create a test class with a test method annotated with **@Test** as given below:

MyFirstClassTest.java

```
package guru99.JUnit;

import static org.junit.Assert.*;
import org.junit.Test;

public class MyFirstClassTest {

    @Test
    public void myFirstMethod(){
        String str= "JUnit is working fine";
        assertEquals("JUnit is working fine",str);
    }
}
```

TestRunner.java

To execute our test method (above) ,we need to create a test runner. In the test runner we have to add test class as a parameter in JUnitCore's runclasses() method . It will return the test result, based on whether the test is passed or failed.

For more details on this see the code below :

```
package guru99.JUnit;

import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
    public static void main(String[] args) {
```

```

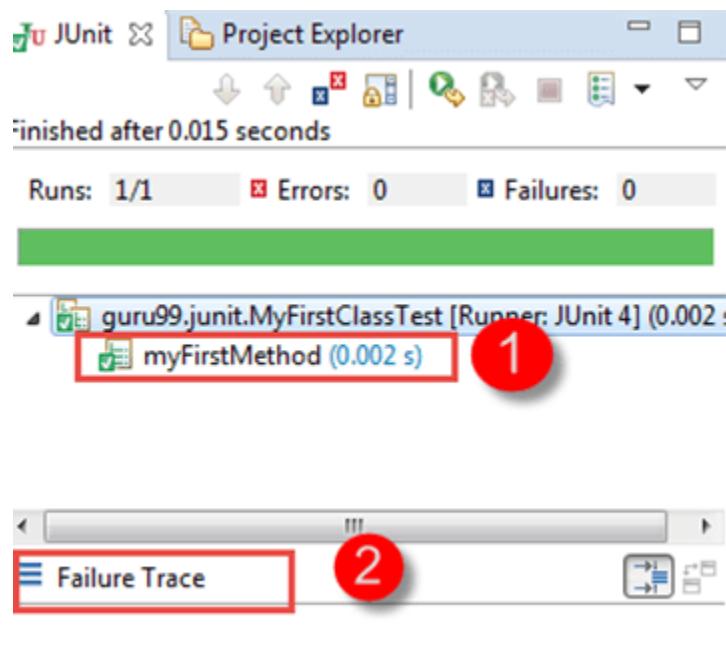
        Result result = JUnitCore.runClasses(MyFirstClassTest.class);
        for (Failure failure : result.getFailures()) {
            System.out.println(failure.toString());
        }
        System.out.println("Result==" + result.wasSuccessful());
    }
}

```

Output

Once **TestRunner.java** executes our test methods we get output as failed or passed. Please find below output explanation:

1. In this example, after executing **MyFirstClassTest.java** , test is passed and result is in green.
2. If it would have failed it should have shown the result as Red and failure can be observed in failure trace. See below JUnit gui :



Summary:

- JUnit is a framework which supports several annotations to identify a method which contains a test.

- JUnit provides an annotation called **@Test**, which tells the JUnit that the public void method in which it is used can run as a test case.
- A test fixture is a context where a test case runs
- To execute multiple tests in a specified order, it can be done by combining all the tests in one place. This place is called as the test suites.
- JUnit provides a tool for execution of the tests where we can run our test cases referred as Test Runner.

Go Microservices, Messaging With RabbitMQ and AMQP

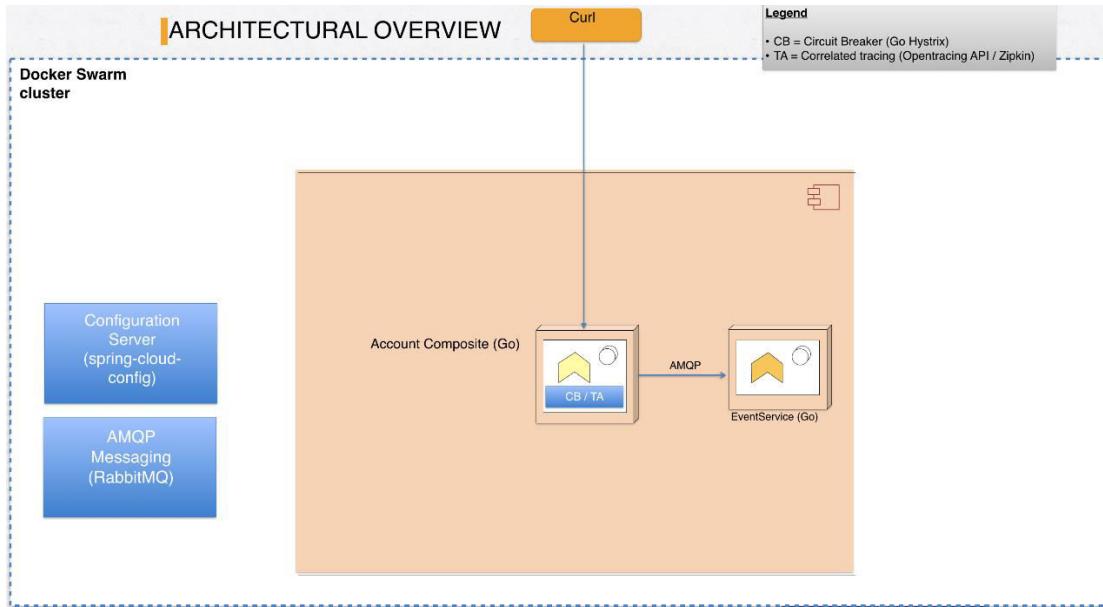
Introduction

Microservices is all about separating your application's business domain into bounded contexts with clearly separated domains, running with process separation where any persistent relations across domain boundaries has to rely on eventual consistency rather than ACID-like transactions or foreign key constraints. A lot of these concepts comes from or has been inspired by [Domain-driven design](#) (DDD). That's yet another huge topic one could write a blog series about.

In the context of our Go microservice blog series and microservice architecture in general, one pattern for accomplishing loose coupling between services is to use messaging for inter-service communication that doesn't need a strict request/response message interchange or similar. That said, using messaging is just one of many strategies one can adopt to facilitate loose coupling between services.

For this part of the blog series, we'll make our "accountservice" place a message on a RabbitMQ *Exchange* whenever a particular account object has been read. This message will be consumed by a brand new microservice we'll write in this blog post. We'll also deal with reusing Go code across multiple microservices by putting them in a "common" library we can import into each service.

Remember the system landscape image from Part 1? Here's an image of what it'll look like after this part has been finished:



There's still a *lot* of stuff missing until we're done. Don't worry, we'll get there.

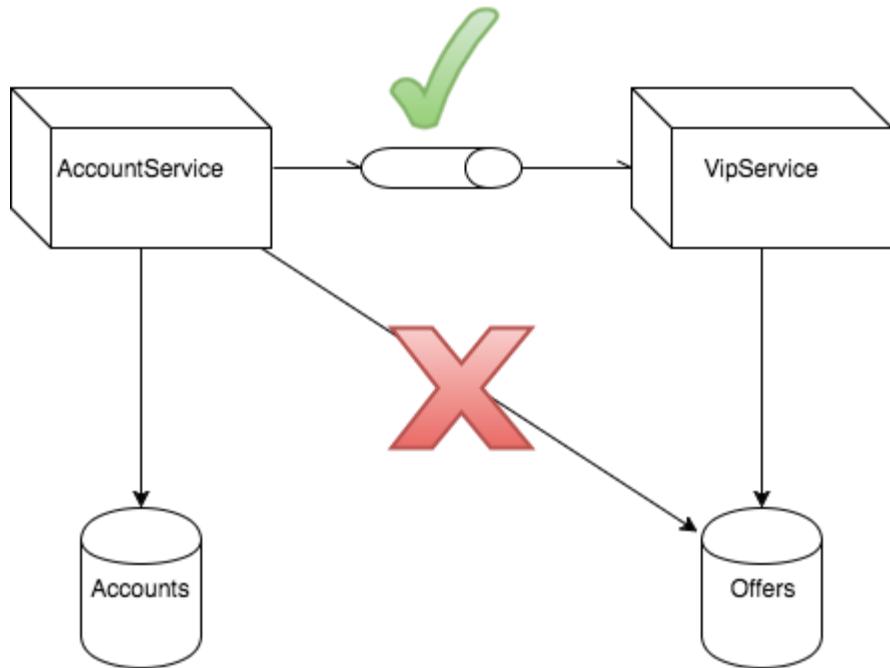
Source Code

There will be a lot of new source code for this part and not all of it will be included in the blog text. For the complete source, clone and switch to the branch for part 9:

```
git checkout P9
```

Sending a Message

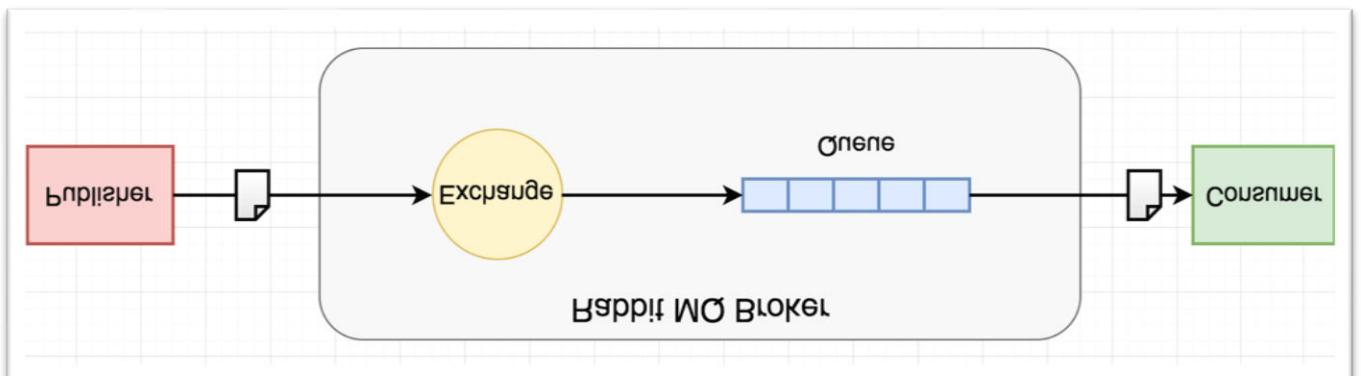
We'll implement a simple make-believe use case: When certain "VIP" accounts are read in the "accountservice," we want to notify a "VIP offer" service that under certain circumstances will generate an "offer" for the account holder. In a properly designed domain model, the accounts objects and VIP offer objects are two independent domains that should have as little knowledge of each other as possible.



The accountservice should **never** access the storage of the VIP service (offers) directly. In this case, we're passing a message to the "vipservice" over RabbitMQ fully delegating both business logic and persistence to the "vipservice."

We'll do all communication using the [AMQP](#) protocol which is an ISO standardized application layer protocol for messaging geared for interoperability. Our Go library of choice for using AMQP is [streadway/amqp](#), just like in part 8 when we consumed configuration updates.

Let's repeat how *exchanges* in AMQP relates to *publishers*, *consumers*, and *queues*:



A message is published to an *exchange*, which then distributes message copies to *queue(s)* based on *routing* rules and bindings which may have registered *consumers*. Check [this thread](#) on Quora for a good explanation.

Messaging Code

Since we'll want to use our new messaging code as well as our existing code for loading configuration from Spring Cloud config in both our existing *accountservice* and the new *vipservice*, we'll create our first shared library.

Start by creating new folders under */goblog* called *common* to keep our new reusable stuff:

```
> mkdir -p common/messaging  
> mkdir -p common/config
```

We'll put all AMQP-related code in the *messaging* folder and the configuration stuff in the *config* folder. You can copy the contents of */goblog/accountservice/config* into */goblog/common/config* - remember that this will require us to update the *import* statements previously importing our config code from within the *accountservice*. Just take a look at the [finished source](#) to see how it's supposed to be.

The messaging code will be encapsulated in a single file that will define both the interface our applications will use to connect, publish and subscribe as well as the actual implementation. In all honesty, there is a lot of boilerplate code required for AMQP-messaging using streadway/amqp so don't get bogged down in the details.

Create a new .go file in */goblog/common/messaging*: [messagingclient.go](#).

Let's have a look at the important stuff:

```
// Defines our interface for connecting, producing and consuming messages.  
type IMessagingClient interface {  
    ConnectToBroker(connectionString string)  
    Publish(msg []byte, exchangeName string, exchangeType string) error  
    PublishOnQueue(msg []byte, queueName string) error  
    Subscribe(exchangeName string, exchangeType string, consumerName string, handlerFunc func(amqp.Delivery)) error  
    SubscribeToQueue(queueName string, consumerName string, handlerFunc func(amqp.Delivery)) error  
    Close()  
}
```

The snippet above defines our messaging interface. This is what our “accountservice” and “vipservice” will deal with when it comes to messaging, hopefully abstracting away most complexity. Note that I’ve chosen two variants of “Produce” and “Consume” to use with *topics* and *direct/queue* messaging patterns.

Next, we’ll define a struct which will hold a pointer to an `amqp.Connection` and that we will attach the requisite methods to so it (implicitly, as always with Go) implements the interface we just declared.

```
// Real implementation, encapsulates a pointer to an amqp.Connection
type MessagingClient struct {
    conn *amqp.Connection
}
```

The implementations are quite verbose so let’s limit ourselves to two of them - `ConnectToBroker()` and `PublishToQueue()`:

```
func (m *MessagingClient) ConnectToBroker(connectionString string) {
    if connectionString == "" {
        panic("Cannot initialize connection to broker, connectionString not set. Have you initialized?")
    }
    var err error
    m.conn, err = amqp.Dial(fmt.Sprintf("%s/", connectionString))
    if err != nil {
        panic("Failed to connect to AMQP compatible broker at: " + connectionString)
    }
}
```

This is how we get hold of the connection pointer, e.g. `amqp.Dial`. If we’re missing our config or cannot contact our broker, we’ll panic our microservice and let the container orchestrator try again with a fresh instance. The passed connection string looks like:

```
amqp://guest:guest@rabbitmq:5672/
```

Note that we’re using the Docker Swarm mode *service* name of the rabbitmq broker.

The `PublishOnQueue()` function is quite long - it’s more or less derived from the official [streadway samples](#), though I’ve simplified it a bit with fewer parameters. To publish a message to a named queue, all we need to pass is:

- Body in the form of a byte array. Could be JSON, XML or some binary.

- `queueName` - the name of the queue you want to send your message to.

For more details about exchanges, see the [RabbitMQ docs](#).

```
func (m *MessagingClient) PublishOnQueue(body []byte, queueName string) error {
    if m.conn == nil {
        panic("Tried to send message before connection was initialized. Don't do that.")
    }
    ch, err := m.conn.Channel()           // Get a channel from the connection
    defer ch.Close()
    // Declare a queue that will be created if not exists with some args
    queue, err := ch.QueueDeclare(
        queueName, // our queue name
        false, // durable
        false, // delete when unused
        false, // exclusive
        false, // no-wait
        nil, // arguments
    )
    // Publishes a message onto the queue.
    err = ch.Publish(
        "", // use the default exchange
        queue.Name, // routing key, e.g. our queue name
        false, // mandatory
        false, // immediate
        amqp.Publishing{
            ContentType: "application/json",
            Body:         body, // Our JSON body as []byte
        })
    fmt.Printf("A message was sent to queue %v: %v", queueName, body)
    return err
}
```

A bit heavy on the boilerplate, but should be easy enough to understand. Declare the queue (so it's created if it does not exist) and then publish our `[]byte` message to it. The code that publishes a message to a named exchange is more complex as it requires boilerplate to first declare an exchange, a queue and then code to *bind* them together. See the [complete source](#) for an example.

Moving on, the actual user of our “MessageClient” will be `/goblog/accounts/service/handlers.go`, so we’ll add a field for that and the hard-coded “is VIP” check that will send a message if the requested account has id “10000”:

```
var DBClient dbclient.IBoltClient
var MessagingClient messaging.IMessagingClient      // NEW
func GetAccount(w http.ResponseWriter, r *http.Request) {
```

```
...
```

A bit further down:

```
...
    notifyVIP(account) // Send VIP notification concurrently.
    // If found, marshal into JSON, write headers and content
    data, _ := json.Marshal(account)
    writeJsonResponse(w, http.StatusOK, data)
}
// If our hard-coded "VIP" account, spawn a goroutine to send a message.
func notifyVIP(account model.Account) {
    if account.Id == "10000" {
        go func(account model.Account) {
            vipNotification := model.VipNotification{AccountId: account.Id, ReadAt: time.Now().UTC().String()}
            data, _ := json.Marshal(vipNotification)
            err := MessagingClient.PublishOnQueue(data, "vipQueue")
            if err != nil {
                fmt.Println(err.Error())
            }
        }(account)
    }
}
```

Taking the opportunity to showcase an inlined anonymous function that we're calling on a new goroutine, i.e. using the *go* keyword. Since we have no reason whatsoever to block the “main” goroutine that's executing the HTTP handler while sending a message, this is a perfect time to add a bit of concurrency.

main.go also needs to be updated so it initializes the AMQ connection on startup using configuration loaded and injected into Viper.

```
// Call this from the main method.
func initializeMessaging() {
if !viper.IsSet("amqp_server_url") {
panic("No 'amqp_server_url' set in configuration, cannot start")
}
service.MessagingClient = &messaging.MessagingClient{}
service.MessagingClient.ConnectToBroker(viper.GetString("amqp_server_url"))
service.MessagingClient.Subscribe(viper.GetString("config_event_bus"), "topic",
", appName, config.HandleRefreshEvent)
}
```

No big deal - we're assigning the *service.MessagingClient* instance by creating an empty messaging struct and the calling *ConnectToBroker* using a property value fetched from Viper. If our configuration doesn't contain a *broker_url*, we panic as we don't want to be running without even the possibility to connect to the broker.

Updating Configuration

We added the `amqp_broker_url` property to our .yml config files back in part 8, so that's already been taken care of.

```
broker_url: amqp://guest:guest@192.168.99.100:5672 _(dev)_  
broker_url: amqp://guest:guest@rabbitmq:5672 _(test)_
```

Note that for the “test” profile, we’re using the Swarm Service name “rabbitmq” instead of the LAN IP address of the Swarm as seen from my dev laptop. (Your actual IP address may vary, 192.168.99.100 seems to be standard when running Docker Toolbox).

As for having clear-text usernames and passwords in configuration files, that’s not recommended, in a real-life scenario, one could typically use the built-in encryption feature of the Spring Cloud Config server we looked at in Part 8.

Unit Testing

Naturally, we should at least write a unit test that makes sure our `GetAccount` function in `handlers.go` does try to send a message whenever someone requests the magical and very very special account identified by “10000.”

For this, we need a mock implementation of the IMessagingClient and a new test case in `handlers_test.go`. Let’s start with the mock. This time we’ll use the 3rd party tool [mockery](#) to generate a mock implementation of our IMessagingClient interface: (*remember to run these commands in a shell with a proper GOPATH set*):

```
> go get github.com/vektra/mockery/.../  
> cd $GOPATH/src/github.com/callistaenterprise/goblog/common/messaging  
> ./$GOPATH/bin/mockery -all -output .  
Generating mock for: IMessagingClient
```

Now we have a mock file `IMessagingClient.go` in our current folder. I don’t like the name of the file nor the camelcasing, so we’ll rename it to something that makes it evident that it’s a mock and follows the conventions for file names used in the blog series:

```
mv IMessagingClient.go mockmessagingclient.go
```

It’s possible you’ll need to adjust the imports somewhat in the generated file, removing the import aliases. Other than that, we’ll use a black-box

approach to this particular mock - just assume it'll work when we start writing tests.

Feel free to examine the [source](#) of the generated mock implementation, it's very similar to the stuff we hand-coded back in part 4 of the blog series.

Moving on to *handlers_test.go*, we're adding a new test case:

```
// declare mock types to make test code a bit more readable
var anyString = mock.AnythingOfType("string")
var anyByteArray = mock.AnythingOfType("[]uint8") // == []byte
func TestNotificationIsSentForVIPAccount(t *testing.T) {
    // Set up the DB client mock
    mockRepo.On("QueryAccount", "10000").Return(model.Account{Id:"10000",
Name:"Person_10000"}, nil)
    DBClient = mockRepo
    mockMessagingClient.On("PublishOnQueue", anyByteArray, anyString).Ret
urn(nil)
    MessagingClient = mockMessagingClient
    Convey("Given a HTTP req for a VIP account", t, func() {
        req := httptest.NewRequest("GET", "/accounts/10000", nil)
        resp := httptest.NewRecorder()
        Convey("When the request is handled by the Router", func() {
            NewRouter().ServeHTTP(resp, req)
            Convey("Then the response should be a 200 and the Mes
sageClient should have been invoked", func() {
                So(resp.Code, ShouldEqual, 200)
                time.Sleep(time.Millisecond * 10) // Sleep
since the Assert below occurs in goroutine
                So(mockMessagingClient.AssertNumberOfCalls(t,
"PublishOnQueue", 1), ShouldBeTrue)
            })
        })
    })
}
```

For details, follow the comments. I don't like that artificial 10 ms sleep just before asserting `numberOfCalls`, but since the mock is called in a goroutine separate from the "main thread" we need to allow it a tiny bit of time to complete. Hope there's a better idiomatic way of unit-testing when there's goroutines and channels involved.

I admit - mocking this way is more verbose than using something like Mockito when writing unit-tests for a Java application. Still, I think it's quite readable and easy enough to write.

Make sure that the test passes:

```
go test ./...
```

Running

If you haven't, run the `springcloud.sh` script to update the config server. Then, run `copyall.sh` and wait a few seconds while our "accountservice" is updated. We'll use curl to fetch our "special" account.

```
> curl http://$ManagerIP:6767/accounts/10000
{"id": "10000", "name": "Person_0", "servedBy": "10.255.0.11"}
```

If things went well, we should be able to open the RabbitMQ admin console and see if we've gotten a message on a queue named `vipQueue`.

```
open http://192.168.99.100:15672/#/queues
```



Overview Connections Channels Exchanges **Queues** Admin

Queues

▼ All queues (2)

Pagination

Page 1 of 1 - Filter: Regex (?)(?)

Overview					Messages		
Name	Features	State	Ready	Unacked	T		
springCloudBus.anonymous.a4nVBWnjQmuDkHke20aM8Q	Excl AD	idle	0	0			
vipQueue		idle	1	0			

At the very bottom of the screenshot above, we see that the "vipQueue" has 1 message. If we use the "Get Message" function within the RabbitMQ admin console, we can look at this message:

Exchange	(AMQP default)
Routing Key	vipQueue
Redelivered	0
Properties	content_type: application/json
Payload 72 bytes Encoding: string	{"accountId": "10000", "readAt": "2017-02-15 14:55:19.954305309 +0000 UTC"}

Writing a Consumer in Go - The “vipservice”

Finally, it’s time to write a brand new microservice from scratch that we’ll use to showcase how to consume a message from RabbitMQ. We’ll make sure to apply the patterns we’ve learned in the blog series up until now, including:

- HTTP server
- Health check
- Centralized configuration
- Reuse of messaging code

If you’ve checked out the P9 source you can already see the “vipservice” in the root `/goblog` folder.

I won’t go through every single line of code here as some parts are repeated from the “accountservice.” Instead, we’ll focus on the consuming the message we just sent. A few things to note:

- Two new .yml files added to the config-repo, `vipservice-dev.yml` and `vipservice-test.yml`.
- `copyall.sh` has been updated so it builds and deploys both the “accountservice” and our new “vipservice.”

Consuming a Message

We’ll use the code from `/goblog/common/messaging` and the `SubscribeToQueue` function, e.g:

```
SubscribeToQueue(queueName string, consumerName string, handlerFunc func(amqp.Delivery)) error
```

Of most note here is the that we’re supposed to provide:

- The name of the queue (e.g. “vip_queue”).
- A consumer name (who we are).
- A handler function that will be invoked with a received delivery - very similar to what we did when consuming config updates in part 8.

The implementation of *SubscribeToQueue* that actually binds our callback function to a queue isn’t that exciting, check the [source](#) if you want the details.

Moving on, a quick peek at an excerpt of the *vipservice*’s *main.go* shows how we’re setting things up:

```
var messagingClient messaging.IMessagingConsumer
func main() {
    fmt.Println("Starting " + appName + "...")
    config.LoadConfigurationFromBranch(viper.GetString("configServerUrl"), appName,
        viper.GetString("profile"), viper.GetString("configBranch"))
    initializeMessaging()
    // Makes sure connection is closed when service exits.
    handleSigterm(func() {
        if messagingClient != nil {
            messagingClient.Close()
        }
    })
    service.StartWebServer(viper.GetString("server_port"))
}
// The callback function that's invoked whenever we get a message on the "vip Queue"
func onMessage(delivery amqp.Delivery) {
    fmt.Printf("Got a message: %v\n", string(delivery.Body))
}
func initializeMessaging() {
    if !viper.IsSet("amqp_server_url") {
        panic("No 'broker_url' set in configuration, cannot start")
    }
    messagingClient = &messaging.MessagingClient{}
    messagingClient.ConnectToBroker(viper.GetString("amqp_server_url"))
    // Call the subscribe method with queue name and callback function
    err := messagingClient.SubscribeToQueue("vip_queue", appName, onMessage)
    failOnError(err, "Could not start subscribe to vip_queue")
    err = messagingClient.Subscribe(viper.GetString("config_event_bus"),
        "topic", appName, config.HandleRefreshEvent)
    failOnError(err, "Could not start subscribe to " + viper.GetString("config_event_bus") + " topic")
}
```

Looks familiar, right? We’ll probably repeat the basics of how to setup and boot each microservice we add.

The *onMessage* function just logs the body of whatever “vip” message we receive. If we would implement more of our make-believe use case it would have invoked some fancy logic to determine if the account holder was eligible for the “super-awesome buy all our stuff (tm)” offer and possibly write an offer to the “VIP offer database”. Feel free to implement and submit a pull request ;)

Not much to add, except this snippet that allows us to clean up whenever we press Ctrl+C or when Docker Swarm thinks it’s time to kill a service instance:

```
func handleSigterm(handleExit func()) {
    c := make(chan os.Signal, 1)
    signal.Notify(c, os.Interrupt)
    signal.Notify(c, syscall.SIGTERM)
    go func() {
        <-c
        handleExit()
        os.Exit(1)
    }()
}
```

Not the most readable piece of code, what it does is that it registers the channel “c” as listener for os.Interrupt and syscall.SIGTERM and a goroutine that will block listening for a message on “c” until either of the signals are received. This allows us to be pretty sure that the *handleExit()* function we supplied will be invoked whenever the microservice is being killed. How sure? Ctrl+C or Docker Swarm scaling works fine. *kill* does too. *kill -9* doesn’t. So please don’t stop stuff using *kill -9* unless you have to.

It will call that *Close()* func we declared on the IMessageConsumer interface, which in the implementation makes sure the AMQP conn is properly closed.

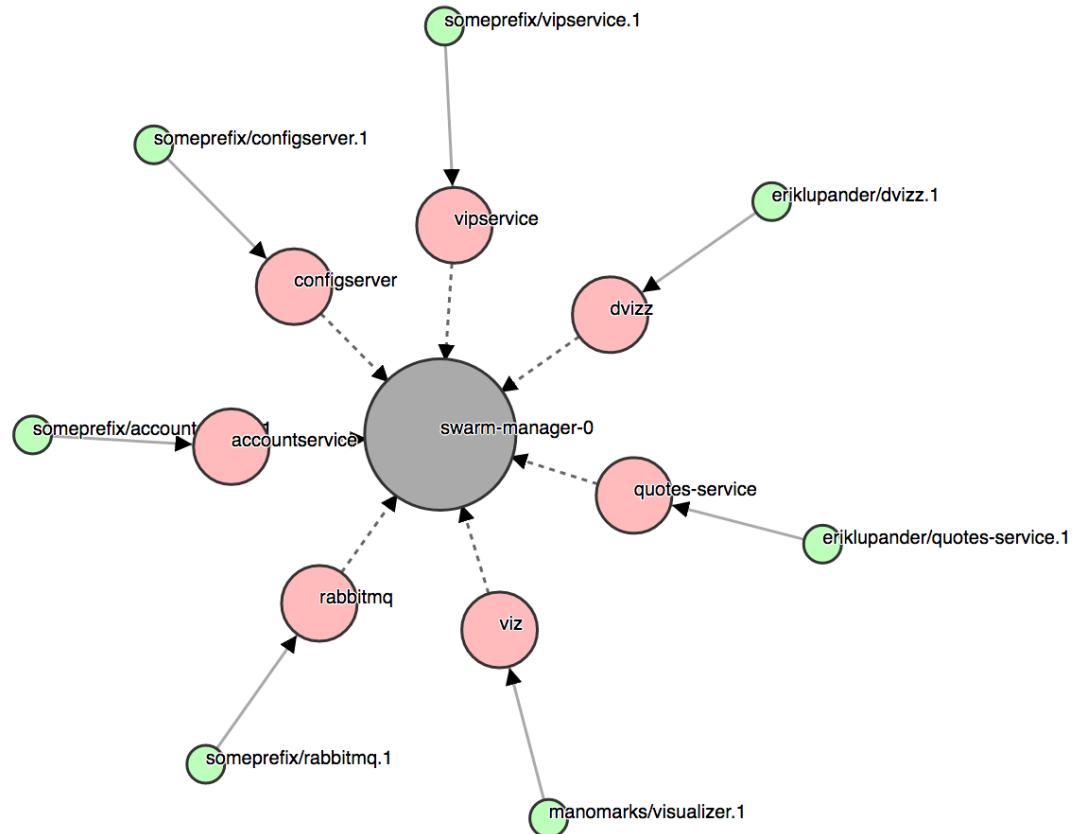
Deploy and Run

The [*copyall.sh*](#) script has been updated, so if you’re following along make sure it’s up-to-date with branch P9 on GitHub and run it. When everything’s done, *docker service ls* should print something like this:

```
> docker service ls
ID          NAME          REPLICAS  IMAGE
kpb1j3mus3tn  accountservice  1/1      someprefix/accountservice
n9xr7wm86do1  configserver   1/1      someprefix/configserver
```

r6bhneq2u89c	rabbitmq	1/1	somewhere/rabbitmq
sy4t9cbf4upl	vipservice	1/1	somewhere/vipservice
ulqcvxm2iqrlr	viz	1/1	manomarks/visualizer:latest

(or using the [dvizz](#) Docker Swarm services renderer):



Checking Logs

Since the *docker service logs* feature is marked as experimental in 1.13.0, we have to look at the “vipservice” logs the old-school way. First, run *docker ps* to figure out the container id:

```

> docker ps
CONTAINER ID        IMAGE
a39e6eca83b3      somewhere/vipservice:latest
b66584ae73ba      somewhere/accountservice:latest
d0074e1553c7      somewhere/configserver:latest
  
```

Pick the CONTAINER ID for the vipservice and check its logs using *docker logs -f*:

```

> docker logs -f a39e6eca83b3
Starting vipservice...
2017/06/06 19:27:22 Declaring Queue ()
  
```

```
2017/06/06 19:27:22 declared Exchange, declaring Queue ()
2017/06/06 19:27:22 declared Queue (0 messages, 0 consumers), binding to Exchange (key 'springCloudBus')
Starting HTTP service at 6868
```

Open another command shell and curl our special Account object.

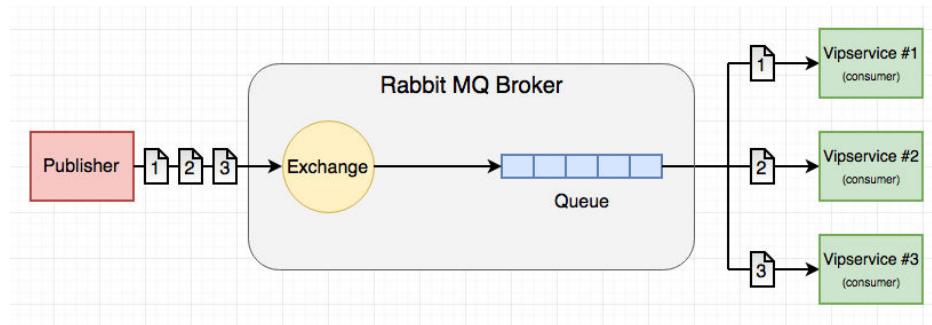
```
> curl http://$ManagerIP:6767/accounts/10000
```

If everything works, we should see a message being consumed in the log of the original window.

```
Got a message: {"accountId":"10000", "readAt":"2017-02-15 20:06:27.033757223 + 0000 UTC"}
```

Work Queues

A pattern for distributing work across multiple instances of a service is to utilize the concept of [work queues](#). Each “vip message” should be processed by a single “vipservice” instance.



So let's see what happens when scale our “vipservice” to two instances using the `docker service scale` command:

```
> docker service scale vipservice=2
```

A new instance of “vipservice” should be available within a few seconds.

Since we're using the *direct/queue* approach in AMQP we expect round-robin behavior. Use `curl` to trigger four VIP account lookups:

```
> curl http://$ManagerIP:6767/accounts/10000
> curl http://$ManagerIP:6767/accounts/10000
> curl http://$ManagerIP:6767/accounts/10000
> curl http://$ManagerIP:6767/accounts/10000
```

Check the log of our original “vipservice” again:

```
> docker logs -f a39e6eca83b3
```

```
Got a message: {"accountId":"10000","readAt":"2017-02-15 20:06:27.033757223 + 0000 UTC"}  
Got a message: {"accountId":"10000","readAt":"2017-02-15 20:06:29.073682324 + 0000 UTC"}
```

As expected, we see that the first instance processed two of the four expected messages. If we'd do *docker logs* for the other "vipservice" instance we'd see two messages there as well. Promise.

Testing the Consumer

Actually, I haven't really come up with an attractive way to unit test the AMQP consumer without spending a ridiculous amount of time mocking the AMQP library. There's a test in [messagingclient test.go](#) that tests the subscriber loop that waits indefinitely for incoming messages to process, but that's it.

For more thorough testing of messaging, I'll probably return to that topic in a future blog post about *integration testing* Go microservices using *go test* with the Docker Remote API or Docker Compose. The test would boot supporting services such as RabbitMQ it can use to send and receive actual messages in test code.

Footprint and Performance

Won't do performance tests this time around, a quick peek at memory use after sending and receiving some messages will have to suffice:

CONTAINER / LIMIT	CPU %	MEM USAGE
vipservice.1.tt47bgnmhef82ajyd9s5hvzs1 / 1.955GiB	0.00%	1.859MiB
accountservice.1.w316okdqbnqz62tg618szsoj / 1.955GiB	0.00%	3.434MiB
rabbitmq.1.i2ixydimyleow0yivaw39xbom / 1.955GiB	0.51%	129.9MiB

The above is after serving a few requests. The new "vipservice" is not as complex as the "accountservice" so it's expected it uses less RAM after startup.

RabbitMQ in Microservices

RabbitMQ, also known as Open-Source Message Broker, supports multiple message protocols and can be deployed on distributed systems. It is lightweight and makes it easy to deploy applications. It mainly acts as a queue where a message which is entered can first be operated on. RabbitMQ runs on many operating systems and cloud environments and provides a wide range of developer tools for most popular languages. It is a Producer-Consumer style pattern where the producer sends a message and the consumer consumes it. The main features of RabbitMQ are mentioned below:

1. Asynchronous Messaging
2. Distributed Deployment
3. Management & Monitoring
4. Enterprise- and Cloud-Ready

Installation

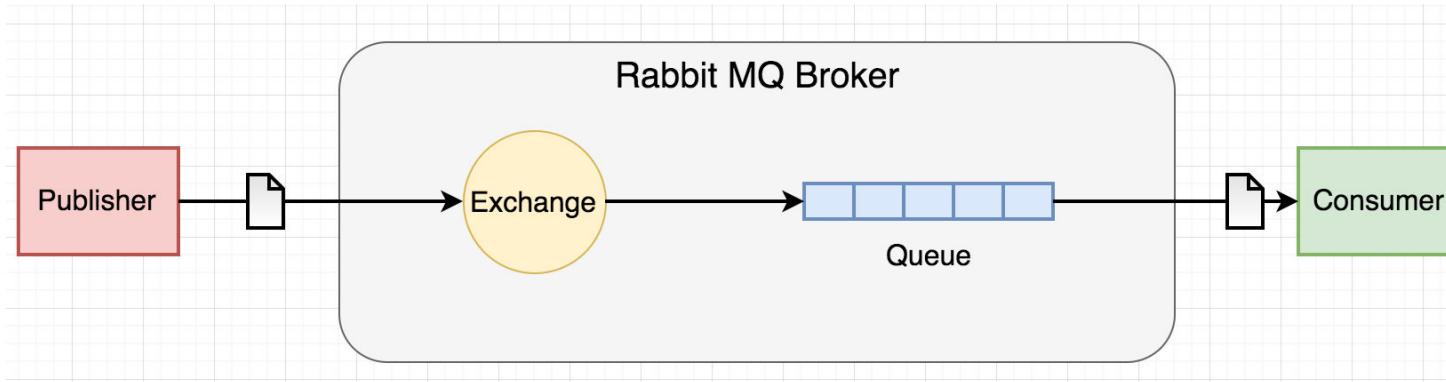
For RabbitMQ, you first need to have [ErLang](#) installed in your system as the RabbitMQ program is written in the ErLang programming language. After ErLang, you can download the latest version of [RabbitMQ](#) from its home site by following instructions given there.

Use of RabbitMQ in Microservices

RabbitMQ is one of the simplest freely available options for implementing messaging queues in your microservices architecture. These queue patterns can help to scale your application by communicating between various microservices. We can use these queues for various purposes, like interaction between core microservices, decoupling of microservices, implementing failover mechanisms, and sending email notifications via message brokers.

Wherever two or more core modules need communicate with each other, we should not make direct HTTP calls, as they can make the core layer tightly coupled and it will be difficult to manage when there are more instances of each core module. Also, whenever a service is down, the HTTP

call pattern will fail, as after a restart, there is no way to track old HTTP request calls. This results in the need for RabbitMQ.



Setting Up RabbitMQ in Microservices

In microservice architecture, for this demonstration, we will use an example pattern of sending email notifications via various core microservices. In this pattern, we will have a producer, any of the core microservices, which will generate the email content and pass it on to the queue. Then this email content is taken by the consumer, who is always listening to the queue for new messages.

Please note that we are using Spring Boot for our microservices, so we will be doing configuration provided for Spring.

1) Producer: This layer is responsible for the generation of email content and passing on this content to the message broker in RabbitMQ.

a) In the properties file, we need to mention the queue name and exchange types along with the host and port where the RabbitMQ server is installed.

```
queue.name=messagequeue
fanout.exchange=messagequeue-exchange
spring.rabbitmq.host: localhost
spring.rabbitmq.port: 5672
spring.rabbitmq.username: guest
spring.rabbitmq.password: guest
```

b) We need to create one configuration class, which will use the queue name and exchange type to bind the queue to the microservice module.

```

@Configuration
public class RabbitConfiguration {
    @Value("${fanout.exchange}")
    private String fanoutExchange;
    @Value("${queue.name}")
    private String queueName;
    @Bean
    Queue queue() {
        return new Queue(queueName, true);
    }
    @Bean
    FanoutExchange exchange() {
        return new FanoutExchange(fanoutExchange);
    }
    @Bean
    Binding binding(Queue queue, FanoutExchange exchange) {
        return BindingBuilder.bind(queue).to(exchange);
    }
}

```

c) Finally, we need one util class, which will be used to send actual email content to the queue using RabbitTemplate, provided by the Spring framework.

```

@Component
public class QueueProducer {
    protected Logger logger = LoggerFactory.getLogger(getClass());
    @Value("${fanout.exchange}")
    private String fanoutExchange;
    private final RabbitTemplate rabbitTemplate;
    @Autowired
    public QueueProducer(RabbitTemplate rabbitTemplate) {
        super();
        this.rabbitTemplate = rabbitTemplate;
    }
    public void produce(NotificationRequestDTO notificationDTO) throws Exception {
        logger.info("Storing notification...");
        rabbitTemplate.setExchange(fanoutExchange);
        rabbitTemplate.convertAndSend(new ObjectMapper().writeValueAsString(notificationDTO));
        logger.info("Notification stored in queue sucessfully");
    }
}

```

d) Then you can call a produce method from anywhere in the module.

```

{
    queueProducer.produce(notificationDTO);
}

```

2) Consumer: This layer is responsible for consuming the message from the RabbitMQ message broker with the FIFO method, then performing email-related operations.

a) In the properties file, we need to mention the queue name and exchange types, along with the host and port where the RabbitMQ server is installed.

```
queue.name=messagequeue
fanout.exchange=messagequeue-exchange
spring.rabbitmq.host: localhost
spring.rabbitmq.port: 5672
spring.rabbitmq.username: guest
spring.rabbitmq.password: guest
```

b) We need to create one configuration class, which will use the queue name and exchange type to bind the queue to the microservice module. Also, in the consumer's RabbitMQ configuration, we will need to create a `MessageListenerAdapter` bean, which will make it act as a consumer and always listen for the incoming message in queue pipeline.

This `MessageListenerAdapter` will have a parameterized constructor with a `Consumer` util class and `defaultListenerMethod`, where we can specify email-related operations.

```
@Configuration
public class RabbitConfiguration {
    private static final String LISTENER_METHOD = "receiveMessage";
    @Value("${queue.name}")
    private String queueName;
    @Value("${fanout.exchange}")
    private String fanoutExchange;
    @Bean
    Queue queue() {
        return new Queue(queueName, true);
    }
    @Bean
    FanoutExchange exchange() {
        return new FanoutExchange(fanoutExchange);
    }
    @Bean
    Binding binding(Queue queue, FanoutExchange exchange) {
        return BindingBuilder.bind(queue).to(exchange);
    }
    @Bean
```

```

SimpleMessageListenerContainer container(ConnectionFactory connectionFactory
,
    MessageListenerAdapter listenerAdapter) {
    SimpleMessageListenerContainer container = new SimpleMessageListenerContainer();
    container.setConnectionFactory(connectionFactory);
    container.setQueueNames(queueName);
    container.setMessageListener(listenerAdapter);
    return container;
}
@Bean
MessageListenerAdapter listenerAdapter(QueueConsumer consumer) {
    return new MessageListenerAdapter(consumer, LISTENER_METHOD);
}
}

```

- c) Then, we need to create the `QueueConsumer` class which will have the specified message listener method where we can do actual email sending operations.

```

@Component
public class QueueConsumer {
    @Autowired
    MailServiceImpl mailServiceImpl;
    protected Logger logger = LoggerFactory.getLogger(getClass());
    public void receiveMessage(String message) {
        logger.info("Received (String) " + message);
        processMessage(message);
    }
    public void receiveMessage(byte[] message) {
        String strMessage = new String(message);
        logger.info("Received (No String) " + strMessage);
        processMessage(strMessage);
    }
    private void processMessage(String message) {
        try {
            MailDTO mailDTO = new ObjectMapper().readValue(message, MailDTO.class);
            ValidationUtil.validateMailDTO(mailDTO);
            mailServiceImpl.sendMail(mailDTO, null);
        } catch (JsonParseException e) {
            logger.warn("Bad JSON in message: " + message);
        } catch (JsonMappingException e) {
            logger.warn("cannot map JSON to NotificationRequest: " + message);
        } catch (Exception e) {
            logger.error(e.getMessage());
        }
    }
}

```

Event Messaging for Microservices with Spring Boot and RabbitMQ

In a microservice environment or any other distributed system you may come upon the requirement to exchange events between services. This article shows how to implement a messaging solution with RabbitMQ.

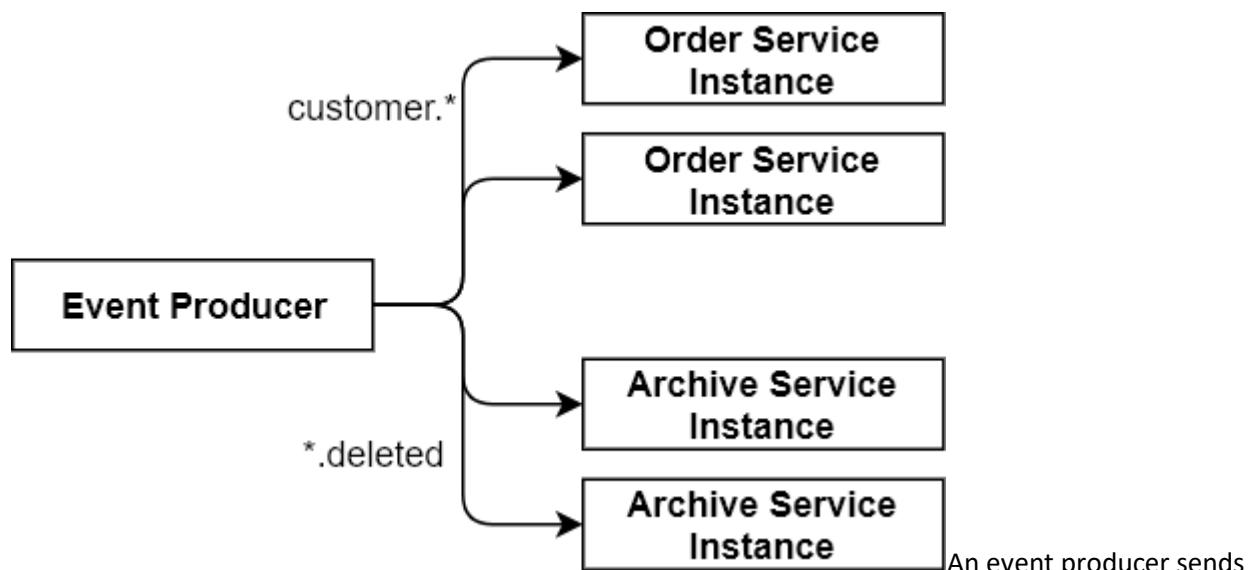
Code Example

This article is accompanied by working example code [on github](#).

<https://github.com/thombergs/code-examples/tree/master/spring-boot/rabbitmq-event-brokering>

Event Messaging Requirements

Before jumping into the solution let's define some requirements that an eventing mechanism in a distributed system should fulfill. We'll use the following diagram to derive those requirements.



- The event producing service must not call the event consuming services directly in order to preserve loose coupling.

- The event producing service must be able to send events of different types (e.g. “customer.created” or “customer.deleted”).
- The event consuming services must be able to receive only events of types they are interested in (e.g. “*.deleted”, which means all events concerning a customer).
- In our distributed system we have several service clusters (e.g. a cluster of “order service” instances and a cluster of “archive service” instances). Each event must be processed by at most one instance per service cluster.

Messaging Concepts

The eventing solution presented in this article makes use of some messaging concepts that are described in the following sections.

Producer

A producer is simply a piece of software that sends a message to a message broker, for example a customer service in a system of microservices that wants to tell other services that a new customer was created by sending the event `customer.created` that contains the newly created customers’ ID as a payload.

Consumer

A consumer is a piece of software that receives messages from a message broker and processes those messages. In our example, this might be an order service that needs the address of all customers to create orders for those customers. It would process the `customer.created` event by reading the ID from the

event and calling the customer service to load the corresponding customers' address.

Queue

A queue is first-in-first-out message store. The messages are put into a queue by a producer and read from it by a consumer. Once a message is read, it is consumed and removed from the queue. A message can thus only be processed exactly once.

Exchange

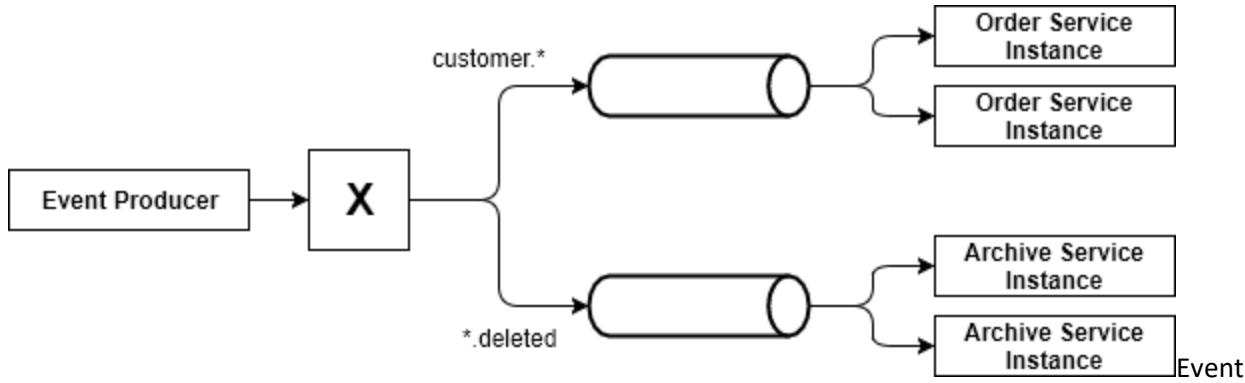
An exchange is a concept that is part of the [AMQP protocol](#). Basically, it acts as an intermediary between the producer and a queue. Instead of sending messages directly to a queue, a producer can send them to an exchange instead. The exchange then sends those messages to one or more queues following a specified set of rules. Thus, the producer does not need to know the queues that eventually receive those messages.

Binding

A binding connects a queue to an exchange. The exchange forwards all messages it receives to the queues it is bound to. A binding can contain a routing key that specifies which events should be forwarded. For example, a binding might contain the routing key `customer.*` meaning that all events whose type starts with `customer.` will be routed to the specified queue.

An Event Messaging Concept with AMQP

Using the concepts above, we can create an eventing solution with RabbitMQ. The solution is depicted in the figure below.



producer and consumers are loosely coupled since an exchange serves as intermediary.

Each service cluster gets its own queue. This is necessary since not all events are relevant to each service cluster. An order service may be interested in all customer events (`customer.*`) whereas an archiving service may be interested in all events where an object has been deleted (`*.deleted`). If we had only one queue for all events that queue would sooner or later overflow since it might contain events that no consumer is interested in.

Each consuming service cluster binds its queue to the central exchange with a routing key that specifies which events it is interested in. Only those events are then routed into the queue. The events are then consumed by exactly one of the service instances connected to that queue.

The event producing services only need to know the central exchange and send all events to that exchange. Since the consuming services take care of the binding and routing, we have a real, loosely coupled eventing mechanism.

Implementing Event Messaging with Spring Boot and RabbitMQ

The eventing concept described above can be implemented with Spring Boot and RabbitMQ. The implementation is pretty straightforward. If you don't feel like reading and more like

delving into code, you will find a link to a github repository with a working example at the end of this article.

Including the Spring Boot AMQP Starter

Spring Boot offers a starter for Messaging with AMQP that integrates the [Spring AMQP](#) project with Spring Boot. The AMQP Starter currently only supports RabbitMQ as underlying message broker, which is fine for us. To use the starter, include the following dependency into your project (Gradle notation):

```
compile('org.springframework.boot:spring-boot-starter-amqp')
```

The starter contains an auto configuration which is automatically activated.

Connecting to RabbitMQ

In order to connect to a RabbitMQ server, the Spring AMQP starter reads the following properties, which you can specify as environment variables, for example in `your application.properties`. The following settings are the default connection settings once you have installed RabbitMQ locally.

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
```

Configuring an Event Producer

Creating an event producer is pretty straightforward. We make use of the `RabbitTemplate` provided by the AMQP starter and call the method `convertAndSend()` to send an event. The event in the code example only contains a String. If the message

should contain a complex object, you can make use of [message converters](#).

The `RabbitTemplate` automatically uses the connection settings provided in the `application.properties` earlier.

```
public class CustomerService {  
  
    private final RabbitTemplate rabbitTemplate;  
  
    private final Exchange exchange;  
  
    public CustomerService(RabbitTemplate rabbitTemplate, Exchange exchange) {  
        this.rabbitTemplate = rabbitTemplate;  
        this.exchange = exchange;  
    }  
  
    public void createCustomer() {  
        // ... do some database stuff  
        String routingKey = "customer.created";  
        String message = "customer created";  
        rabbitTemplate.convertAndSend(exchange.getName(), routingKey, message);  
    }  
}
```

Note that the call to `RabbitTemplate` needs the name of the exchange to which the event should be sent. To wire our application against a specific exchange, we simply create a Spring Bean of type `TopicExchange` and choose a name for that exchange (in case of the example code below, the exchange is called `eventExchange`). The application will automatically connect to RabbitMQ and create an exchange with this name, if it doesn't exist yet. We use a so-called “topic exchange” here, since it allows to specify a routing key (a “topic”) when sending a message to it.

The `RabbitTemplate` passed into the `CustomerService` is provided to the Spring application context by the AMQP starter.

```
@Configuration  
public class EventProducerConfiguration {  
  
    @Bean  
    public Exchange eventExchange() {
```

```

        return new TopicExchange("eventExchange");
    }

    @Bean
    public CustomerService customerService(RabbitTemplate rabbitTemplate,
    Exchange eventExchange) {
        return new CustomerService(rabbitTemplate, senderTopicExchange);
    }
}

```

Configuring an Event Consumer

First off, the event consumer itself is a simple java class. Again, to process more complex objects than simple strings, you can use Spring AMQP's message converters. We use the `@RabbitListener` annotation on a method to mark it as an event receiver.

```

public class EventConsumer {

    private Logger logger = LoggerFactory.getLogger(EventConsumer.class);

    @RabbitListener(queues="orderServiceQueue")
    public void receive(String message) {
        logger.info("Received message '{}'", message);
    }
}

```

We now need to declare a queue and bind it to the same exchange used in the event producer.

First, we define the same `Exchange` as we did in the event consumer configuration. Then, we define a `Queue` with a unique name. This is the queue for our service cluster. To connect the two, we then create a `Binding` with the routing key `customer.*` specifying that we are only interested in customer events.

As with the exchange before, a `Queue` and a `Binding` will be automatically created on the RabbitMQ server if they do not exist yet.

```
@Configuration
public class EventConsumerConfiguration {

    @Bean
    public Exchange eventExchange() {
        return new TopicExchange("eventExchange");
    }

    @Bean
    public Queue queue() {
        return new Queue("orderServiceQueue");
    }

    @Bean
    public Binding binding(Queue queue, Exchange eventExchange) {
        return BindingBuilder
            .bind(queue)
            .to(eventExchange)
            .with("customer.*");
    }

    @Bean
    public EventConsumer eventReceiver() {
        return new EventConsumer();
    }

}
```

RabbitMQ Message Dispatching with Spring AMQP

1. Introduction

In this article, we'll explore the concept of *fanout* and topic exchanges with [Spring AMQP](#) and [RabbitMQ](#).

At a high level, fanout exchanges will **broadcast the same message to all bound queues**, while topic exchanges use a routing key for **passing messages to a particular bound queue** or queues.

Prior reading of [Messaging With Spring AMQP](#) is recommended for this article.

2. Setting Up a Fanout Exchange

Let's set up one fanout exchange with two bound queues.

Spring AMQP allows us to aggregate all the declarations of queues, exchanges, and bindings from Java configurations in a collection and returned as a *List<Declarable>*:

```
1      @Bean
2      public List<Declarable> fanoutBindings() {
3          Queue fanoutQueue1 = new Queue("fanout.queue1", false);
4          Queue fanoutQueue2 = new Queue("fanout.queue2", false);
5          FanoutExchange fanoutExchange = new FanoutExchange("fanout.exchange");
6
7          return Arrays.asList(
8              fanoutQueue1,
9              fanoutQueue2,
10             fanoutExchange,
11             bind(fanoutQueue1).to(fanoutExchange),
12             BindingBuilder.bind(fanoutQueue2).to(fanoutExchange));
13 }
```

With this configuration, we expect that both queues will get all the messages sent to this exchange.

3. Setting Up a Topic Exchange

We'll also set up a topic exchange with two bound queues each with specific routing keys:

```
1  @Bean
2
3  public List<Declarable> topicBindings() {
4
5      Queue topicQueue1 = new Queue(topicQueue1Name, false);
6
7      Queue topicQueue2 = new Queue(topicQueue2Name, false);
8
9
10     TopicExchange topicExchange = new TopicExchange(topicExchangeName);
11
12
13     return Arrays.asList(
14         topicQueue1,
15         topicQueue2,
16         topicExchange,
17         BindingBuilder
18             .bind(topicQueue1)
19                 .to(topicExchange).with("*.important.*"),
20         BindingBuilder
21             .bind(topicQueue2)
22                 .to(topicExchange).with("#{.error}"));
23     }
24 }
```

With this configuration, we expect `topicQueue1` to get all messages with routing keys having a three-word pattern with the middle word being “important” – for example: “`user.important.error`” or “`blog.important.notification`”.

The second queue will get all messages with routing keys ending in error; matching examples are “`user.important.error`” or “`blog.post.save.error`”.

4. Setting Up a Producer

We'll need to create a message producer to use the exchanges we configured. The `BroadcastMessageProducer` class will use a `RabbitTemplate` and its method `convertAndSend` to send messages:

1 @Component

```

2     public class BroadcastMessageProducer {
3
4         @Autowired
5         private RabbitTemplate rabbitTemplate;
6
7         public void sendMessages(String message) {
8             rabbitTemplate.convertAndSend(
9                 SpringAmqpConfig.fanoutExchangeName, "", message);
10            rabbitTemplate.convertAndSend(
11                SpringAmqpConfig.topicExchangeName, "user.not-important.info", message);
12            rabbitTemplate.convertAndSend(
13                SpringAmqpConfig.topicExchangeName, "user.important.error", message);
14        }
15

```

The *RabbitTemplate* provides many overloaded *convertAndSend()* methods for different exchange types.

When we send a message to fanout exchange, the routing key is ignored, and the message is passed to all bound queues.

When we send a message to the topic exchange, we need to pass a routing key. Based on this routing key the message will be delivered to specific queues.

5. Configuring Consumers

Finally, let's set up four consumers – one for each queue – to pick up the messages produced:

```

1     @Component
2     public class BroadcastMessageConsumers {
3
4         @RabbitListener(queues = {SpringAmqpConfig.fanoutQueue1Name})
5         public void receiveMessageFromFanout1(String message) {

```

```

6      }
7
8      @RabbitListener(queues = {SpringAmqpConfig.fanoutQueue2Name})
9      public void receiveMessageFromFanout2(String message) {
10     }
11
12     @RabbitListener(queues = {SpringAmqpConfig.topicQueue1Name})
13     public void receiveMessageFromTopic1(String message) {
14     }
15     @RabbitListener(queues = {SpringAmqpConfig.topicQueue2Name})
16     public void receiveMessageFromTopic2(String message) {
17     }
18 }
19

```

We configure consumers using the `@RabbitListener` annotation. The only argument passed here is the queues' name. Consumers are not aware here of exchanges or routing keys.

6. Running the Example

Our sample project is a Spring Boot application, and so it will initialize the application together with a connection to RabbitMQ and set up all queues, exchanges, and bindings.

By default, our application expects a RabbitMQ instance running on the localhost on port 5672. You can modify the defaults in `application.yaml`.

Our project exposes HTTP endpoint on the URI: “/broadcast”, that accepts POST calls with a message in the request body.

When we send a request to this URI with body “Test” we should see something similar to this in log output:

```

1  2017-04-14 12:27:59.611  INFO 4534 --- [cTaskExecutor-1] c.b.springamqpsimple.MessageC
2  2017-04-14 12:27:59.611  INFO 4534 --- [cTaskExecutor-1] c.b.springamqpsimple.MessageC

```

3 2017-04-14 12:27:59.611 INFO 4534 --- [cTaskExecutor-1] c.b.springamqpsimple.MessageC
4 2017-04-14 12:27:59.611 INFO 4534 --- [cTaskExecutor-1] c.b.springamqpsimple.MessageC
5 2017-04-14 12:27:59.612 INFO 4534 --- [cTaskExecutor-1] c.b.springamqpsimple.MessageC

Create Microservices Architecture Spring Boot

In this Microservices Architecture Spring Boot tutorial, we will discuss to creating a microservices with spring and will see microservices architecture. Microservices allow large systems to be built up from a number of collaborating components. Microservices allows doing loose coupling between application processes instead of loose coupling between application components as Spring does.

1. Introduction

Microservices is not a new term. It coined in 2005 by Dr Peter Rodgers then called micro web services based on SOAP. It became more popular since 2010. Microservices allows us to break our large system into the number of independent collaborating processes. Let us see below microservices architecture.

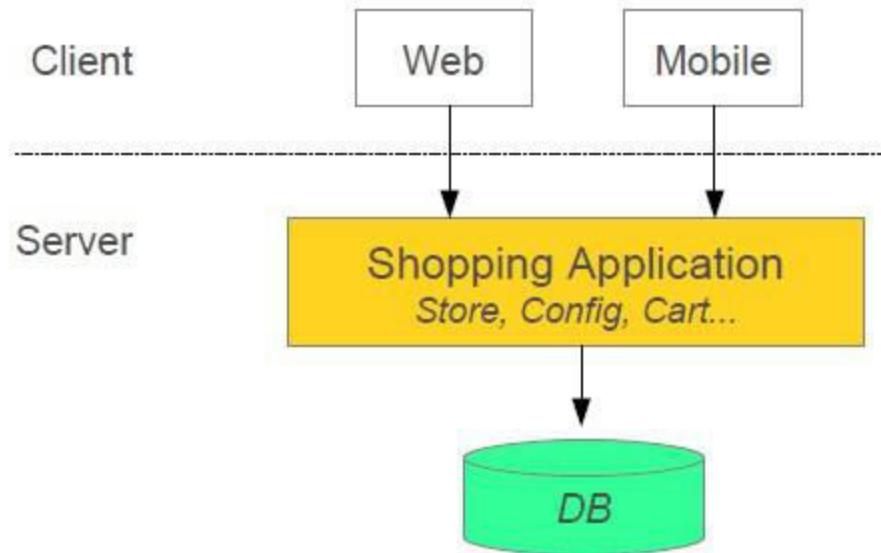
1.1 What is Microservices Architecture?

Microservices architecture allows avoiding monolith application for the large system. It provides loose coupling between collaborating processes which running independently in different environments with tight cohesion. So let's discuss it by an example as below.

For example imagine an online shop with separate microservices for user-accounts, product-catalog order-processing and shopping carts. So these components are inevitably important for such a large online shopping portal. For online shopping system, we could use following architectures.

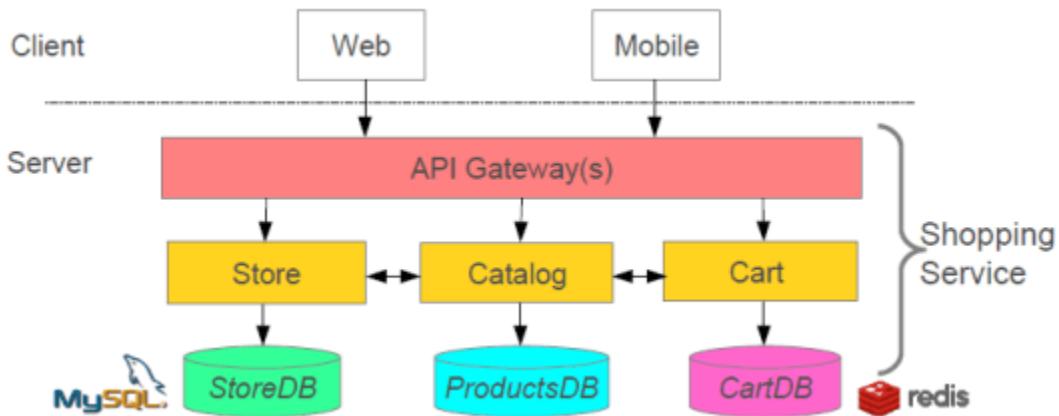
1.2 Shopping system without Microservices (Monolith architecture)

In this architecture we are using Monolith architecture i.e. all collaborating components combine all in one application.



1.3 Shopping system with Microservices

In this architecture style, the main application is divided into a set of sub-applications called microservices. One large application is divided into multiple collaborating processes as below.



Spring enables separation-of-concerns

- **Loose Coupling**- Effect of changes isolated
- **Tight Cohesion**- Code performs a single well-defined task

Microservices provide the same strength as Spring provides

- **Loose Coupling**- Application built from collaborating services or processes, so any process change without affecting another process.
- **Tight Cohesion**- An individual service or process that deals with a single view of data.

There are a number of moving parts that you have to set up and configure to build such a system. For implementing this system is not too obvious you have to know about spring boot,

spring cloud and Netflix. In this post, I will discuss one example for this architecture before the example lets first discuss pros and cons of microservices architecture.

2. Microservices Benefits

- The smaller code base is easy to maintain.
- Easy to scale as an individual component.
- Technology diversity i.e. we can mix libraries, databases, frameworks etc.
- Fault isolation i.e. a process failure should not bring the whole system down.
- Better support for smaller and parallel team.
- Independent deployment
- Deployment time reduce

3. Microservices Challenges

- Difficult to achieve strong consistency across services
- ACID transactions do not span multiple processes.
- Distributed System so hard to debug and trace the issues
- Greater need for an end to end testing
- Required cultural changes in across teams like Dev and Ops working together even in the same team.

4. Microservices Infrastructure

- Platform as a Service like Pivotal Cloud Foundry help to deployment, easily run, scale, monitor etc.
- It supports for continuous deployment, rolling upgrades fo new versions of code, running multiple versions of the same service at same time.

5. Microservices Tooling Supports

5.1 Using Spring for creating Microservices

- Setup new service by using Spring Boot
- Expose resources via a RestController
- Consume remote services using RestTemplate

5.2 Adding Spring Cloud and Discovery server

What is Spring Cloud?

- It is building blocks for Cloud and Microservices
- It provides microservices infrastructure like provide use services such as Service Discovery, a Configuration server and Monitoring.
- It provides several other open source projects like Netflix OSS.
- It provides PaaS like Cloud Foundry, AWS and Heroku.
- It uses Spring Boot style starters

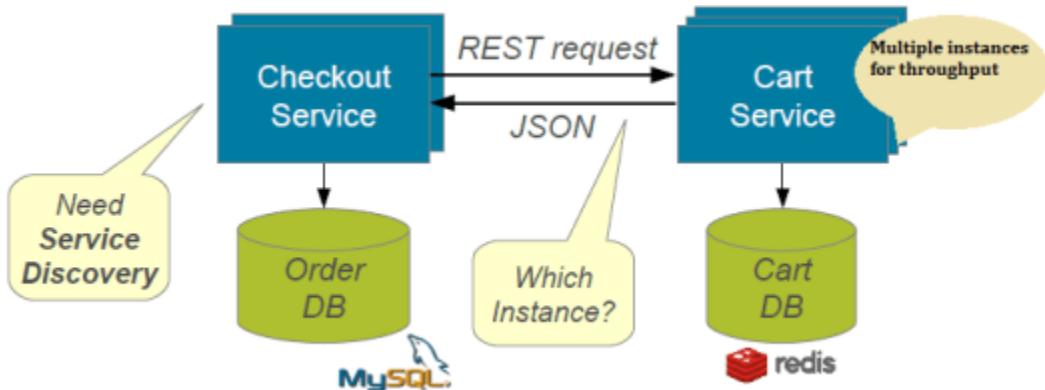
There are many use-cases supported by Spring Cloud like Cloud Integration, Dynamic Reconfiguration, Service Discovery, Security, Client-side Load Balancing etc. But in this post we concentrate on following microservices support

- Service Discovery (How do services find each other?)
- Client-side Load Balancing (How do we decide which service instance to use?)

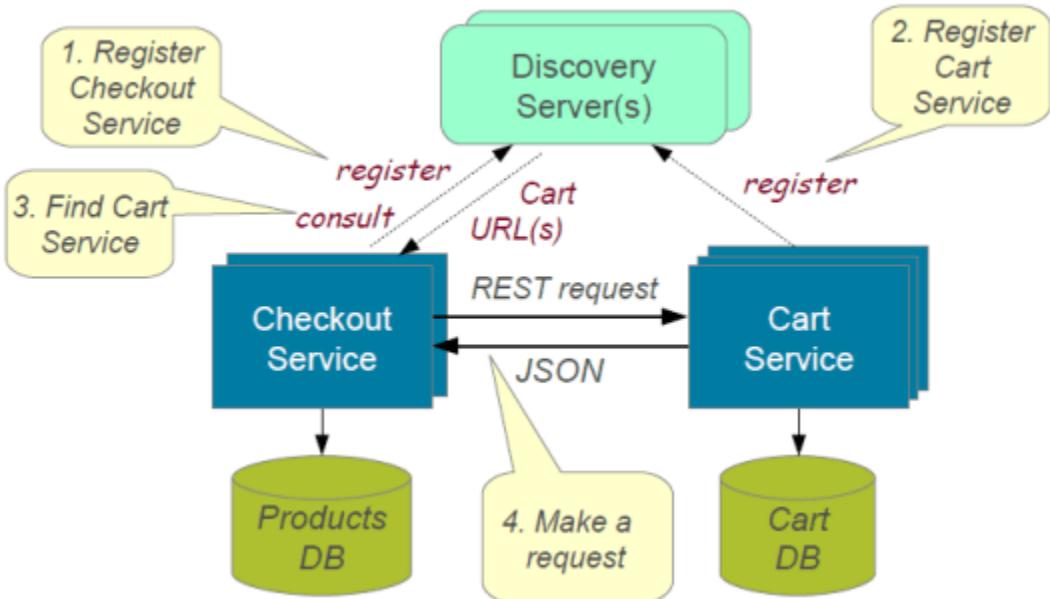
Service Discovery

Problem without discovery

- How do services find each other?
- What happens if we run multiple instances for a service



Resolution with service discovery



Implementing Service Discovery

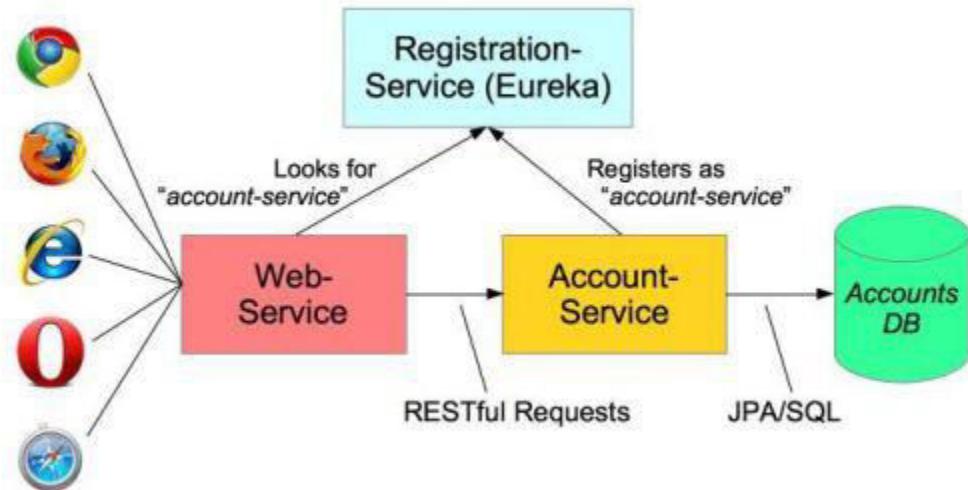
Spring Cloud support several ways to implement service discovery but for this, I am going to use Eureka created by Netflix. Spring Cloud provides several annotation to make it use easy and hiding lots of complexity.

Client-side Load Balancing

Each service typically deployed as multiple instances for fault tolerance and load sharing. But there is the problem how to decide which instance to use?

Implementing Client-Side Load Balancing

We will use Netflix Ribbon, it provides several algorithms for Client-Side Load Balancing. Spring provides smart **RestTemplate** for service discovery and load balancing by using **@LoadBalanced** annotation with **RestTemplate** instance.



@ImageSource-Spring.io

6. Developing Simple Microservices Example

To build a simple microservices system following steps required

1. Creating Discovery Service (Creating Eureka Discovery Service)
2. Creating MicroService (the Producer)
 1. Register itself with Discovery Service with logical service.
3. Create Microservice Consumers find Service registered with Discovery Service
 1. Discovery client using a smart **RestTemplate** to find microservice.

Maven Dependencies

```
<dependencies>

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

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

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

    <dependency>
        <groupId>org.hsqldb</groupId>
        <artifactId>hsqldb</artifactId>
        <scope>runtime</scope>
    </dependency>

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

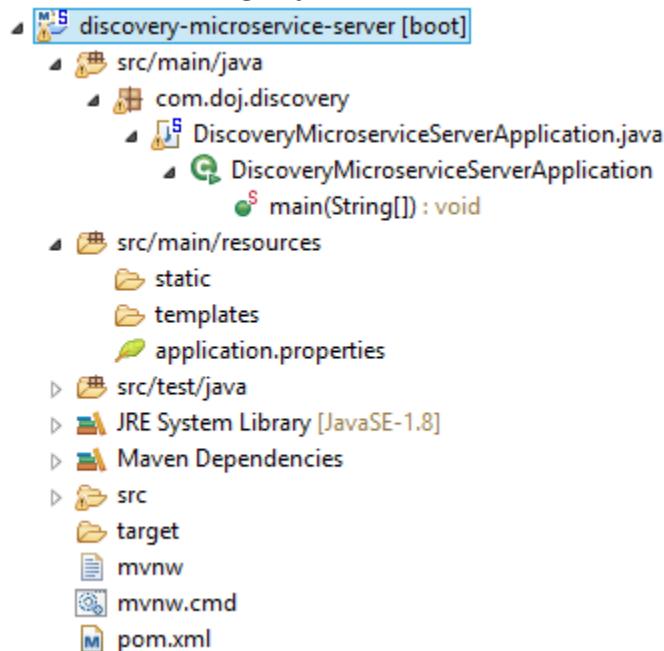
```
<scope>test</scope>

</dependency>

</dependencies>
```

Step 1: Creating Discovery Service (Creating Eureka Discovery Service)

- Eureka Server using Spring Cloud
- We need to implement our own registry service as below.



application.yml

```
# Configure this Discovery Server

eureka:

  instance:

    hostname: localhost

  client: #Not a client

  registerWithEureka: false

  fetchRegistry: false
```

```
# HTTP (Tomcat) port  
  
server:  
  
    port: 1111
```

DiscoveryMicroserviceServerApplication.java

```
package com.doj.discovery;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;  
  
@SpringBootApplication  
@EnableEurekaServer  
public class DiscoveryMicroserviceServerApplication {  
  
    public static void main(String[] args) {  
        SpringApplication.run(DiscoveryMicroserviceServerApplication.class,  
args);  
    }  
}
```

pom.xml

```
<!-- Eureka registration server -->

<dependency>

<groupId>org.springframework.cloud</groupId>

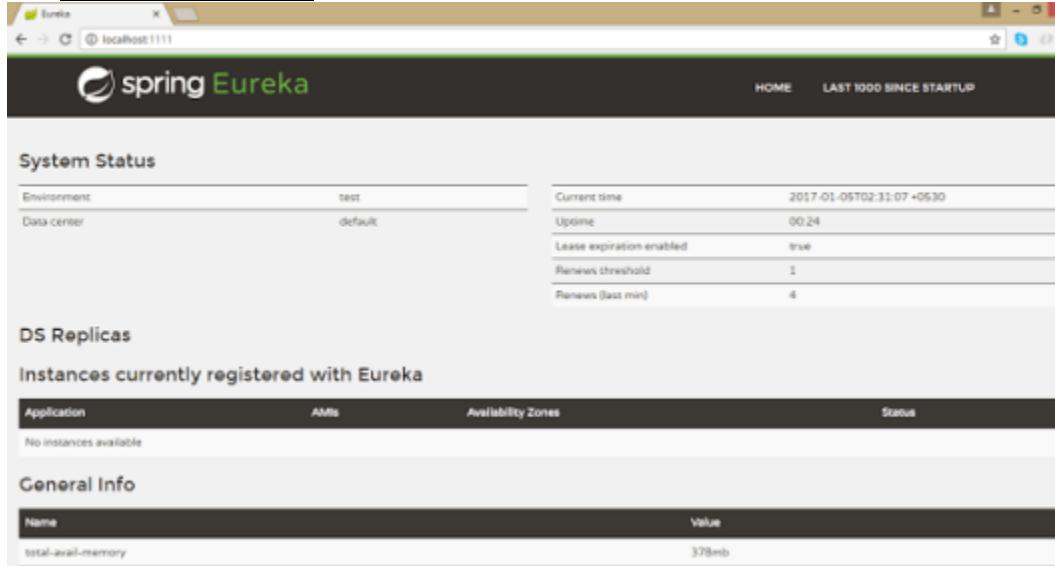
<artifactId>spring-cloud-starter-eureka-server</artifactId>

</dependency>
```

For Whole Source Code for the Discover Server Application, you could download from GitHub as below link.

[discovery-microservice-server](#)

Run this Eureka Server application with right click and run as Spring Boot Application and open in browser <http://localhost:1111>

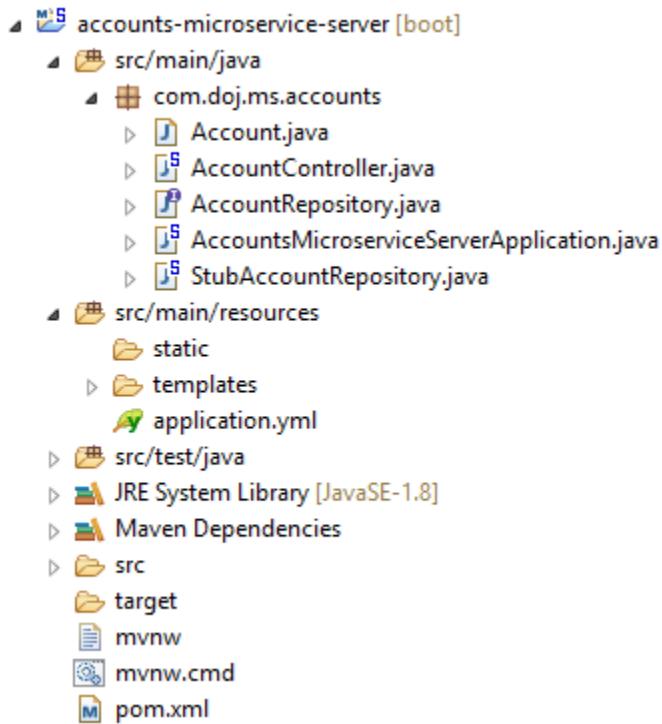


Step 2: Creating Account Producer MicroService

Microservice declares itself as an available service and register to Discovery Server created in **Step 1**.

- Using `@EnableDiscoveryClient`
- Registers using its application name

Let's see the service producer application structure as below.



application.yml

```
### Spring properties

# Service registers under this name

spring:
  application:
    name: accounts-microservice

# Discovery Server Access

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:1111/eureka/
```

```
# HTTP Server (Tomcat) Port

server:

    port: 2222


# Disable Spring Boot's "Whitelabel" default error page, so we can use our
own

error:

    whitelabel:

        enabled: false
```

AccountsMicroserviceServerApplication.java

```
package com.doj.ms.accounts;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@SpringBootApplication
@EnableDiscoveryClient

public class AccountsMicroserviceServerApplication {

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

```
}
```

```
}
```

pom.xml

```
<dependencies>

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

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

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

</dependencies>
```

Other required source files related to this application you could download from GitHub link as given below

accounts-microservice-server

Now run this account service application as **Spring Boot application** and after few seconds refresh the browser to the home page of **Eureka Discovery Server** at <http://localhost:1111/> in previous **Step 1**. Now one Service registered to the Eureka registered instances with Service Name "ACCOUNT-MICROSERVICE" as below

The screenshot shows the Spring Eureka home page at localhost:1111. The top navigation bar includes links for HOME and LAST 1000 SINCE STARTUP. The main content area is divided into sections: System Status, DS Replicas, and General Info.

System Status: Displays environment (test), data center (default), current time (2017-01-05T03:30:44 +0530), uptime (00:00), lease expiration enabled (false), renew threshold (3), and renew (last min) (0).

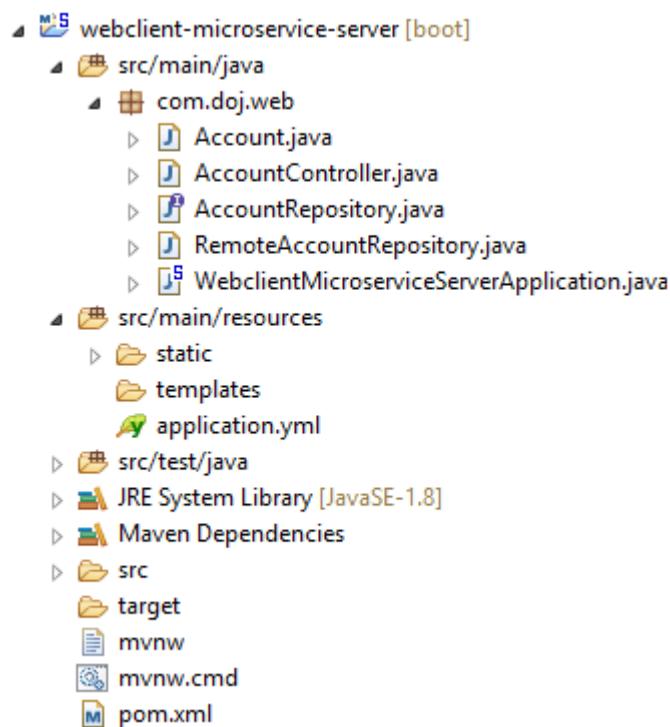
DS Replicas: Shows instances currently registered with Eureka. One instance is listed: ACCOUNTS-MICROSERVICE, with 1 AMI and 1 Availability Zone, status UP (1 - NODDBSL206AG.timesgroup.com:accounts-microservice:2222).

General Info: Displays system metrics and environment variables. Metrics include total-avail-memory (366mb) and environment (test). Environment variables show the application name as ACCOUNTS-MICROSERVICE.

Step 3: Consumer Service

- Create Consumers to find the Producer Service registered with Discovery Service at Step 1.
- `@EnableDiscoveryClient` annotation also allows us to query Discovery server to find microservices.

Let's see the consumer application structure as below.



application.yml

```
# Service registers under this name

# Control the InternalResourceViewResolver:

spring:
  application:
    name: accounts-web

  mvc:
    view:
      prefix: /WEB-INF/views/
      suffix: .jsp

# Discovery Server Access

eureka:
```

```
client:  
  
    serviceUrl:  
  
        defaultZone: http://localhost:1111/eureka/  
  
  
# Disable Spring Boot's "Whitelabel" default error page, so we can use our  
own  
  
error:  
  
    whitelabel:  
  
        enabled: false
```

WebclientMicroserviceServerApplication.java

```
package com.doj.web;  
  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;  
import org.springframework.cloud.client.loadbalancer.LoadBalanced;  
import org.springframework.context.annotation.Bean;  
import org.springframework.web.client.RestTemplate;  
  
  
@SpringBootApplication  
@EnableDiscoveryClient  
public class WebclientMicroserviceServerApplication {
```

```
public static final String ACCOUNTS_SERVICE_URL = "http://ACCOUNTS-MICROSERVICE";  
  
public static void main(String[] args) {  
  
    SpringApplication.run(WebclientMicroserviceServerApplication.class,  
    args);  
  
}  
  
@Bean  
@LoadBalanced  
public RestTemplate restTemplate() {  
  
    return new RestTemplate();  
  
}  
  
@Bean  
public AccountRepository accountRepository(){  
  
    return new RemoteAccountRepository(ACCOUNTS_SERVICE_URL);  
  
}  
  
}
```

pom.xml

```
<dependencies>  
  
<dependency>
```

```
<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-ribbon</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<!-- These dependencies enable JSP usage -->

<dependency>

<groupId>org.apache.tomcat.embed</groupId>
```

```

<artifactId>tomcat-embed-jasper</artifactId>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

</dependency>

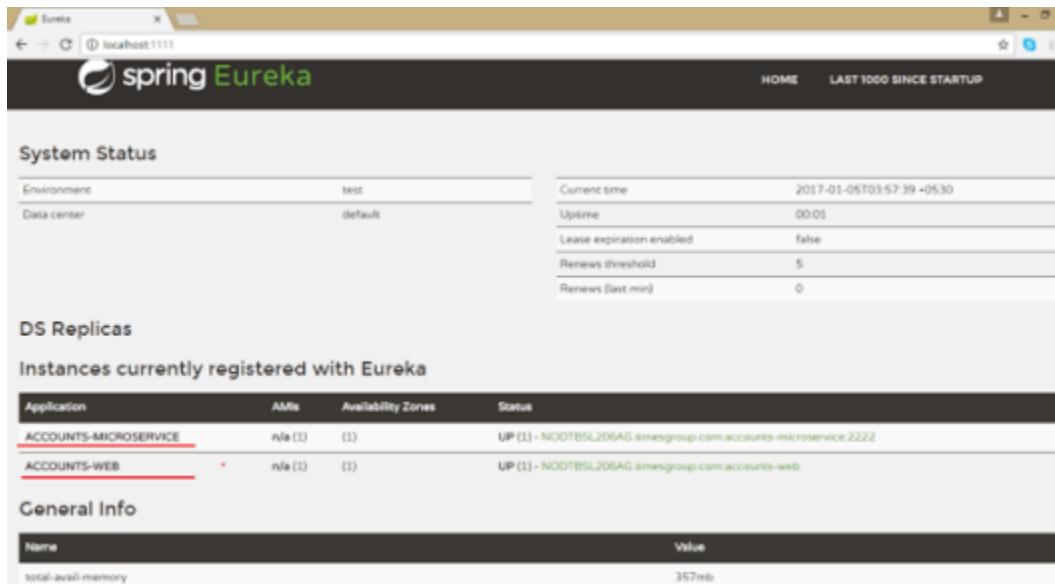
</dependencies>

```

Other required source files related to this application you could download from GitHub link as given below

[web client-microservice-server](#)

Now run this consumer service application as **Spring Boot application** and after few seconds refresh the browser to the home page of **Eureka Discovery Server** at <http://localhost:1111> in previous **Step 1**. Now one more Service registered to the Eureka registered instances with Service Name “ACCOUNTS-WEB” as below



The screenshot shows the Spring Eureka dashboard. At the top, it displays "System Status" with various metrics like Current time (2017-01-05T03:57:39 +0530), Uptime (00:01), Lease expiration enabled (false), Renews threshold (5), and Renews (last min) (0). Below this is the "DS Replicas" section, which lists instances currently registered with Eureka. It shows two entries: "ACCOUNTS-MICROSERVICE" and "ACCOUNTS-WEB". Both instances are listed under the "Status" column as "UP [1] - NOOTB5L206AG.simesgroup.com.accounts-microservice:2222" and "UP [1] - NOOTB5L206AG.simesgroup.com.accounts-web". The "General Info" section at the bottom shows a single entry: "Name" (total-avail-memory) and "Value" (357mb).

Application	AMIs	Availability Zones	Status
ACCOUNTS-MICROSERVICE	n/a [1]	[1]	UP [1] - NOOTB5L206AG.simesgroup.com.accounts-microservice:2222
ACCOUNTS-WEB	n/a [1]	[1]	UP [1] - NOOTB5L206AG.simesgroup.com.accounts-web

Lets our consumer consume the service of producer registered at discovery server.

```
package com.doj.web;

import java.util.Arrays;
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.client.RestTemplate;

/**
 * @author Dinesh.Rajput
 *
 */
public class RemoteAccountRepository implements AccountRepository {

    @Autowired
    protected RestTemplate restTemplate;

    protected String serviceUrl;

    public RemoteAccountRepository(String serviceUrl) {
        this.serviceUrl = serviceUrl.startsWith("http") ? serviceUrl
                : "http://" + serviceUrl;
    }
}
```

```
@Override  
  
public List<Account> getAllAccounts() {  
  
    Account[] accounts = restTemplate.getForObject(serviceUrl + "/accounts",  
    Account[].class);  
  
    return Arrays.asList(accounts);  
  
}  
  
  
@Override  
  
public Account getAccount(String number) {  
  
    return restTemplate.getForObject(serviceUrl + "/accounts/{id}",  
    Account.class, number);  
  
}  
  
}
```

Let's open web application which is a consumer of the account microservice registered at Eureka Discovery Server.

<http://localhost:8080/> as below

spring-microservices: Ho

localhost:8080

spring Pivotal.

Accounts Web - Home Page

- [View Account List](#)

Now click on **View Account List** then fetch all accounts from account microservice.
<http://localhost:8080/accountList>

spring-microservices: Acc

localhost:8080/accountList

spring Pivotal.

Account List

[Home]

- [Arnav](#)
- [Anamika](#)
- [Dinesh](#)

Now click on any account from the list of accounts to fetch the details of the account for account number from account microservice.

<http://localhost:8080/accountDetails?number=5115>

spring-microservices: Acc

localhost:8080/accountDetails?number=5115

spring Pivotal.

Account Details

[Home | Accounts]

Account:	5115
Name:	Arnav
Amount:	1000

Load Balanced *RestTemplate*

Create using **@LoadBalanced**- Spring enhances it to service lookup & load balancing

```
@Bean  
  
    @LoadBalanced  
  
    public RestTemplate restTemplate() {  
  
        return new RestTemplate();  
  
    }
```

Must inject using the same qualifier-

- If there are multiple **RestTemplate** you get the right one.
- It can be used to access multiple microservices

```
@Autowired  
  
    @LoadBalanced  
  
    protected RestTemplate restTemplate;
```

Load Balancing with Ribbon

Our smart RestTemplate automatically integrates two Netflix utilities

- **Eureka** Service Discovery
- **Ribbon** Client Side Load Balancer

Eureka returns the URL of all available instances

Ribbon determine the best available service too use

Just inject the load balanced **RestTemplate** automatic lookup by *logical service-name*

7. Summary

After completion of this article you should have learned:

- What is the MicroServices Architecture
- Advantages and Challenges of MicroServices
- And some information about Spring Cloud such as Eureka Discover Server by Netflix and Ribbon.

Creating Real Time MVC Projects Step by Step with n-layer architecture

Introduction

Model view controller (MVC) is a software architectural pattern for developing web applications. It divides a given application into three interconnected parts.

Model - The model represents the data. The model does not depend on the controller or the view. It is a virtual representation of data used to perform any operation in the project.

View - This is responsible for displaying the UI to the user.

Controller - It mainly handles the incoming request coming from browser and processes these requests to implement View.

The MVC design pattern decouples these major components allowing us for code reusability and parallel development.

In this pattern Controller receives all incoming requests from browser and then works with the Model to identify the data used to create a view. The View then uses the data and generates a final presentable UI.

Here in this article we will learn how to create a real time project using MVC architecture.

Table of contents

Creating multi-layer MVC project

1. Setting up project layers.
2. Creating UI, BLL, DAL for the project.
3. Understanding the flow between the layers.

Dynamic Home Page Design for the Project

1. Creating Master page for the project
2. A slider in the Homepage.
3. Dynamically data to populate in Home page.

Working with Database

1. Creating Sample Table Structure,
2. Writing multi-purpose Store procedure.

Master Details Entry for the Project

1. Creating a form for Patient Entry.
2. Displaying the patient entry using a MVC web grid.

3. Template entry of for the project.
4. Displaying all template with(Searching,Sorting) using Jquery DataTable.

Setting Up Project Layers

Before setting up project layer let me tell you all what a layer exactly in a project.

Layers refer to the internal architecture of a project,For ex: You divide your project into different layers like Data access layer, Business logic layer,User Interface layer etc.. So they are internal to the project, and these layers interact with each other internally to form the entire working component.

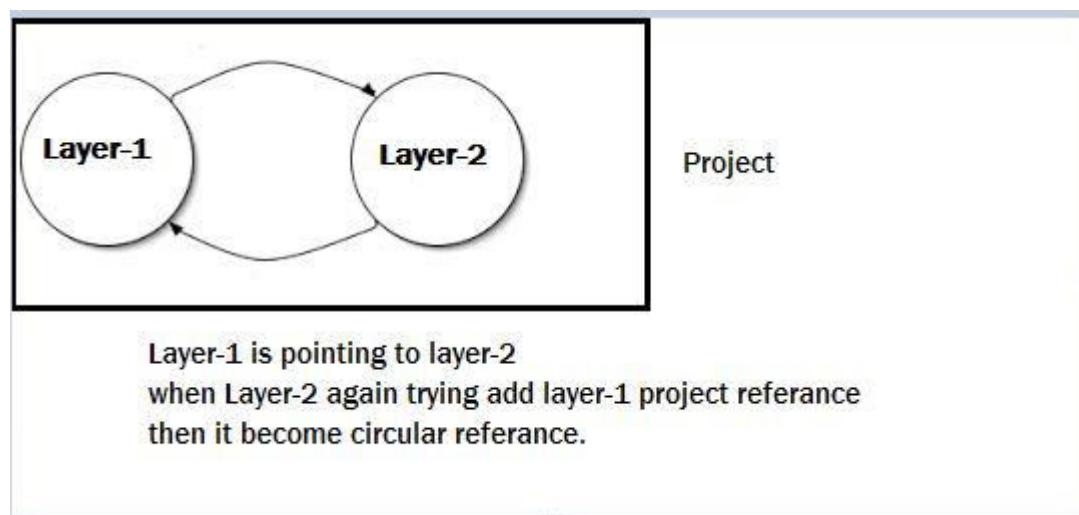
The main benefits of the layered architectural style are:

Abstraction,Isolation,Reusability, Testability.

while creating project layers please make focus on following things-

1. Beware of Circular Dependency
2. Each layer should have their own Responsibility(SRP).
3. The layers which are commonly used try to make these layers as parallel dependant.

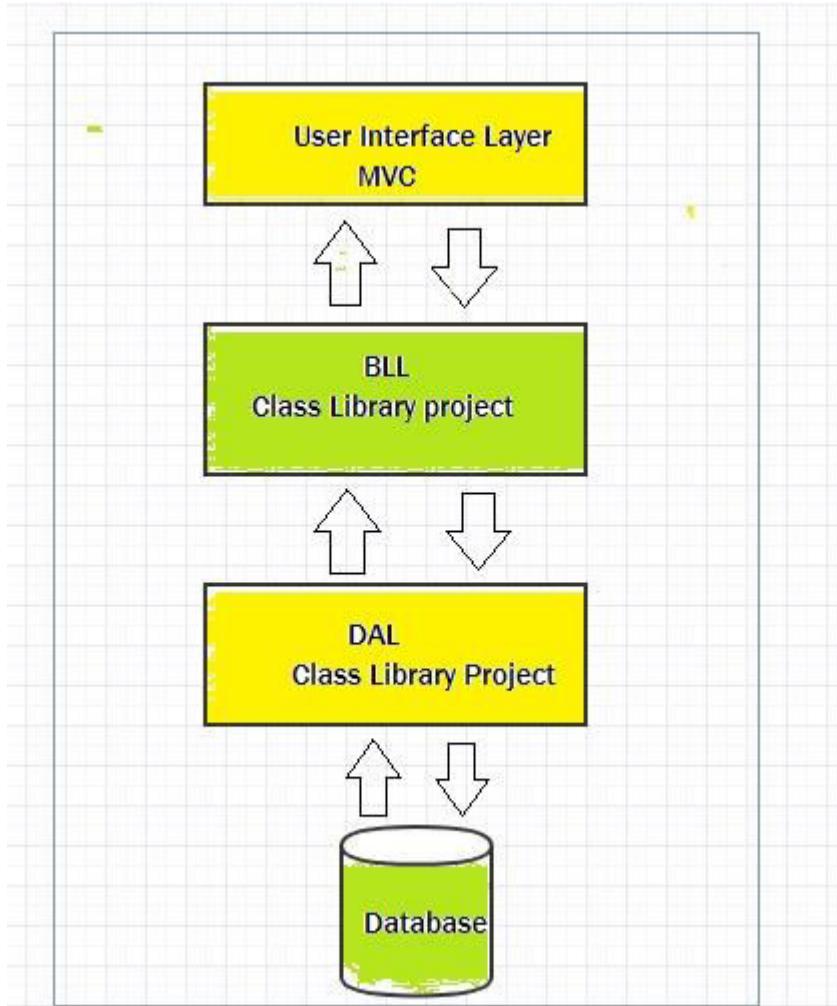
So lets see what exactly Circular dependency is and how it will affect the project architecture .



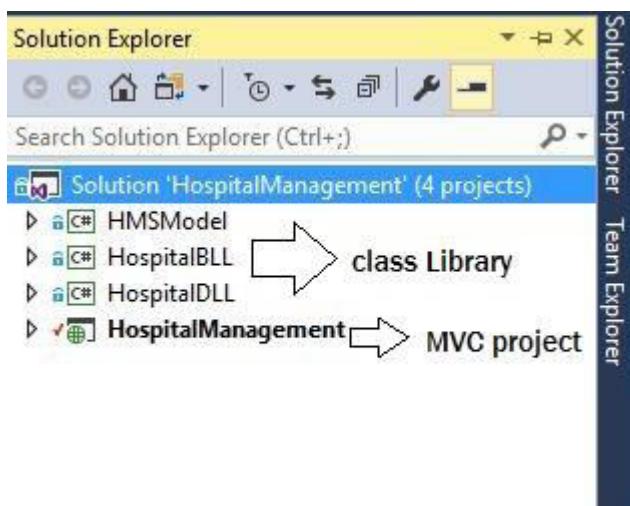
If you try to reference two class libraries with each other,then Visual Studio throws circular reference error.So while working on project please avoid this scenario.

If we have 2 layers in the project,The Layer-1 is reference to the Layer-2 ,that mean some resources of layer-2 will be consumed by Layer-1, similarly at any point of time Layer-2 wants any resources from Layer-1 then we need to reference it to layer-1.

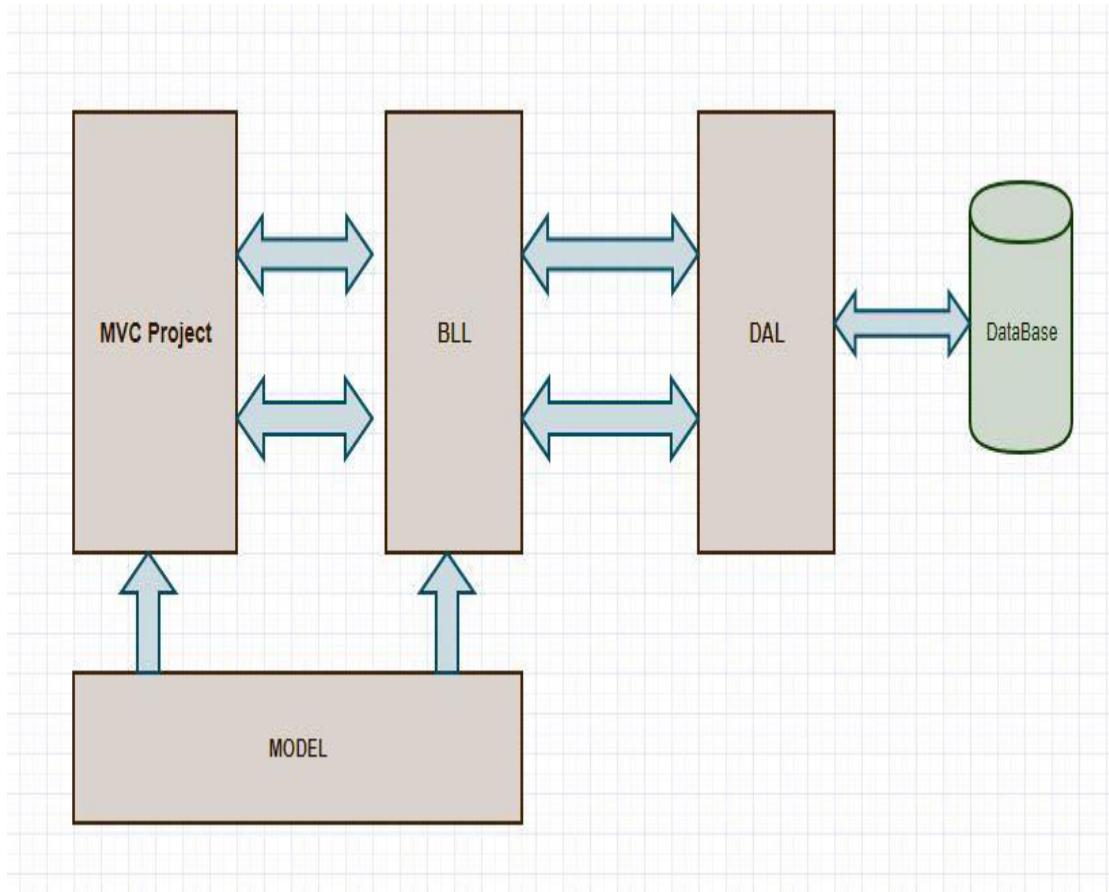
If we will do this a Circular dependency error will arise,so to overcome it we need to create an intermediate module between the 2 layers.This will be clearly describe while start working on project. Here is how our project flow look alike.



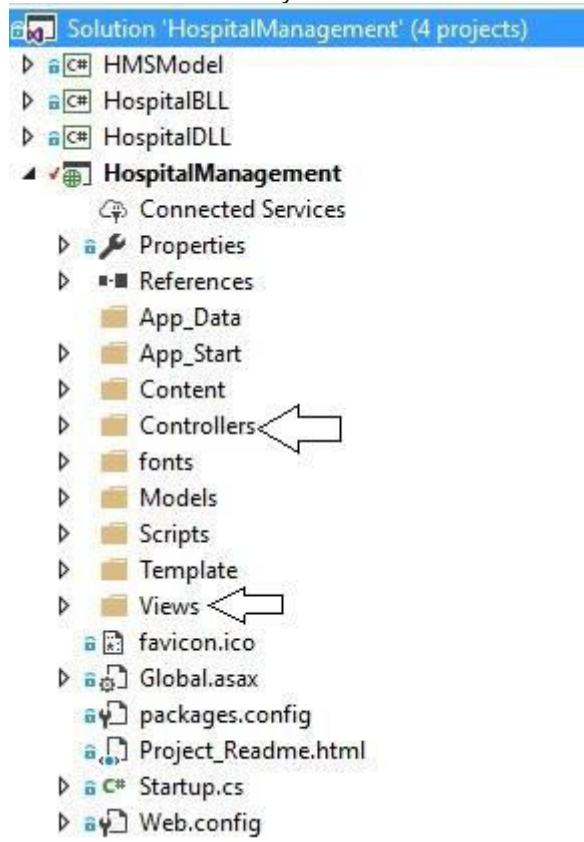
So as per the above sketch diagram i have added the projects layer as shown below.



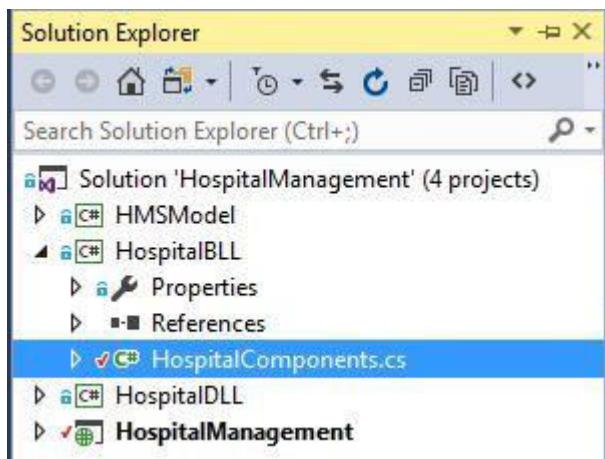
In this diagram you can find out how the project layers are related with each other,



Here is the UI or MVC layer of the application. In this you will find a Template Folder which basically contain a template a Home Page for this project. If you want there are several templates available in internet which you can download and use as for your project.



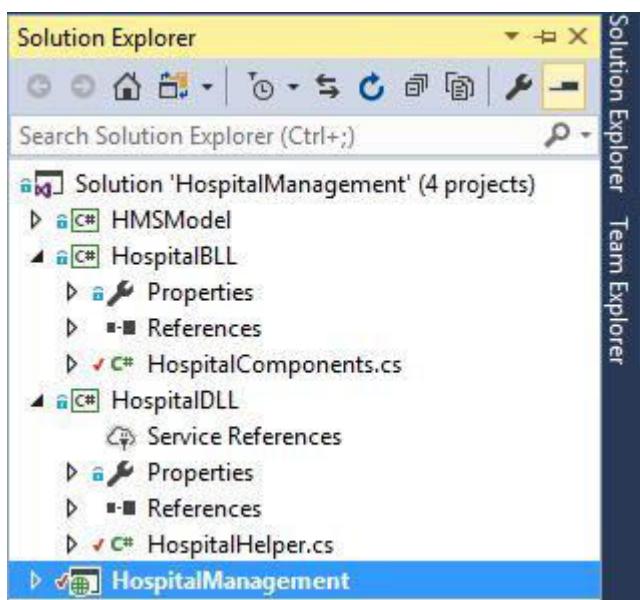
I have a BusinessLogicLayer called (**HospitalBLL**). This Layer is the intermediate layer between the main Project and the DataAccess Layer(**HospitalDLL**). The BusinessLogicLayer contains all the Business rules, The business logic layer (BLL) contains logic specific to the business domain. Also Business logic is used to maximize reuse opportunities.



As it is very specific to the business we will work on it as per our requirement. Here in this diagram I have shown the business component. We will start coding on it while interacting with the project.

I have added a class as HospitalComponent.cs, but basically the BLL should be based on mainly Controller requirement. A single controller can have one BLL or may be 2-3 controllers have one BLL component.

Now Lets check what is this Data Access Layer. Data Access Layer is mainly used to interact with the DB, so its contains all the operation related to the DataBase.



Hide Shrink ▲ Copy Code

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Data;
using System.Data.SqlClient;
using System.Configuration;
using HMSMODEL;

namespace HospitalDLL
{
    public class HospitalHelper
    {
        SqlCommand cmd;
        DataTable dt;
```

```

SqlDataAdapter ad;
public static SqlConnection connect()
{
    string myconnection =
ConfigurationManager.ConnectionStrings["connectHMS"].ToString();
    SqlConnection connection = new SqlConnection(myconnection);
    if(connection.State==ConnectionState.Open)
    {
        connection.Close();
    }
    else
    {
        connection.Open();
    }
    return connection;
}

public DataTable getData( string str)
{
    ad = new SqlDataAdapter(str, HospitalHelper.connect());
    dt = new DataTable();
    ad.Fill(dt);
    return dt;
}

public bool DML(SqlCommand cmd)
{
    int result = 0;
    using (connect())
    {

        using (cmd)
        {
            cmd.Connection = connect();

            result = cmd.ExecuteNonQuery();
        }
    }

    if(result>0)
    {
        return true;
    }
    else
    {
        return false;
    }
}

public DataTable GetAll(SqlCommand cmd)
{
    ad = new SqlDataAdapter();
    DataTable dt = new DataTable();

    using (connect())
    {

        using (cmd)
        {
            cmd.Connection = connect();
            ad.SelectCommand = cmd;

            ad.Fill(dt);
        }
    }
}

```

```

        return dt;

    }

}

}

```

Project Description:

Iclinic is a platform which bridges the gap between the doctors, patients and clinic. This platform is concerned about end to end consultation between the doctors and patients via the medium of digitating clinical Ticketing system.

In order to achieve this facility we have provided a portal where a patient can register his symptoms, get a online ticket, later can consult with any of the collabarated clinic. During his/her visit to clinic they need to quote the registration id which will be later process by doctor.

Here in this article we will mainly focus on creating Home page, Master entry forms, etc. Here we have a dynamic Home Page where we have (Slider, dynamic content update, etc).

Let us see the creating of Home Page with Slider.

Lets check the steps to create a Carousal slider.

A carousal slider actually contains 3 parts.

1. The Banners
2. The Indecators
3. The Controls

Indicators are specific circular Icons which the sliding/fading kind of effects inside the banner. These are represented as a floating content over the banner images. Indecators are represented in a ordered list manner. We need to fix the first list item as active, thus making the slidding effect to be started from the first image and later slidded to second, third and soon.

[Hide](#) [Copy Code](#)

```

<ol class="carousel-indicators">
    <li data-target="#responsive-slider" data-slide-to="0"
class="active"></li>
    <li data-target="#responsive-slider" data-slide-to="1"></li>
    <li data-target="#responsive-slider" data-slide-to="2"></li>
    <li data-target="#responsive-slider" data-slide-to="3"></li>
    <li data-target="#responsive-slider" data-slide-to="4"></li>
</ol>

```

As we have thought to fit 3 images in banner for that purpose we have created 3 indecators, the process of which is described above.

The **Banner** area contains a set of images which will be shown to the users with an animated effect. We can give as much images as we want for the banner area, which will be later processed by indecators. The process is shown below.

```

<div class="slides" data-group="slides">
    <ul>
        <li>
            <div class="slide-body" data-group="slide">
                
                <div class="caption header" data-
animate="slideAppearUpToDown" data-interval="20" data-length="300">

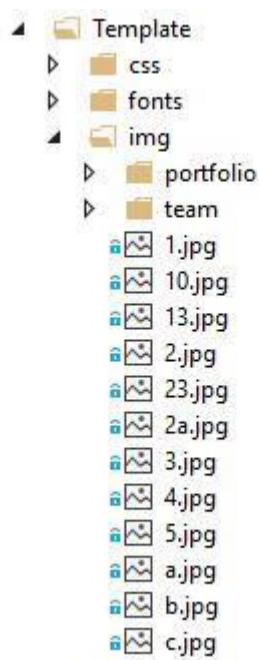
                    </div>
            </li>
        <li>
            <div class="slide-body" data-group="slide">
                
                <div class="caption header" data-
animate="slideAppearDownToUp" data-interval="20" data-length="300">

                    </div>
            </div>
        </li>
        <li>
            <div class="slide-body" data-group="slide">
                
                <div class="caption header" data-
animate="slideAppearUpToDown" data-interval="20" data-length="300">

                    </div>
            </div>
        </li>
        <li group="slide">
            <div class="slide-body" data-
alt="">
                
                <div class="caption header" data-
animate="slideAppearUpToDown" data-interval="20" data-
length="300">
                    </div>
            </div>
        </li>
    </ul>
</div>

```

The first part contain all the slider image with the delay time mentioned in "**data-interval**". Here is the image references inside the project.



The final step is to create controls for the carousel. This can be achived by using a class called "data-

slide" with specifically two arguments "previous & next". When being rendered, the content is shown as two indicators over the banner. The process is shown below-

Hide Copy Code

```
<a class="slider-control left" href="#" data-jump="prev"><i class="fa fa-angle-left fa-2x"></i></a>
<a class="slider-control right" href="#" data-jump="next"><i class="fa fa-angle-right fa-2x"></i></a>
```

As we can see, two controls as left and right angle icon, are being shown over the banner.

The complete corousel format is being shown below, for further clarification.

Hide Shrink ▲ Copy Code

```
<!-- Responsive slider - START -->
<div class="slider">
    <div class="container">
        <div class="row">
            <div class="responsive-slider" data-spy="responsive-slider" data-autoplay="true">
                <ol class="carousel-indicators">
                    <li data-target="#responsive-slider" data-slide-to="0" class="active"></li>
                    <li data-target="#responsive-slider" data-slide-to="1"></li>
                    <li data-target="#responsive-slider" data-slide-to="2"></li>
                </ol>
                <div class="slides" data-group="slides">
                    <ul>
                        <li>
                            <div class="slide-body" data-group="slide">
                                
                                <div class="caption header" data-animate="slideAppearUpToDown" data-interval="20" data-length="300">
                                    </div>
                            </div>
                        </li>
                        <li>
                            <div class="slide-body" data-group="slide">
                                
                                <div class="caption header" data-animate="slideAppearDownToUp" data-interval="20" data-length="300">
                                    </div>
                            </div>
                        </li>
                        <li>
                            <div class="slide-body" data-group="slide">
                                
                                <div class="caption header" data-animate="slideAppearUpToDown" data-interval="20ssssss" data-length="300">
                                    </div>
                            </div>
                        </li>
                    </ul>
                </div>
            <div class="slider-control left" href="#" data-jump="prev"><i class="fa fa-angle-left fa-2x"></i></a>
                <div>
                    <a class="slider-control right" href="#" data-jump="next"><i class="fa fa-angle-right fa-2x"></i></a>
                </div>
            </div>
        </div>
    </div>
<!-- Responsive slider - END -->
```

Actually in this project we have used a template, if you want to use you can download any template and use in your project.

Now our next thing is to create a dynamic notification which will give information to the user in the Home Page.

Here is the table for the above requirements.

[Hide](#) [Copy Code](#)

```
USE [HMS]
GO

/******** Object: Table [dbo].[tbl_Template]      Script Date: 04-09-2017 09:54:47 *****/
SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO

CREATE TABLE [dbo].[tbl_Template](
    [Header] [nchar](250) NOT NULL,
    [Status] [bit] NOT NULL,
    [id] [int] IDENTITY(1,1) NOT NULL,
    [Description] [nvarchar](max) NULL
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]

GO
```

Now create a multipurpose stored procedure to perform all functionality.

[Hide](#) [Copy Code](#)

```
USE [HMS]
GO
/******** Object: StoredProcedure [dbo].[sp_SaveTemplates]      Script Date: 04-09-2017 11:17:16 *****/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
ALTER procedure [dbo].[sp_SaveTemplates]
(
@Header nvarchar(300),
@Description nvarchar(max),
@status bit
)
AS
Begin
    INSERT INTO tbl_Template(Header,Description,Status) VALUES(@Header, @Description,@Status)
End
```

Template entry Form:- This form is used to dynamically entry and template for the patient and shows there details in Home page.

So now we will have the following form for template entry. Here is the image given below.

The screenshot shows a web application interface. At the top, there's a header bar with a back arrow, forward arrow, refresh icon, and a search bar containing 'localhost:5652/Home/TemplateEntry'. To the right of the search bar are icons for search, star, and more. Below the header is a navigation bar with links for 'Iclinic', 'Home', 'About', 'Contact', 'Register', and 'Log in'. The main content area has a title 'Template Details Entry Form'. Inside, there are three input fields: 'Header' (a white text area), 'Description' (a larger white text area), and 'Status' (a small checkbox). At the bottom right of the form is a blue 'Create' button.

[Back to List](#)

© 2017 - Iclinic

Here is the code for the following view.

[Hide](#) [Shrink](#) [Copy Code](#)

```
@model HMSMODEL.Template

@{
    ViewBag.Title = "TemplateEntry";
}

<script src="https://code.jquery.com/jquery-1.11.1.min.js"></script>

<script src="https://cdn.datatables.net/1.10.4/js/jquery.dataTables.min.js"></script>
<link rel="stylesheet" href="https://cdn.datatables.net/1.10.4/css/jquery.dataTables.min.css">

<style type="text/css">
    .textbox {
        width: 600px;
        height: 100px
    }

    .Checkbox {
        width: 20px;
        height: 20px;
    }
</style>
```

```

        display: block;
        background: url("link_to_image");
    }
</style>
<style type="text/css">
    .panel-body {
        background-color: #C0C0C0;
    }
</style>

@using (Html.BeginForm("TemplateEntry", "Home", FormMethod.Post))
{
    @Html.AntiForgeryToken()

    <div class="panel panel-default">
        <div class="panel-heading">
            <h3 class="panel-title"><b>Template Details Entry Form</b></h3>
        </div>
        <div class="panel-body">

            <div class="row; ba">
                <div class="span6">

                    @Html.ValidationSummary(true)

                    <div class="form-group">
                        @Html.LabelFor(model => model.Header, new { @class = "control-label col-md-2" })
                        <div class="col-md-10">
                            @Html.EditorFor(model => model.Header)
                            @Html.ValidationMessageFor(model => model.Header)
                        </div>
                    </div>

                    <br />
                    <br />

                    <div class="form-group">
                        @Html.LabelFor(model => model.Description, new { @class = "control-label col-md-2" })
                        <div class="col-md-10">

                            @Html.TextAreaFor(model => model.Description, new { @class = "textbox" })

                            @Html.ValidationMessageFor(model => model.Description)
                        </div>
                    </div>
                    <br />
                    <br />
                    <div class="form-group">
                        @Html.LabelFor(model => model.Status, new { @class = "control-label col-md-2" })
                        <div class="col-md-10">
                            @Html.CheckBoxFor(model => model.Status, new { @class = "checkbox" })

                            @Html.ValidationMessageFor(model => model.Status)
                        </div>
                    </div>

                    <br />
                    <br />
                    <br />

                    <div class="form-group">
                        <br />
                        <br />
                        <div class="col-md-offset-2 col-md-10">
                            <input type="submit" value="Create" class="btn btn-info" />
                        </div>
                    </div>
                </div>
            </div>
        </div>
    </div>
}

```

```

        </div>
    </div>

}

<div>
    @Html.ActionLink("Back to List", "Index")
</div>

```

Here is the Home controller method where data is posted for saving.

[Hide](#) [Copy Code](#)

```
HospitalBLL.HospitalComponents obj = new HospitalBLL.HospitalComponents();
```

This is the object of our business Logic Layer.

And Here is the Template entry GET and POST Method.

[Hide](#) [Copy Code](#)

```

public ActionResult TemplateEntry()
{
    return View();
}

[HttpPost]
public ActionResult TemplateEntry(Template temp)
{
    bool x = obj.tabEntry(temp);
    if(x==true)
    {
        ViewBag.result = "Data Saved Successfully";
        return View();
    }

    else
    {
        return View();
    }
}

```

Now here is the HMS Model layer where we have the Template Classss.

[Hide](#) [Copy Code](#)

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HMSCODEL
{
    public class Template

```

```

    {
        public int id { get; set; }
        public string Header { get; set; }
        public bool Status { get; set; }
        public string Description { get; set; }
    }
}

```

Now this is the "tabEntry" method on HospitalBLL.

[Hide](#) [Copy Code](#)

```

public bool tabEntry(Template temp)
{
    cmd = new SqlCommand("sp_SaveTemplates");
    cmd.CommandType = CommandType.StoredProcedure;
    cmd.Parameters.AddWithValue("@Header", temp.Header);
    cmd.Parameters.AddWithValue("@Description", temp.Description);
    cmd.Parameters.AddWithValue("@Status", temp.Status);
    bool result= _hospitalhelper.DML(cmd);
    if(result==true)
    {
        return true;
    }
    else
    {
        return false;
    }
}

```

Template Details Entry Form

Header	Free Heart CheckUp
Description	Free Heart checkup on 15-07-2017 by Dr.Ritika agrawal So please come to Agarwal Hospital
Status	<input type="checkbox"/>
<input type="button" value="Create"/>	

Now after creating templates if you want to fetch all template in a dynamic manner,you have the following code.

[Hide](#) [Copy Code](#)

```

HospitalBLL.HospitalComponents obj = new HospitalBLL.HospitalComponents();
DataTable dt;

public ActionResult getData()
{

```

```

        dt = new DataTable();

        dt = obj.getTemplateData();

        List<Template> list = new List<Template>();

        for (int i = 0; i < dt.Rows.Count; i++)
        {
            Template temp1 = new Template();
            temp1.id = Convert.ToInt32(dt.Rows[i]["id"]);
            temp1.Header = dt.Rows[i]["Header"].ToString();
            temp1.Description = dt.Rows[i]["Description"].ToString();
            list.Add(temp1);
        }
        var data = list;
        return Json(new { data = data }, JsonRequestBehavior.AllowGet);
    }
}

```

now here is how we will design the view to show the Templates in index page.

[Hide](#) [Shrink](#) [Copy Code](#)

```

@model IEnumerable<HMSMODEL.Template>

 @{
     ViewBag.Title = "ViewTemplate";
 }

<script src="~/Scripts/jquery-1.10.2.min.js"></script>
<br />

<div style="width:90%; margin:0 auto; background-color:blanchedalmond">
    <table id="myTable">
        <thead>
            <tr>
                <th>ID</th>
                <th>Header</th>
                <th>Description</th>
            </tr>
        </thead>
        </table>
    </div>

    /* CSS for the DataTable */
<link href="//cdn.datatables.net/1.10.9/css/jquery.dataTables.min.css" rel="stylesheet" />

<style>
    tr.even {
        background-color:blueviolet;
    }
</style>
/* Load datatable js */
@section Scripts{
    <script src="//cdn.datatables.net/1.10.9/js/jquery.dataTables.min.js"></script>
    <script>
        $(document).ready(function () {
            $('#myTable').DataTable({
                "ajax": {
                    "url": "/home/getData",
                    "type": "GET",
                    "datatype": "json"
                },
                "columns" : [
                    { "data": "id", "autoWidth": true },
                    { "data": "Header", "autoWidth": true },
                    { "data": "Description", "autoWidth": true }
                ]
            });
        });
    </script>
}

```

```

        ],
        select: true

    });

});

</script>
}

```

ID	Header	Description
1	Consult a doctor online	Historically, undergraduate medical education focused on the diagnostic process. After qualifying, doctors were expected to refine these skills but also to develop knowledge and skills with a greater
2	View Our History	Find out if your doctor, hospital, drug store, lab, or health insurance company offers Blue Button. Although Blue Button is in its early stages, it is expanding rapidly. If your doctor does not par
3	Order Medicine	It may still be in clinical trial development, or have finished clinical trials but yet to be commercially launched; the manufacturer may not have proceeded with licensing in your country or the medic
4	Contact Us	HomeCountact Us. We would love to hear from you. If you need to contact us for any feedback or suggestion, please do fill the adjoining form and we will get ...
8	Free Heart CheckUp	Free Heart checkup on 15-07-2017 by Dr.Ritika agrawal. So please come to Agarawal Hospital.

Showing 1 to 5 of 5 entries

1

To design this we have used Jquery DataTable. We have enabled paging, sorting, searching functionality as shown below.

Show 10 entries		Search: <input type="text"/>
ID *	Header	Description
4	Contact Us	HomeCountact Us. We would love to hear from you. If you need to contact us for any feedback or suggestion, please do fill the adjoining form and we will get...
8	Free Heart CheckUp	Free Heart checkup on 15-07-2017 by Dr Ritika agrawal So please come to Agarawal Hospital.
Showing 1 to 2 of 2 entries (filtered from 5 total entries)		
Previous 1 Next		

© 2017 - iClinic

patient Entry form:This is the form where patient entry their details to get consulted with a doctor.

Now lets desing a patient Entry form as follow.

Patients Details Entry Form

FirstName	<input type="text"/>
LastName	<input type="text"/>
Age	<input type="text"/>
Address	<input type="text"/>
Gender	Male: <input type="radio"/> Female: <input type="radio"/>
EntryFee	<input type="text"/>
Description	<input type="text"/>
<input type="button" value="SAVE"/> <input type="button" value="SHOW"/>	

Here is the complete View code.

Hide Shrink ▲ Copy Code

```
@model HMSMODEL.PatientEntry

@{
    ViewBag.Title = "PatientsEntry";
}


<link href="~/Template/css/bootstrap.min.css" rel="stylesheet" />
<style type="text/css">
    .panel-body {
        background-color:#C0C0C0;
    }
</style>

@{
    if ( TempData["alertMessage"]!=null)
}
```

```

    {
        <script type="text/javascript">
            alert("@ TempData["alertMessage"]");
        </script>
    }
    else
    {
    }

}

<h4 style="text-align:center;color:aquamarine"></h4>

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <br />

    <div class="panel panel-default">
        <div class="panel-heading">
            <h3 class="panel-title">Patients Details Entry Form</h3>
        </div>
        <div class="panel-body">

            <div class="row; ba">
                <div class="span6">

                    @Html.ValidationSummary(true)
                    <div class="form-group">
                        @Html.LabelFor(model => model.FirstName, htmlAttributes: new { @class = "control-label col-md-2" })
                        <div class="col-md-10">
                            @Html.EditorFor(model => model.FirstName, new { @class = "form-control", id = "txt_firstname" })
                            @Html.ValidationMessageFor(model => model.FirstName)
                        </div>
                    </div>
                    <br />
                    <br />
                    <div class="form-group">
                        @Html.LabelFor(model => model.LastName, htmlAttributes: new { @class = "control-label col-md-2" })
                        <div class="col-md-10">
                            @Html.EditorFor(model => model.LastName, new { @class = "form-control" })
                            @Html.ValidationMessageFor(model => model.LastName)
                        </div>
                    </div>
                    <br />
                    <div class="form-group">
                        @Html.LabelFor(model => model.Age, htmlAttributes: new { @class = "control-label col-md-2" })
                        <div class="col-md-10">
                            @Html.EditorFor(model => model.Age, new { @class = "form-control" })
                            @Html.ValidationMessageFor(model => model.Age)
                        </div>
                    </div>
                    <br />
                    <div class="form-group">
                        @Html.LabelFor(model => model.Address, htmlAttributes: new { @class = "control-label col-md-2" })
                        <div class="col-md-10">
                            @Html.EditorFor(model => model.Address, new { @class = "form-control" })
                        </div>
                    </div>
                </div>
            </div>
        </div>
    </div>
}

```

```

                @Html.ValidationMessageFor(model => model.Address)
            </div>
        </div>
        <br />
        <div class="form-group">
            @Html.LabelFor(model => model.Gender, htmlAttributes: new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                Male: @Html.RadioButton("Gender", "Male")
                Female: @Html.RadioButton("Gender", "Female")
            </div>
        </div>
        <br />
        <div class="form-group">
            @Html.LabelFor(model => model.EntryFee, htmlAttributes: new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.EntryFee, new { @class = "form-control" })
                @Html.ValidationMessageFor(model => model.EntryFee)
            </div>
        </div>
        <br />

        <div class="form-group">
            @Html.LabelFor(model => model.Description, htmlAttributes: new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.TextAreaFor(model => model.Description, new { @class = "form-control" })
                @Html.ValidationMessageFor(model => model.Description)
            </div>
        </div>
        <br />
        <br />

        <div class="form-group">
            <br />
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="SAVE" class="btn btn-info" onclick="location.href='@Url.Action("PatientsEntry", "PatientsEntry")'" />
                <input type="submit" id="btn_Show" value="SHOW" class="btn btn-info" onclick="location.href='@Url.Action("patientDetails", "PatientsEntry")'" />
            </div>
        </div>
    </div>

```

}

```

<div>
    @Html.ActionLink("Back to List", "Index")
</div>

```

Now here is the Patient entry controller for all functions.

Hide Shrink  Copy Code

```

using System;
using System.Collections.Generic;

```

```
using System.Linq;
using System.Web;
using System.Web.Mvc;
using HMSMODEL;
using System.Data;

namespace HospitalManagement.Controllers
{
    public class PatientEntryController : Controller
    {
        HospitalBLL.HospitalComponents obj = new HospitalBLL.HospitalComponents();
        DataTable dt;

        // GET: PatientEntry
        public ActionResult PatientsEntry()
        {

            return View();
        }
        [HttpPost]
        public ActionResult PatientsEntry(PatientEntry patient)
        {
            if (ModelState.IsValid)
            {

                bool details = obj.PatientEntry(patient);
                if (details == true)
                {
                    TempData["alertMessage"] = "Success!!!";
                    ModelState.Clear();

                    return View();
                }
                else
                {
                    ViewBag.SuccessMessage = "<p>Please try once!!!!</p>";
                    return View();
                }
            }
            else
            {
                ViewBag.SuccessMessage = null;
                return View();
            }
        }

        public ActionResult patientDetails()
        {

            dt = new DataTable();

            dt= obj.GetPatientEntry_onParticularDate();

            List<PatientEntry> list = new List<PatientEntry>();

            for (int i = 0; i < dt.Rows.Count; i++)
            {
                PatientEntry patient = new PatientEntry();
                patient.ID = dt.Rows[i]["ID"].ToString();
                patient.FirstName = dt.Rows[i]["FirstName"].ToString();
                patient.LastName = dt.Rows[i]["LastName"].ToString();
                patient.Date = Convert.ToDateTime(dt.Rows[i]["date"]);
                patient.Address = dt.Rows[i]["Address"].ToString();
                patient.Age = Convert.ToInt32(dt.Rows[i]["Age"]);
                patient.Description = dt.Rows[i]["Description"].ToString();
            }
        }
    }
}
```

```
        list.Add(patient);
    }

    return View(list);

}
}
```

The model is given here

[Hide](#) [Copy Code](#)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HMSMODEL
{
    public class PatientEntry
    {
        public string ID { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int Age { get; set; }
        public string Address { get; set; }
        public string Description { get; set; }
        public DateTime Date { get; set; }
        public int EntryFee { get; set; }
        public string Gender { get; set; }
    }
}
```

Here is the complete Business Logic for both Template entry and Patient Entry.

[Hide](#) [Shrink](#) [Copy Code](#)

```
using HMSMODEL;
using HospitalDLL;
using System;
using System.Collections.Generic;
using System.Data;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Data;
using System.Data.SqlClient;

namespace HospitalBLL
{
    public class HospitalComponents
    {

        HospitalHelper _hospitalhelper = new HospitalHelper();
        DataTable dt = new DataTable();
        SqlCommand cmd;

        public DataTable getTemplateData()
        {
            dt = _hospitalhelper.getData("select id ,Header ,LEFT(Description,200)
as  Description from dbo.tbl_Template");
            if (dt.Rows.Count > 0)
            {

                return dt;
            }
        }
}
```

```

        else
    {
        return dt;
    }

}

public bool tabEntry(Template temp)
{

    cmd = new SqlCommand("sp_SaveTemplates");
    cmd.CommandType = CommandType.StoredProcedure;
    cmd.Parameters.AddWithValue("@Header", temp.Header);
    cmd.Parameters.AddWithValue("@Description", temp.Description);
    cmd.Parameters.AddWithValue("@Status", temp.Status);
    bool result= _hospitalhelper.DML(cmd);
    if(result==true)
    {
        return true;
    }

    else
    {
        return false;
    }
}

public bool editTemplate(Template temp)
{
    cmd = new SqlCommand("sp_EditTemplates");
    cmd.CommandType = CommandType.StoredProcedure;
    cmd.Parameters.AddWithValue("@Header", temp.Header);
    cmd.Parameters.AddWithValue("@Description", temp.Description);
    cmd.Parameters.AddWithValue("@Status", temp.Status);
    cmd.Parameters.AddWithValue("@id", temp.id);
    _hospitalhelper.DML(cmd);
    return true;
}

public bool PatientEntry(PatientEntry patent)
{
    patent.ID = Guid.NewGuid().ToString();

    cmd = new SqlCommand("SP_PatientEntry");
    cmd.CommandType = CommandType.StoredProcedure;
    cmd.Parameters.AddWithValue("@EntryId", patent.ID);
    cmd.Parameters.AddWithValue("@FirstName", patent.FirstName);
    cmd.Parameters.AddWithValue("@LastName", patent.LastName);
    cmd.Parameters.AddWithValue("@Age", patent.Age);
    cmd.Parameters.AddWithValue("@Address", patent.Address);
    cmd.Parameters.AddWithValue("@Description", patent.Description);
    cmd.Parameters.AddWithValue("@EntryFee",patent.EntryFee);
    cmd.Parameters.AddWithValue("@Gender",patent.Gender);

    cmd.Parameters.AddWithValue("@Date", DateTime.Today);

    cmd.Parameters.AddWithValue("@Opptype", "Save");
    bool result= _hospitalhelper.DML(cmd);
    return result;
}

public DataTable GetPatientEntry_onParticularDate()
{
    dt = new DataTable();
    cmd = new SqlCommand("SP_PatientEntry");
    cmd.CommandType = CommandType.StoredProcedure;
    cmd.Parameters.AddWithValue("@Opptype", "Getall");
    dt = _hospitalhelper.GetALL(cmd);
    return dt;
}

```

```
}
```

This is the table structure for patient Entry.

Hide Shrink ▲ Copy Code

```
USE [HMS]
GO

/******** Object: Table [dbo].[PatientEntry]      Script Date: 04-09-2017 20:05:15 *****/
SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO

CREATE TABLE [dbo].[PatientEntry](
    [Id] [int] IDENTITY(1,1) NOT NULL,
    [EntryId] [nvarchar](150) NOT NULL,
    [FirstName] [nvarchar](50) NULL,
    [LastName] [nvarchar](50) NULL,
    [Age] [int] NULL,
    [Address] [nvarchar](350) NULL,
    [Description] [nvarchar](max) NULL,
    [Date] [date] NULL,
    [EntryFee] [int] NULL,
    [Gender] [nvarchar](10) NULL,
    CONSTRAINT [PK_PatientEntry] PRIMARY KEY CLUSTERED
    (
        [EntryId] ASC
    )WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON,
ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]

GO
```

Here is the Stored Procedure for all Patient Related operation.

Hide Shrink ▲ Copy Code

```
USE [HMS]
GO
/******** Object: StoredProcedure [dbo].[SP_PatientEntry]      Script Date: 04-09-2017 20:03:37 *****/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
ALTER PROCEDURE [dbo].[SP_PatientEntry]
(
    @EntryId as nvarchar(150)=null,
    @FirstName as nvarchar(50)=null,
    @LastName as nvarchar(50)=null,
    @Age as int=null,
    @Address as nvarchar(350)=null,
    @Description as nvarchar(max)=null,
    @Date as Date=null,
    @EntryFee as int=null,
    @Gender as nvarchar(10)=null,
    @Opptype as nvarchar(20)=null
)
AS
BEGIN
    if (@opptype='Save')
    begin
```

```

insert into
[dbo].[PatientEntry]([EntryId],[FirstName],[LastName],[Age],[Address],[Description],[Date],[EntryFee],[Gender])
values(@EntryId,@FirstName,@LastName,@Age,@Address,@Description,GETDATE(),@EntryFee,@Gender)

end
if(@Opptype='Getall')
begin
select * from [dbo].[PatientEntry]
end

END

```

Here is the code for the View. Here we have used HTML Grid to show the data.

[Hide](#) [Shrink](#) [Copy Code](#)

```

@model IEnumerable<HMSMODEL.PatientEntry>

 @{
    ViewBag.Title = "patientDetails";
}

<script src="~/Scripts/jquery-1.10.2.min.js"></script>
<script src="~/Scripts/bootstrap.min.js"></script>
<link href="~/Scripts/Calender/calendrical.css" rel="stylesheet" />
<script src="~/Scripts/Calender/jquery.calendrical.js"></script>
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js"></script>

<style type="text/css">
.webgrid-table {
    font-family: "Trebuchet MS", Arial, Helvetica, sans-serif;
    font-size: 1.2em;
    width: 300px;
    display: table;
    border-collapse: separate;
    border: solid 1px;
    background-color: purple;
}

.webgrid-table td, th {
    border: 3px solid;
    padding: 3px 7px 2px;
    width: 230px;
}

.webgrid-header {
    background-color: whitesmoke;
    color: #FFFFFF;
    padding-bottom: 4px;
    padding-top: 5px;
    text-align: left;
    width: 20%;
}

.webgrid-footer {
}

.webgrid-row-style {
    padding: 3px 7px 2px;
}

.webgrid-alternating-row {
    background-color: #EAF2D3;
    padding: 3px 7px 2px;
}
</style>

<script type="text/javascript">

$(function () {

```

```

$('.edit-mode').hide();
$('.edit-user, .cancel-user').on('click', function () {
    var tr = $(this).parents('tr:first');
    tr.find('.edit-mode, .display-mode').toggle();
});
$.ajax({
    url: '/Home/Update/',
    data: JSON.stringify(UserModel),
    type: 'POST',
    contentType: 'application/json; charset=utf-8',
    success: function (data) {
        window.location.href = window.location.href;
    }
});

});

})
</script>
<br />

<div>
@{
    var grid = new WebGrid(Model, canPage: true, rowsPerPage: 10);
    grid.Pager(WebGridPagerModes.All);
}

</div>
<h3>List of Patients</h3>

<div>
@grid.GetHtml(
    headerStyle: "webgrid-header",
    footerStyle: "webgrid-footer",
    alternatingRowStyle: "webgrid-alternating-row",
    selectedRowStyle: "webgrid-selected-row",
    rowStyle: "webgrid-row-style",
    mode: WebGridPagerModes.All,
    columns: grid.Columns(
        grid.Column("ID", format: @<text> <span class="display-mode">@item.ID </span>
<label id="lbl_Empid" class="edit-mode">@item.ID</label> </text>, style: "col1Width"),
        grid.Column(columnName: "FirstName", header: "First Name", format: @<text> <span class="display-mode">@item.FirstName <label id="lblFirstName"></label> </span> <input type="text" id="txt(firstName" value="@item.FirstName" class="edit-mode" /> </text>, style: "col2Width"),
        grid.Column(columnName: "LastName", header: "Last Name", format: @<text> <span class="display-mode">@item.LastName <label id="lbl_lastname"></label> </span> <input type="text" id="txt_lastname" value="@item.LastName" class="edit-mode" /> </text>, style: "col2Width"),
        grid.Column(columnName: "Description", header: "Description", format: @<text> <span class="display-mode">@item.Description <label id="lbl_desc"></label> </span> <input type="text" id="txt_desc" value="@item.Description" class="edit-mode" /> </text>, style: "col2Width"),
        grid.Column(columnName: "Address", header: "User Address", format: @<text> <span class="display-mode">@item.Address <label id="lbladdress"></label> </span> <input type="text" id="txt_address" value="@item.Address" class="edit-mode" /> </text>, style: "col2Width"),
        grid.Column("Action", format: @<text>
            <button class="edit-user display-mode" style="background-color: #1E90FF; border-bottom-style: groove">Edit</button>
            <button class="save-user edit-mode" style="background-color: #FF5733; border-bottom-style: groove">Save</button>
            <button class="cancel-user edit-mode" style="background-color: #FF5733; border-bottom-style: groove">Cancel</button>
        )
    )
}

```

```

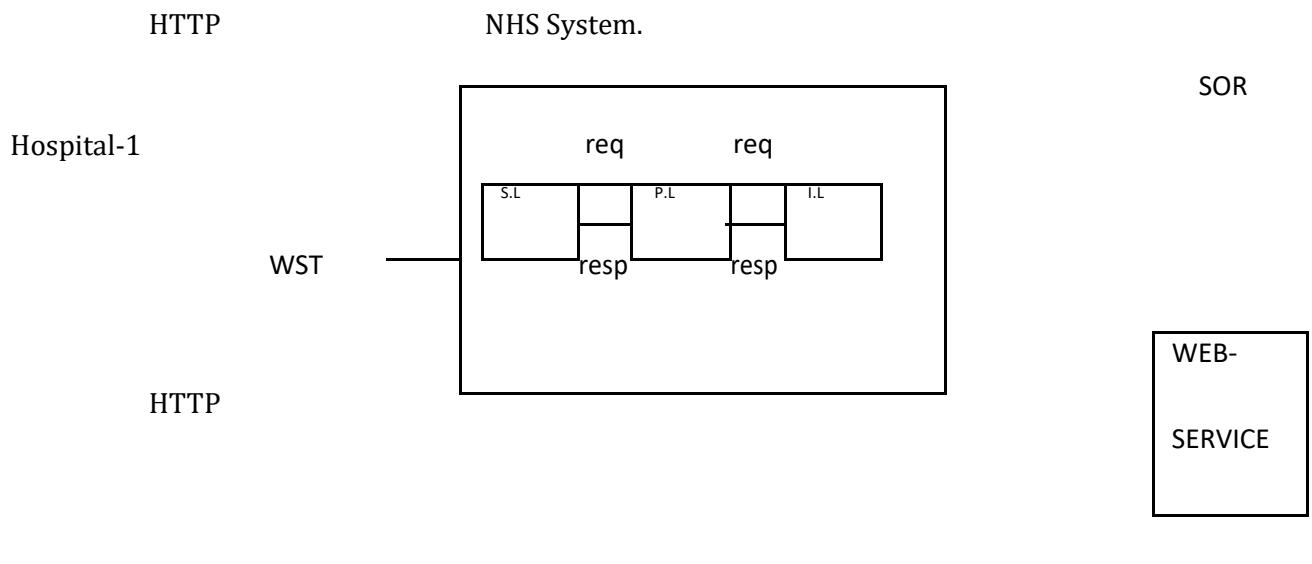
        <button class="Delete-user display-mode" style="background-color: #1E90FF;
border-bottom-style: groove">Delete</button>
    </text>, style: "col3Width", canSort: true)))

</div>

</div>

```

3. TELL ME ABOUT YOUR PROJECT ARCHITECTURE:



A part of my project architecture, I am working as provider side. In this provider side having three layer

A). Service Layer.

B). Process Layer.

C). Integration Layer.

Here our NHS interacted with the SOR(DataBase) system.

- Here with respect to the project architecture, Hospitals will send the HTTPs request, by this request going to the provider side (a) Service Layer (b) Process Layer (c) Integration Layer, in this service layer we are validating the data and calling the process layer & send it to the process layer, by this data we perform the converting point logic.

4. TELL ME ABOUT YOUR PROJECT TECHNICAL FLOW:

A part of my project technical flow, here first we are taking the business requirement from the client and by this business requirement we prepare the Data Mapping Sheet, and we prepare the Schema file, and by using this XSD file we generate the WSDL, and then create the Artifacts (creating the .java and .classes file).

And by the artifacts create Service layer implementation class, in this class calling the interface of whatever generated by the WSDL file.

Here is the Technical Flow of my project (Partner Integration).

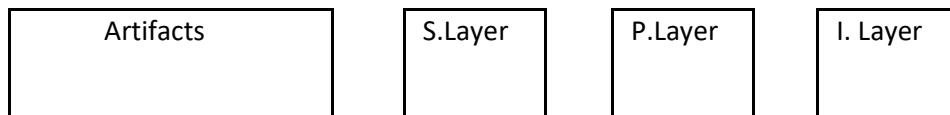
Business Requirement

Data Mapping Sheet

XSD Schema File,

WSDL File

SOR



Web-

services

Fig: - Technical flow of Spine(Patient Details) Service.

As of our project, here first we take the request and response element by the Data mapping sheet, by this we create the XSD files and then create the WSDL files also. After completion of this work we create the Artifacts and by this create the Layers like-

- A). Service Layer.
- B). Process Layer.
- C). Integration Layer.

In the Service layer, calling the interface by the implementation class and we prepare the web service request object and then building the request for process layer and then call the process layer, In process layer building the request for Integration layer and calling the Integration layer, In integration layer transfer the request data to the SOR system or web-service for storing the data permanent.

5. DATA Flow Diagram

5.1. Registration Module

5.2. Patient Details Module

5.3.Import and Export Modules

6. How you are handling exception in your project?

As part of my project we are handling the exception, generally in my project we have handle user defined exception, based on the business requirement we have categorized user defined exception into different types of exception like-

- A). Business Exception.
- B). System Exception.
- C). Service Exception.

Here, when the user sends the request element (like- fistName, LastName, D.O.B, MobNo and etc.) data, if that data is not valid then we are throwing the some user defined exception. This data not sending to the process layer, here only we are handle the exception.

- a) **Business Exception** :- Here business exception means if the User sending the request element , that request element is not valid then that time we handle the business exception. To handle this, here we are writing the some user defined exception class and some Enum class/properties files/XML files. By this class we handle the business exception. In business exception coming the some DataException this one also related to the request data only.

#64/3RT, SR Nagar, Near Community Hall, Hyderabad

Contacts: +91-8019697596, 040-40061799

- b) System Exception: - This Exception coming at the time of running the service, this one also related to the hardware system, this exception will be handling by the Admin team. This exception we handle in Throw able class exception.
- c) Service Exception: - In service exception we handle the service related exception like- DBConnection error.

In my project having layers like- service layer, process layer, Integration layer.

In Integration layer, as per business requirement we are categorized different kind of user defined exceptions.

Some exception we are throwing, the exception in process layer and service layer.

Here, I am explaining the some exceptions.

NumberFormatException, ClassCastException, ClassNotFoundException, FileNotFoundException, NullPointerException and etc.

NullPointerException :- To resolve the NullPointerException before creating the object we can write some if_else (for avoiding this exception) statement and checkout this, and make sure that object not null, if object whatever you pass is null then that time you will get NullPointerException.

ClassNotFoundException :- At the time of loading the class if that class is not available then that you will get the ClassNotFoundException.

To resolve this Exception you make sure whatever you load the class, that class is available. Another style of handle this exception is you should pass the class otherwise you will get this exception.

7. What are the exceptions you have used in project during deployment? A part

of our project we are facing the different type of exception like,

- A). OutOfMemoryException. B).
- ResourcesFailedException. C).
- NullPointerException, and etc.

A). OutOfMemoryException : - To handle this exception, we raise request/ticket to support/admin team. They will resolve issue and update the ticket, in stage server we handled or resolve this issue.

Here have some reasons.

(1). Heap Dumps.

(2). More no. of application. B).

ResourceFailedException: -

8. What is Encapsulation?Where you have used in project?

Encapsulation is the ability to package data. It is all about packaging related stuff together and hides them from external elements. Encapsulation importance here is for binding data and methods together.

How you are implemented polymorphism in your project?

Normally we are developing our application interface base approach for example service and serviceImpl so my controller need to inject serviceImpl to get business functionality so in this case in controller class we are injection service bean by taking service interface reference as below .

```

public interface BankService {
    public void doTransaction();
}

@Service
public class BankServiceImpl implements BankService{
    @Override
    public void doTransaction() {
        // LOGIC
    }
}

@Controller
public class BankController {
    @Autowired(required = true)
    private BankService service;
}

```

Here internally IOC container instantiate my service bean as below approach

```
BankService service=new BankServiceImpl(); //Runtime polymorphism
```

What is serialization? Have you implement serialization in your project?

Serialization is a process where we can change state of object to the file over the network or simply we can transfer our object from one layer to another layer, that's why java provides streaming API.

In my project I used in pojo class means my pojo class should be implements from Serializable interface because that business object will be transfer over the network, that's why it's recommended to implements BO object from Serializable interface.

How to create web-services project and spring project using maven?

Simply we have to create one maven project like maven-archetype-webapp then add the dependency from local repository if available else download from central repository in pom.xml.

Add spring dependency along with JAX-WS implementation class dependency in pom.xml

.how you implement exception handling in your project?

Answer:

Normally we are developing Spring based application so in Spring to handle exception multiple predefined class is there .so simply in my project we are throwing custom exception from service layer and when my controller call service it will catch that exception by using Spring

Aop , We have to create a class which should be annotated as @ControllerAdvice and we have to take one method whose return type is Model And View and method should be annotated as

@ExceptionHandler so in this method we have to write the logic for map the exception. And return the same view which is return by controller class at the time of exception raise. And we have to return some user understandable message by view page.

Note: Both return logical view should be same

Introduction to Spring Batch

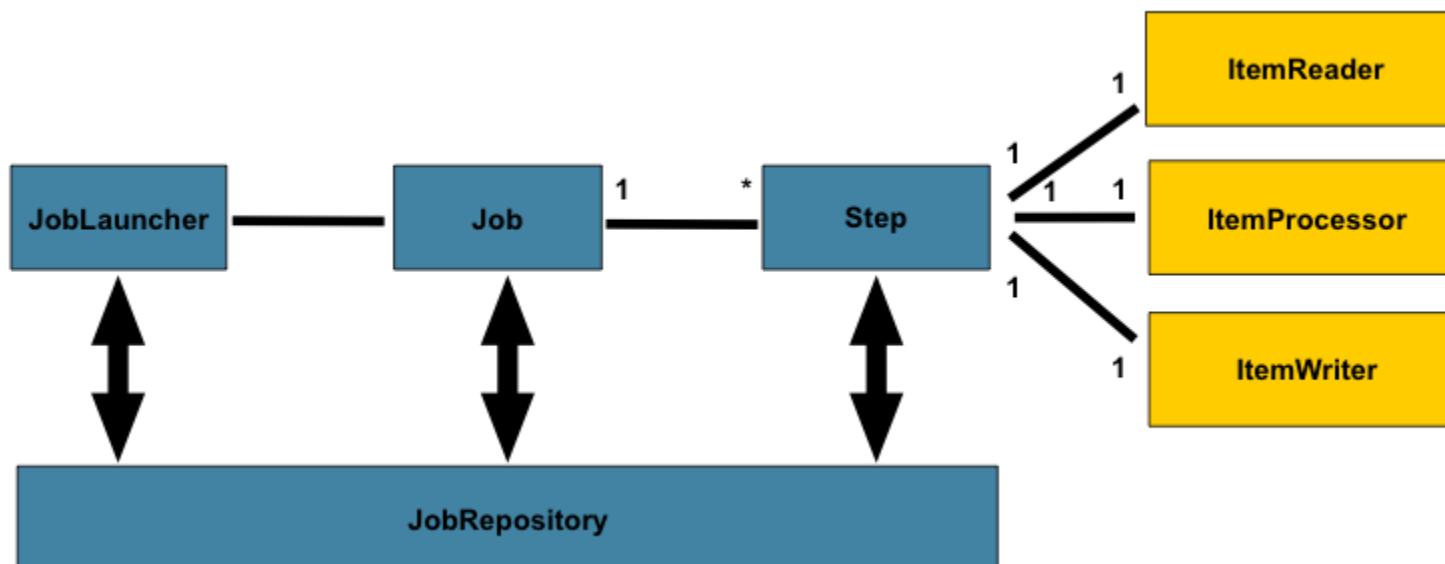
1. Introduction

In this article we're going to focus on a practical, code-focused intro to Spring Batch. Spring Batch is a processing framework designed for robust execution of jobs.

It's current version 3.0, which supports Spring 4 and Java 8. It also accommodates JSR-352, which is new java specification for batch processing.

[Here are](#) a few interesting and practical use-cases of the framework.

2. Workflow Basics



Spring batch follows the traditional batch architecture where a job repository does the work of scheduling and interacting with the job.

A job can have more than one steps – and every step typically follows the sequence of reading data, processing it and writing it.

And of course the framework will do most of the heavy lifting for us here – especially when it comes to the low level persistence work of dealing with the jobs – using *sqlite* for the job repository.

2.1. Our Example Usecase

The simple usecase we’re going to tackle here is – we’re going to migrate some financial transaction data from CSV to XML.

The input file has a very simple structure – it contains a transaction per line, made up of: a username, the user id, the date of the transaction and the amount:

```
1  username, userid, transaction_date, transaction_amount
2  devendra, 1234, 31/10/2015, 10000
3  john, 2134, 3/12/2015, 12321
4  robin, 2134, 2/02/2015, 23411
```

3. The Maven pom

Dependencies required for this project are spring core, spring batch, and *sqlite* jdbc connector:

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0"
2    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4      http://maven.apache.org/xsd/maven-4.0.0.xsd">
5    <modelVersion>4.0.0</modelVersion>
6
7    <groupId>org.baeldung</groupId>
8    <artifactId>spring-batch-intro</artifactId>
9    <version>0.1-SNAPSHOT</version>
10   <packaging>jar</packaging>
11
12   <name>spring-batch-intro</name>
13   <url>http://maven.apache.org</url>
14
15   <properties>
16     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
17     <spring.version>4.2.0.RELEASE</spring.version>
18     <spring.batch.version>3.0.5.RELEASE</spring.batch.version>
19     <sqlite.version>3.8.11.2</sqlite.version>
20   </properties>
21
22   <dependencies>
```

```

20      <!-- SQLite database driver -->
21      <dependency>
22          <groupId>org.xerial</groupId>
23          <artifactId>sqlite-jdbc</artifactId>
24          <version>${sqlite.version}</version>
25      </dependency>
26      <dependency>
27          <groupId>org.springframework</groupId>
28          <artifactId>spring-oxm</artifactId>
29          <version>${spring.version}</version>
30      </dependency>
31      <dependency>
32          <groupId>org.springframework</groupId>
33          <artifactId>spring-jdbc</artifactId>
34          <version>${spring.version}</version>
35      </dependency>
36      <dependency>
37          <groupId>org.springframework.batch</groupId>
38          <artifactId>spring-batch-core</artifactId>
39          <version>${spring.batch.version}</version>
40      </dependency>
41  </dependencies>
42 </project>
43
44
45

```

4. Spring Batch Config

First thing we'll do is to configure Spring Batch with XML:

```

1   <beans xmlns="http://www.springframework.org/schema/beans"
2       xmlns:jdbc="http://www.springframework.org/schema/jdbc"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xsi:schemaLocation="
5           http://www.springframework.org/schema/beans
6           http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
7           http://www.springframework.org/schema/jdbc
8           http://www.springframework.org/schema/jdbc/spring-jdbc-4.2.xsd
9       ">
10
11      <!-- connect to SQLite database -->
12      <bean id="dataSource"
13          class="org.springframework.jdbc.datasource.DriverManagerDataSource">
14          <property name="driverClassName" value="org.sqlite.JDBC" />
15          <property name="url" value="jdbc:sqlite:repository.sqlite" />
16          <property name="username" value="" />
17          <property name="password" value="" />

```

```

15    </bean>
16
17    <!-- create job-meta tables automatically -->
18    <jdbc:initialize-database data-source="dataSource">
19        <jdbc:script
20            location="org/springframework/batch/core/schema-drop-sqlite.sql" />
21            <jdbc:script location="org/springframework/batch/core/schema-sqlite.sql" />
22    </jdbc:initialize-database>
23
24    <!-- stored job-meta in memory -->
25    <!--
26    <bean id="jobRepository"
27        class="org.springframework.batch.core.repository.support.MapJobRepositoryFactor
28        <property name="transactionManager" ref="transactionManager" />
29    </bean>
30    -->
31
32    <!-- stored job-meta in database -->
33    <bean id="jobRepository"
34        class="org.springframework.batch.core.repository.support.JobRepositoryFactoryBe
35        <property name="dataSource" ref="dataSource" />
36        <property name="transactionManager" ref="transactionManager" />
37        <property name="databaseType" value="sqlite" />
38    </bean>
39
40    <bean id="transactionManager" class=
41        "org.springframework.batch.support.transaction.ResourcelessTransactionManager" ,
42
43    </beans>
44
45
46
47
48
49
50
51

```

Of course a Java configuration is also available:

```

1  @Configuration
2  @EnableBatchProcessing
3  public class SpringConfig {
4
5      @Value("org/springframework/batch/core/schema-drop-sqlite.sql")
6      private Resource dropReopsitoryTables;
7
8      @Value("org/springframework/batch/core/schema-sqlite.sql")
9      private Resource dataReopsitorySchema;

```

```

9
10    @Bean
11    public DataSource dataSource() {
12        DriverManagerDataSource dataSource = new DriverManagerDataSource();
13        dataSource.setDriverClassName("org.sqlite.JDBC");
14        dataSource.setUrl("jdbc:sqlite:repository.sqlite");
15        return dataSource;
16    }
17
18    @Bean
19    public DataSourceInitializer dataSourceInitializer(DataSource dataSource)
20        throws MalformedURLException {
21        ResourceDatabasePopulator databasePopulator =
22            new ResourceDatabasePopulator();
23
24        databasePopulator.addScript(dropRepositoryTables);
25        databasePopulator.addScript(dataRepositorySchema);
26        databasePopulator.setIgnoreFailedDrops(true);
27
28        DataSourceInitializer initializer = new DataSourceInitializer();
29        initializer.setDataSource(dataSource);
30        initializer.setDatabasePopulator(databasePopulator);
31
32        return initializer;
33    }
34
35    private JobRepository getJobRepository() throws Exception {
36        JobRepositoryFactoryBean factory = new JobRepositoryFactoryBean();
37        factory.setDataSource(dataSource());
38        factory.setTransactionManager(getTransactionManager());
39        factory.afterPropertiesSet();
40        return (JobRepository) factory.getObject();
41    }
42
43    private PlatformTransactionManager getTransactionManager() {
44        return new ResourcelessTransactionManager();
45    }
46
47    public JobLauncher getJobLauncher() throws Exception {
48        SimpleJobLauncher jobLauncher = new SimpleJobLauncher();
49        jobLauncher.setJobRepository(getJobRepository());
50        jobLauncher.afterPropertiesSet();
51        return jobLauncher;
52    }
53
54}

```

5. Spring Batch Job Config

Let's now write our job description for the CSV to XML work:

```
1  <beans xmlns="http://www.springframework.org/schema/beans"
2    xmlns:batch="http://www.springframework.org/schema/batch"
3    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4    xsi:schemaLocation="http://www.springframework.org/schema/batch
5      http://www.springframework.org/schema/batch/spring-batch-3.0.xsd
6      http://www.springframework.org/schema/beans
7      http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
8    ">
9
10   <import resource="spring.xml" />
11
12   <bean id="record" class="org.baeldung.spring_batch_intro.model.Transaction"></bean>
13   <bean id="itemReader"
14     class="org.springframework.batch.item.file.FlatFileItemReader">
15
16     <property name="resource" value="input/record.csv" />
17
18     <property name="lineMapper">
19       <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
20         <property name="lineTokenizer">
21           <bean class=
22             "org.springframework.batch.item.file.transform.DelimitedLineTok
23               <property name="names" value="username,userid,transactiondate,
24             </bean>
25           </property>
26           <property name="fieldSetMapper">
27             <bean class="org.baeldung.spring_batch_intro.service.RecordFieldSe
28             </bean>
29           </property>
30         </bean>
31     </property>
32   </bean>
33
34   <bean id="itemProcessor"
35     class="org.baeldung.spring_batch_intro.service.CustomItemProcessor" />
36
37   <bean id="itemWriter"
38     class="org.springframework.batch.item.xml.StaxEventItemWriter">
39     <property name="resource" value="file:xml/output.xml" />
40     <property name="marshaller" ref="recordMarshaller" />
41     <property name="rootTagName" value="transactionRecord" />
42   </bean>
43
44   <bean id="recordMarshaller" class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
45     <property name="classesToBeBound">
46       <list>
47         <value>org.baeldung.spring_batch_intro.model.Transaction</value>
48       </list>
49     </property>
50   </bean>
51
52 <batch:job id="firstBatchJob">
```

```

43      <batch:step id="step1">
44          <batch:tasklet>
45              <batch:chunk reader="itemReader" writer="itemWriter"
46                  processor="itemProcessor" commit-interval="10">
47                  </batch:chunk>
48          </batch:tasklet>
49      </batch:step>
50  </batch:job>
51
52
53
54
55
56
57
58
59

```

And of course, the similar java based job config:

```

1   public class SpringBatchConfig {
2
3       @Autowired
4       private JobBuilderFactory jobs;
5
6       @Autowired
7       private StepBuilderFactory steps;
8
9       @Value("input/record.csv")
10      private Resource inputCsv;
11
12      @Value("file:xml/output.xml")
13      private Resource outputXml;
14
15      @Bean
16      public ItemReader<Transaction> itemReader()
17          throws UnexpectedInputException, ParseException {
18          FlatFileItemReader<Transaction> reader = new FlatFileItemReader<Transaction>()
19          DelimitedLineTokenizer tokenizer = new DelimitedLineTokenizer();
20          String[] tokens = { "username", "userid", "transactiondate", "amount" };
21          tokenizer.setNames(tokens);
22          reader.setResource(inputCsv);
23          DefaultLineMapper<Transaction> lineMapper =
24              new DefaultLineMapper<Transaction>();
25          lineMapper.setLineTokenizer(tokenizer);
26          lineMapper.setFieldSetMapper(new RecordFieldSetMapper());
27          reader.setLineMapper(lineMapper);
28          return reader;
29      }
30
31      @Bean
32      public ItemProcessor<Transaction, Transaction> itemProcessor() {
33          return new CustomItemProcessor();
34      }

```

```

29 }
30
31     @Bean
32     public ItemWriter<Transaction> itemWriter(Marshaller marshall)
33         throws MalformedURLException {
34         StaxEventItemWriter<Transaction> itemWriter =
35             new StaxEventItemWriter<Transaction>();
36         itemWriter.setMarshaller(marshall);
37         itemWriter.setRootTagName("transactionRecord");
38         itemWriter.setResource(outputXml);
39         return itemWriter;
40     }
41
42     @Bean
43     public Marshaller marshall() {
44         Jaxb2Marshaller marshall = new Jaxb2Marshaller();
45         marshall.setClassesToBeBound(new Class[] { Transaction.class });
46         return marshall;
47     }
48
49     @Bean
50     protected Step step1(ItemReader<Transaction> reader,
51         ItemProcessor<Transaction, Transaction> processor,
52         ItemWriter<Transaction> writer) {
53         return steps.get("step1").<Transaction, Transaction> chunk(10)
54             .reader(reader).processor(processor).writer(writer).build();
55     }
56 }
57
58
59
60
61
62
63
64
65
66

```

OK, so now that we have the whole config, let's break it down and start discussing it.

5.1. Read Data and Create Objects with *ItemReader*

First we configured the *cvsFileItemReader* which will read the data from the *record.csv* and convert it into the *Transaction* object:

```

1
2     @SuppressWarnings("restriction")
3     @XmlRootElement(name = "transactionRecord")
4     public class Transaction {
5         private String username;
6         private int userId;
7         private Date transactionDate;
8         private double amount;
9
10        /* getters and setters for the attributes */
11
12        @Override
13        public String toString() {
14            return "Transaction [username=" + username + ", userId=" + userId
15            + ", transactionDate=" + transactionDate + ", amount=" + amount
16            + "]";
17        }
18    }

```

To do so – it uses a custom mapper:

```

1
2     public class RecordFieldSetMapper implements FieldSetMapper<Transaction> {
3
4     public Transaction mapFieldSet(FieldSet fieldSet) throws BindException {
5         SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");
6         Transaction transaction = new Transaction();
7
8         transaction.setUsername(fieldSet.readString("username"));
9         transaction.setUserId(fieldSet.readInt(1));
10        transaction.setAmount(fieldSet.readDouble(3));
11        String dateString = fieldSet.readString(2);
12        try {
13            transaction.setTransactionDate(dateFormat.parse(dateString));
14        } catch (ParseException e) {
15            e.printStackTrace();
16        }
17        return transaction;
18    }

```

5.2. Processing Data with *ItemProcessor*

We have created our own item processor, *CustomItemProcessor*. This doesn't process anything related to the transaction object – all it does is passes the original object coming from reader to the writer:

```

1     public class CustomItemProcessor implements ItemProcessor<Transaction, Transaction> {

```

```

2     public Transaction process(Transaction item) {
3         return item;
4     }
5
6

```

5.3. Writing Objects to the FS with *ItemWriter*

Finally, we are going to store this *transaction* into an xml file located at *xml/output.xml*:

```

1 <bean id="itemWriter"
2   class="org.springframework.batch.item.xml.StaxEventItemWriter">
3     <property name="resource" value="file:xml/output.xml" />
4     <property name="marshaller" ref="recordMarshaller" />
5     <property name="rootTagName" value="transactionRecord" />
6 </bean>

```

5.4. Configuring the Batch Job

So all we have to do is connect the dots with a job – using the *batch:job* syntax.

Note the *commit-interval* – that's the number of transactions to be kept in memory before committing the batch to the *itemWriter*; it will hold the transactions in memory until that point (or until the end of the input data is encountered):

```

1 <batch:job id="firstBatchJob">
2   <batch:step id="step1">
3     <batch:tasklet>
4       <batch:chunk reader="itemReader" writer="itemWriter"
5         processor="itemProcessor" commit-interval="10">
6           </batch:chunk>
7         </batch:tasklet>
8       </batch:step>
9     </batch:job>

```

5.5. Running the Batch Job

That's it – let's now set up and run everything:

```

1 public class App {
2     public static void main(String[] args) {
3         // Spring Java config
4         AnnotationConfigApplicationContext context = new AnnotationConfigApplicationCo

```

```

4     context.register(SpringConfig.class);
5     context.register(SpringBatchConfig.class);
6     context.refresh();
7
8     JobLauncher jobLauncher = (JobLauncher) context.getBean("jobLauncher");
9     Job job = (Job) context.getBean("firstBatchJob");
10    System.out.println("Starting the batch job");
11    try {
12        JobExecution execution = jobLauncher.run(job, new JobParameters());
13        System.out.println("Job Status : " + execution.getStatus());
14        System.out.println("Job completed");
15    } catch (Exception e) {
16        e.printStackTrace();
17        System.out.println("Job failed");
18    }
19
20
21

```

This is link below Spring batch in program

<https://github.com/eugenp/tutorials/tree/master/spring-batch>

Spring Batch Tutorial : Introduction-

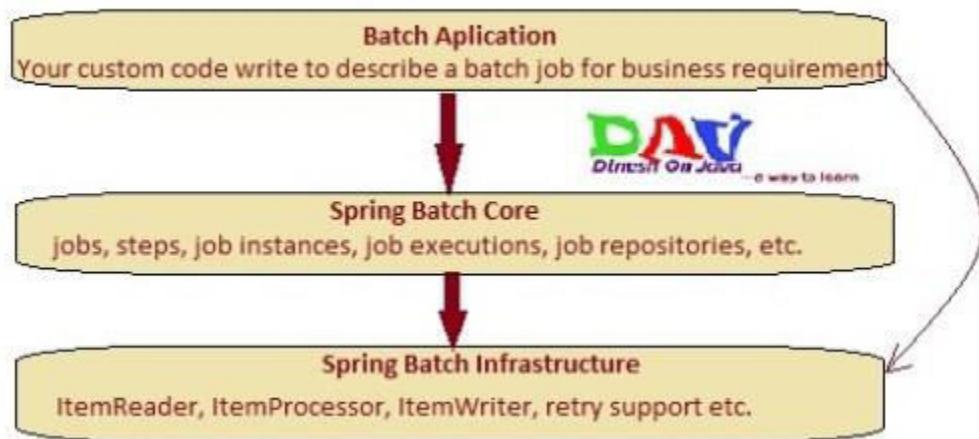
Many applications within the enterprise domain require bulk processing to perform business operations in mission critical environments. These business operations include automated, complex processing of large volumes of information that is most efficiently processed without user interaction. These operations typically include time based events (e.g. month-end calculations, notices or correspondence), periodic application of complex business rules processed repetitively across very large data sets (e.g. insurance benefit determination or rate adjustments), or the integration of information that is received from internal and external systems that typically requires formatting, validation and processing in a transactional manner into the system of record. Batch processing is used to process billions of transactions every day for enterprises.

Spring Batch-

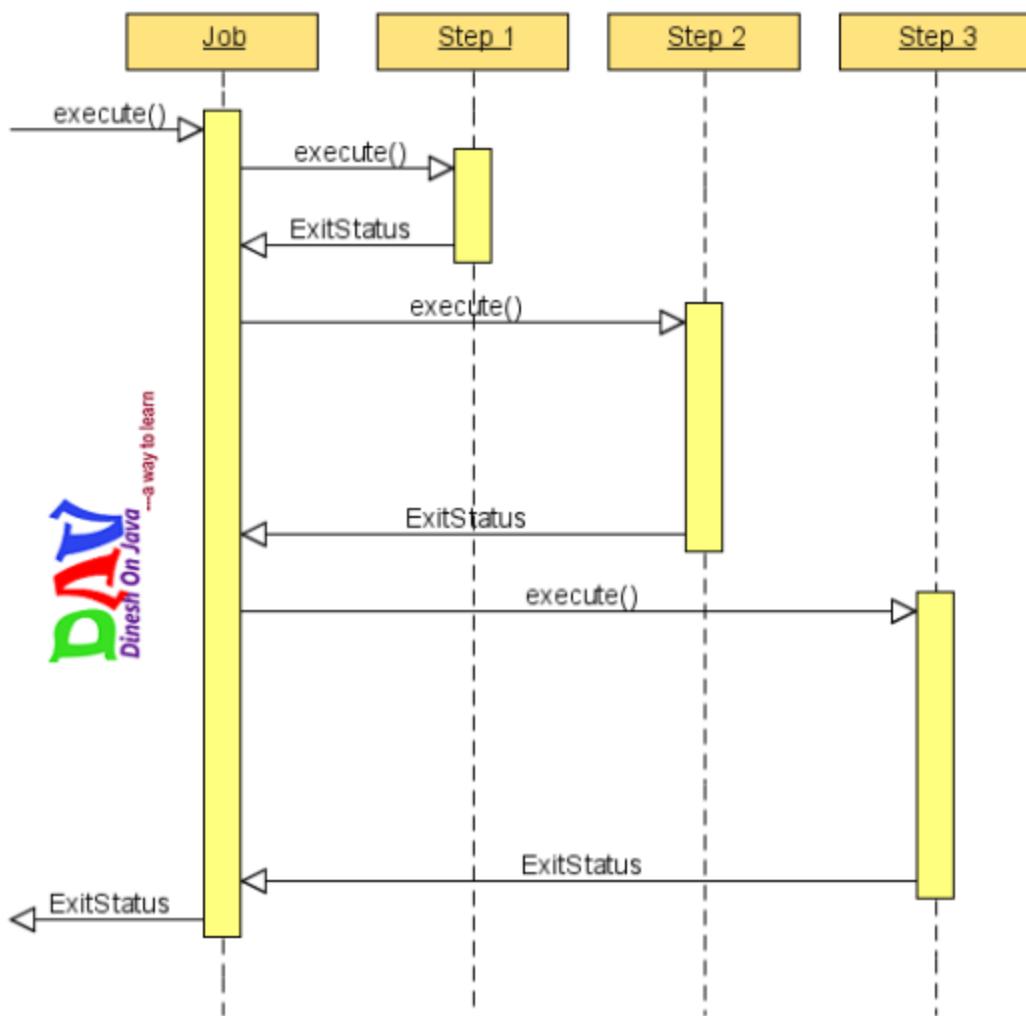
Spring Batch is a lightweight, comprehensive batch framework designed to enable the development of robust batch applications vital for the daily operations of enterprise systems. Spring Batch builds upon the productivity, POJO-based development approach, and general ease of use capabilities people have come to know from the Spring Framework, while making it easy for developers to access and leverage more advanced enterprise services when necessary.

Spring Batch provides reusable functions that are essential in processing large volumes of records, including logging/tracing, transaction management, job processing statistics, job restart, skip, and resource management. It also provides more advanced technical services and features that will enable extremely high-volume and high performance batch jobs through optimization and partitioning techniques. Simple as well as complex, high-volume batch jobs can leverage the framework in a highly scalable manner to process significant volumes of information.

High Level Architecture:-

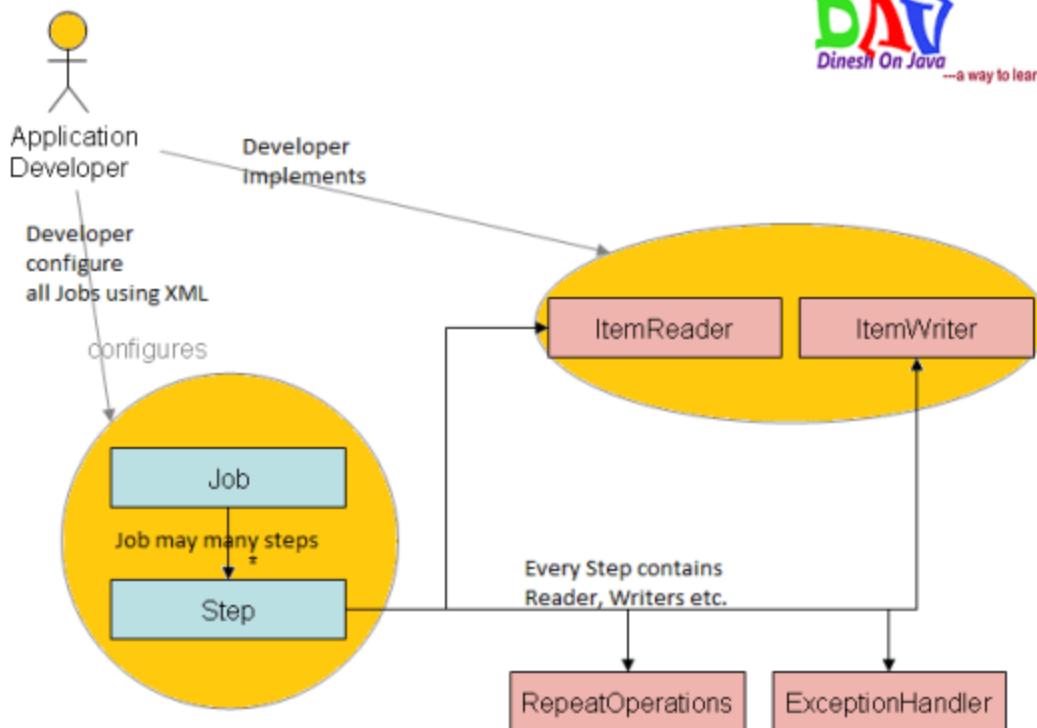


In above figure, the top of the hierarchy is the batch application itself. This is whatever batch processing application you want to write. It depends on the Spring Batch core module, which primarily provides a runtime environment for your batch jobs. Both the batch app and the core module in turn depend upon an infrastructure module that provides classes useful for both building and running batch apps.



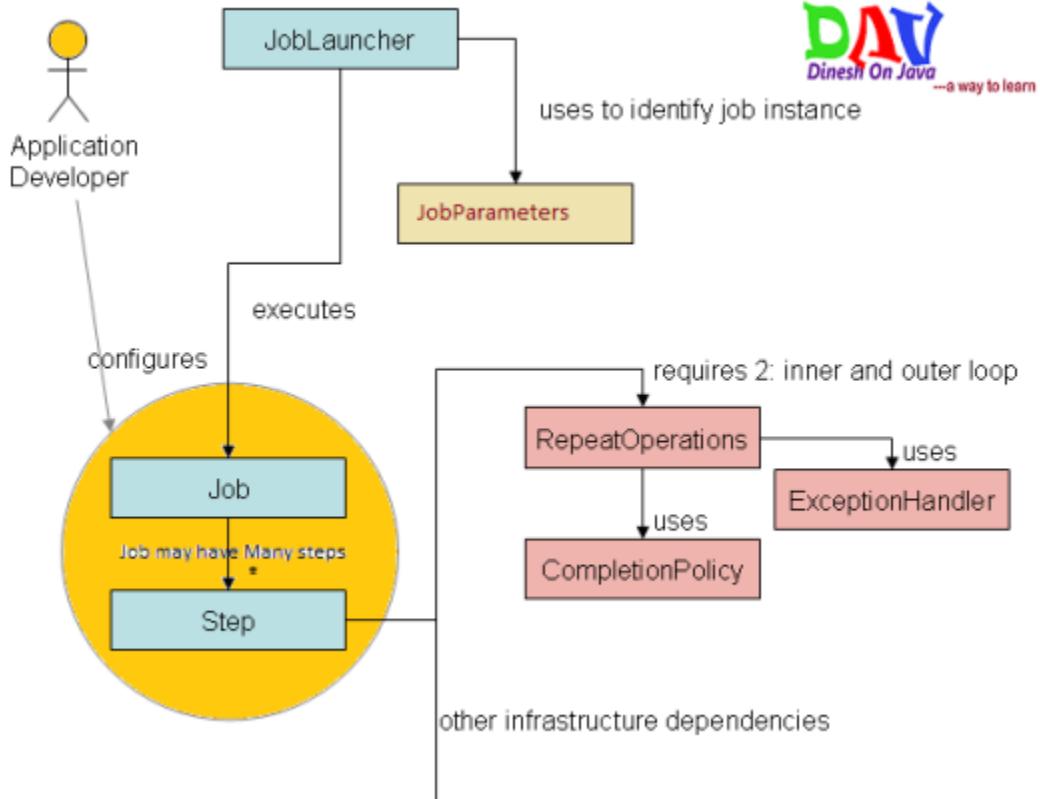
As we see in above figure a hypothetical three-step job, though obviously a job can have arbitrarily many steps. The steps are typically sequential, though as of Spring Batch 2.0 it's possible to define conditional flows (e.g., execute step 2 if step 1 succeeds; otherwise execute step 3). Within any given step, the basic process is as follows: read a bunch of "items" (e.g., database rows, XML elements, lines in a flat file—whatever), process them, and write them out somewhere to make it convenient for subsequent steps to work with the result.

Overview of the Spring Batch Core- The Spring Batch Core Domain consists of an API for launching, monitoring and managing batch jobs.



The figure above shows the central parts of the core domain and its main touch points with the batch application developer (**Job and Step**). To launch a job there is a **JobLauncher** interface that can be used to simplify the launching for dumb clients like JMX or a command line.

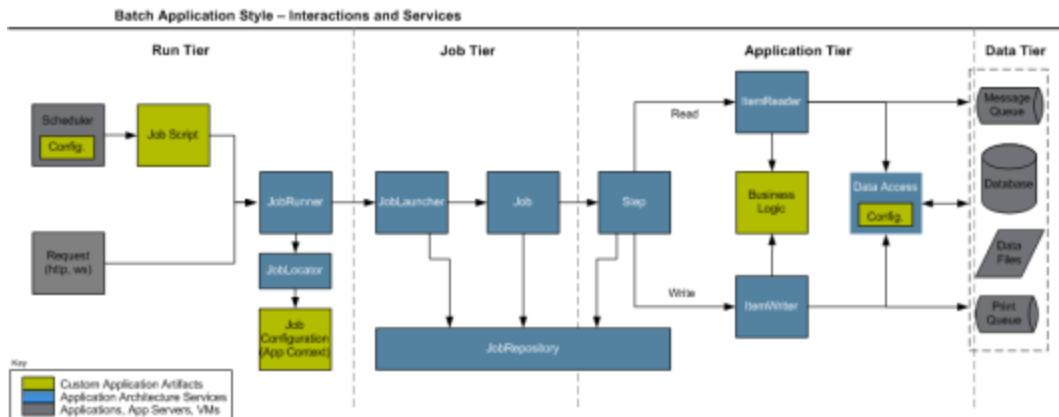
A **Job** is composed of a list of **Steps**, each of which is executed in turn by the **Job**. The **Step** is a central strategy in the **Spring Batch Core**. Implementations of **Step** are responsible for sharing the work out, but in ways that the configuration doesn't need to be aware of. For instance, the same or very similar **Step** configuration might be used in a simple in-process sequential executor, or in a multi-threaded implementation, or one that delegates to remote calls to a distributed system.



A Job can be re-used to create multiple job instances and this is reflected in the figure above showing an extended picture of the core domain. When a Job is launched it first checks to see if a job with the same **JobParameters** was already executed. We expect one of the following outcomes, depending on the Job:

- If the job was not previously launched then it can be created and executed. A new **JobInstance** is created and stored in a repository (usually a database). A new **JobExecution** is also created to track the progress of this particular execution.
- If the job was previously launched and failed the Job is responsible for indicating whether it believes it is restartable (is a restart legal and expected). There is a flag for this purpose on the Job. If the there was a previous failure – maybe the operator has fixed some bad input and wants to run it again – then we might want to restart the previous job.
- If the job was previously launched with the same **JobParameters** and completed successfully, then it is an error to restart it. An ad-hoc request needs to be distinguished from previous runs by adding a unique job parameter. In either case a new **JobExecution** is created and stored to monitor this execution of the **JobInstance**.

Spring Batch Launch Environment-



The application style is organized into four logical tiers, which include **Run**, **Job**, **Application**, and **Data** tiers. The primary goal for organizing an application according to the tiers is to embed what is known as “**separation of concerns**” within the system. Effective separation of concerns results in reducing the impact of change to the system.

- **Run Tier:** The Run Tier is concerned with the scheduling and launching of the application. A vendor product is typically used in this tier to allow time-based and interdependent scheduling of batch jobs as well as providing parallel processing capabilities.
- **Job Tier:** The Job Tier is responsible for the overall execution of a batch job. It sequentially executes batch steps, ensuring that all steps are in the correct state and all appropriate policies are enforced.
- **Application Tier:** The Application Tier contains components required to execute the program. It contains specific modules that address the required batch functionality and enforces policies around a module execution (e.g., commit intervals, capture of statistics, etc.)
- **Data Tier:** The Data Tier provides the integration with the physical data sources that might include databases, files, or queues. Note: In some cases the Job tier can be completely missing and in other cases one job script can start several batch job instances.

What's New in Spring Batch 2.X.X

- **Java 5-** The 1.x.x releases of Spring Batch were all based on Java 1.4. This prevented the framework from using many enhancements provided in Java 5 such as **generics**, **parametrized types**, etc. The entire framework has been updated to utilize these features. As a result, **Java 1.4 is no longer supported**. Most of the interfaces developers work with have been updated to support generic types. As an example, the **ItemReader** interface from 1.1 is below:

```
▪ public interface ItemReader {
```

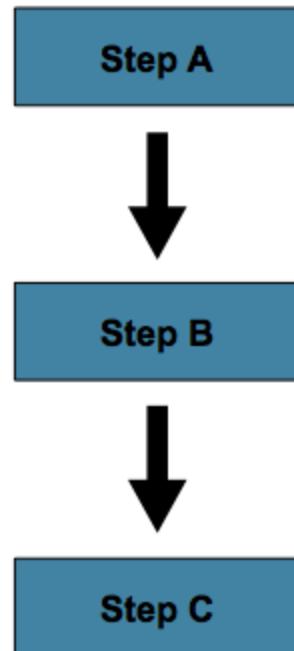
-
- Object read() throws Exception;
-
-
- void mark() throws MarkFailedException;
-
-
- void reset() throws ResetFailedException;
- }

same thing in 2.0 version given below.

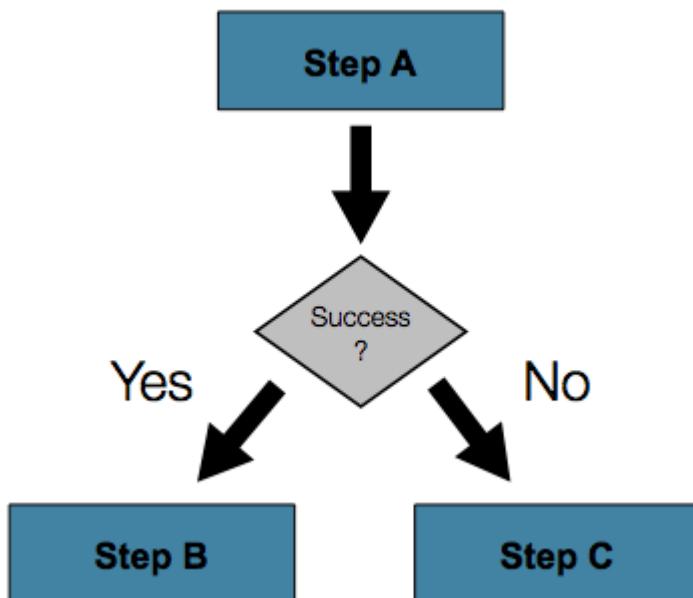
```
public interface ItemReader<T> {  
  
    T read() throws Exception, UnexpectedInputException, ParseException;  
  
}  
  
public interface ItemWriter<T> {  
  
    void write(List<? extends T> items) throws Exception;  
  
}
```

As you can see, **ItemReader** now supports the generic type, T, which is returned from read. You may also notice that mark and reset have been removed. This is due to step processing strategy changes, which are discussed below. Many other interfaces have been similarly updated.

- **Non Sequential Step Execution**-2.0 has also seen improvements in how steps can be configured. Rather than requiring that they solely be sequential:



They may now be conditional:



This new 'conditional flow' support is made easy to configure via the new namespace:

```

<job id="job">
  <step id="stepA">
    <next on="FAILED" to="stepB"></next>
  </step>
</job>

```

```

<next on="*" to="stepC"></next>

</step>

<step id="stepB" next="stepC"></step>

<step id="stepC"></step>

</job>

```

- Chunk oriented processing – In version 1.x.x – In **item-oriented processing**, the **ItemReader** returns one Object (the ‘item’) which is then handed to the **ItemWriter**, periodically committing when the number of items hits the commit interval. For example, if the commit interval is 5, **ItemReader** and **ItemWriter** will each be called 5 times. This is illustrated in a simplified code example below:

```

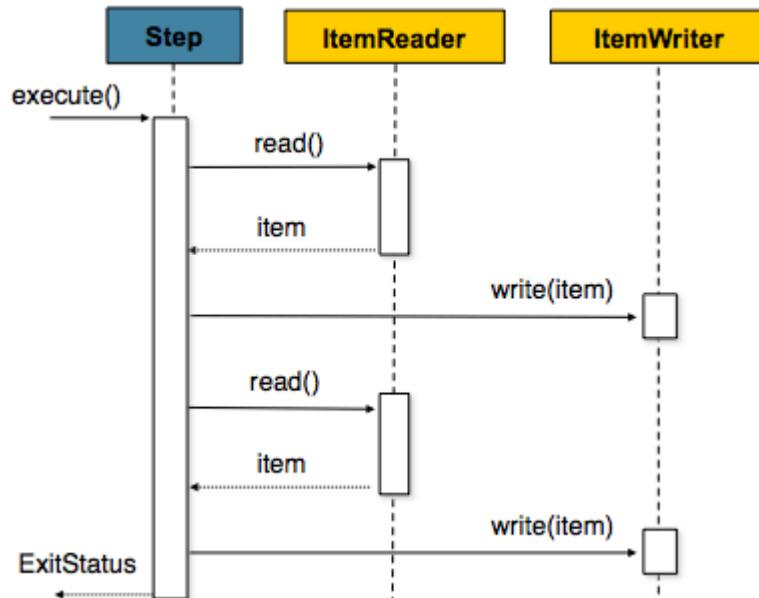
▪ for(int i = 0; i < commitInterval; i++){

▪     Object item = itemReader.read();

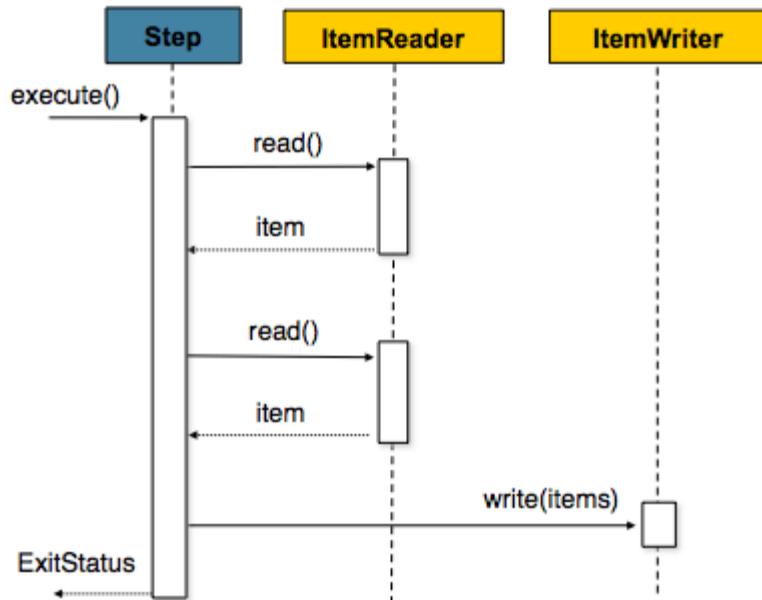
▪     itemWriter.write(item);

▪ }

```



In 2.x.x, this strategy has been changed to a chunk-oriented approach:



Using the same example from above, if the commit interval is five, read will be called 5 times, and write once. The items read will be aggregated into a list, that will ultimately be written out, as the simplified example below illustrates:

```

List items = new ArrayList();

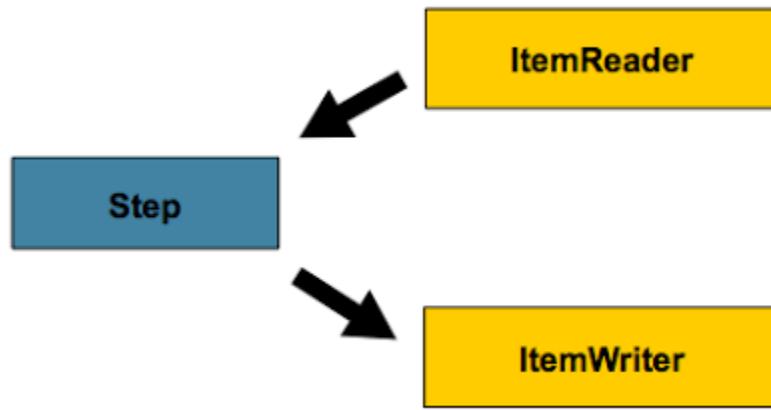
for(int i = 0; i < commitInterval; i++){

    items.add(itemReader.read());

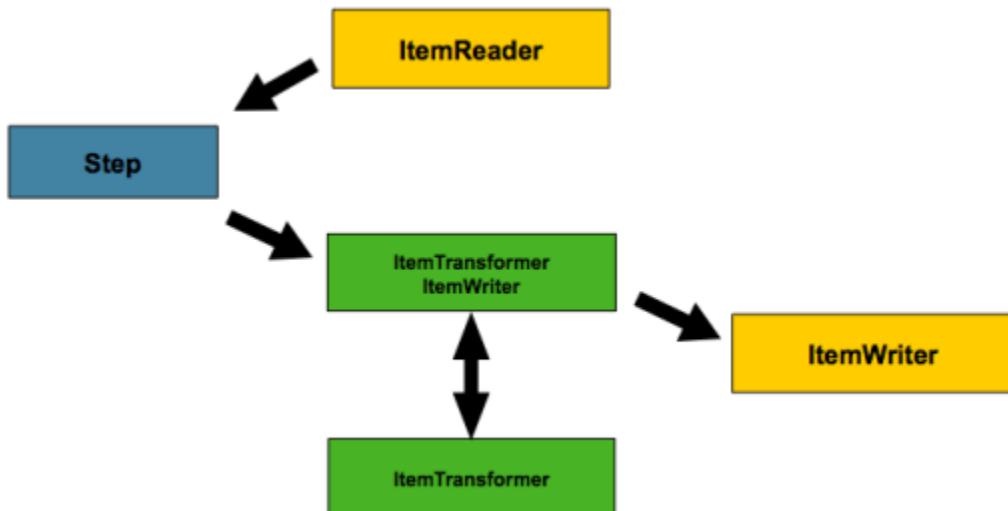
}

itemWriter.write(items);
  
```

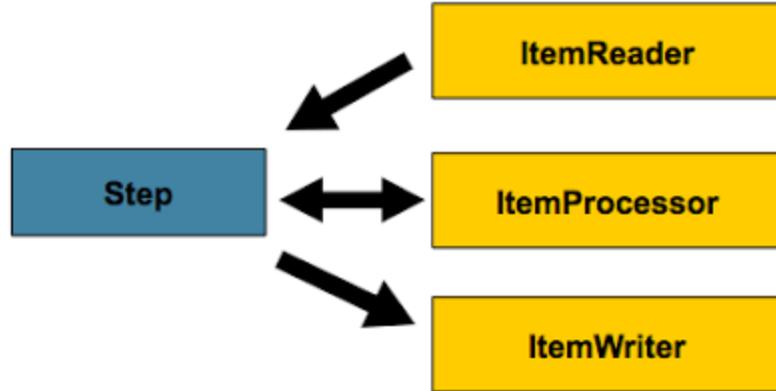
In previous version 1.x.x, Steps had only two dependencies, *ItemReader* and *ItemWriter*:



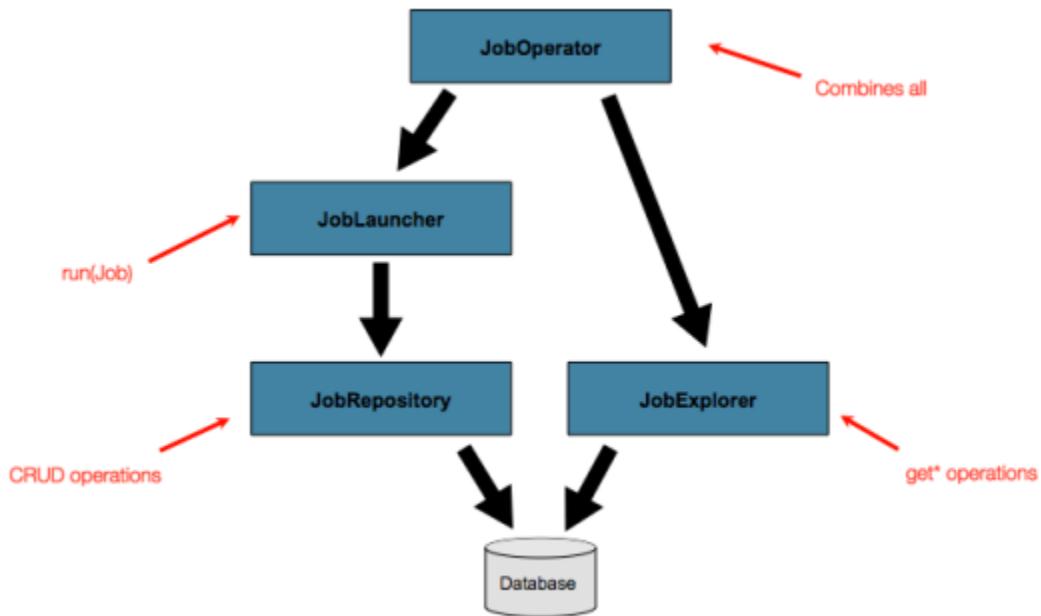
The basic configuration above is fairly robust. However, there are many cases where the item needs to be transformed before writing. In version 1.x.x this can be achieved using the composite pattern:



This approach works. However, it requires an extra layer between either the reader or the writer and the Step. Furthermore, the **ItemWriter** would need to be registered separately as an **ItemStream** with the **Step**. For this reason, **in the version 2.x.x** the **ItemTransformer** was renamed to **ItemProcessor** and moved up to the same level as **ItemReader** and **ItemWriter**:



- **Meta Data enhancements**-The *JobRepository* interface represents basic CRUD operations with Job meta-data. However, it may also be useful to query the meta-data. For that reason, the *JobExplorer* and *JobOperator* interfaces have been created:



- **Scalability**-

Spring Batch 1.x was always intended as a single VM, possibly multi-threaded model, but many features were built into it that support parallel execution in multiple processes. Many projects have successfully implemented a scalable solution relying on the quality of service features of Spring Batch to ensure that processing only happens in the correct sequence. In 2.X those features have been exposed more explicitly. There are two approaches to scalability: **remote chunking**, and **partitioning**.

Remote Chunking-

Remote chunking is a technique for dividing up the work of a step without any explicit knowledge of the structure of the data. Any input source can be split up dynamically by

reading it in a single process (as per normal in 1.x) and sending the items as a chunk to a remote worker process. The remote process implements a listener pattern, responding to the request, processing the data and sending an asynchronous reply. The transport for the request and reply has to be durable with guaranteed delivery and a single consumer, and those features are readily available with any JMS implementation. But Spring Batch is building the remote chunking feature on top of Spring Integration, therefore it is agnostic to the actual implementation of the message middleware.

Partitioning-

Partitioning is an alternative approach which in contrast depends on having some knowledge of the structure of the input data, like a range of primary keys, or the name of a file to process. The advantage of this model is that the processors of each element in a partition can act as if they are a single step in a normal Spring Batch job. They don't have to implement any special or new patterns, which makes them easy to configure and test. Partitioning in principle is more scalable than remote chunking because there is no serialization bottleneck arising from reading all the input data in one place. In Spring Batch 2.0 partitioning is supported by two interfaces: PartitionHandler and StepExecutionSplitter. The PartitionHandler is the one that knows about the execution fabric – it has to transmit requests to remote steps and collect the results using whatever grid or remoting technology is available. PartitionHandler is an SPI, and Spring Batch provides one implementation out of the box for local execution through a TaskExecutor. This will be useful immediately when parallel processing of heavily IO bound tasks is required, since in those cases remote execution only complicates the deployment and doesn't necessarily help much with the performance. Other implementations will be specific to the execution fabric. (e.g. one of the grid providers such as IBM, Oracle, Terracotta, Appistry etc.), Spring Batch makes no preference for any of grid provider over another.

- **Configuration– Until 2.X.X**, the only option for configuring batch jobs has been normal spring bean configuration. However, in **version 2.X.X** there is a new namespace for configuration. For example, in 1.1, configuring a job looked like the following:

```
▪ <bean class="org.springframework.batch.core.job.SimpleJob"  
id="myEmpExpireJob">  
  
▪   <property name="steps">  
  
▪     <list>  
  
▪       <!-- Step bean details omitted for clarity -->  
  
▪         <bean id="readEmployeeData"></bean>
```

- <bean id="writeEmployeeData"></bean>
- <bean id="employeeDataProcess"></bean>
- </list>
- </property>
- <property name="jobRepository" ref="jobRepository"></property>
- </bean>

In version 2.X.X, the equivalent would be:

```
<job id="myEmpExpireJob">

    <!-- Step bean details omitted for clarity -->

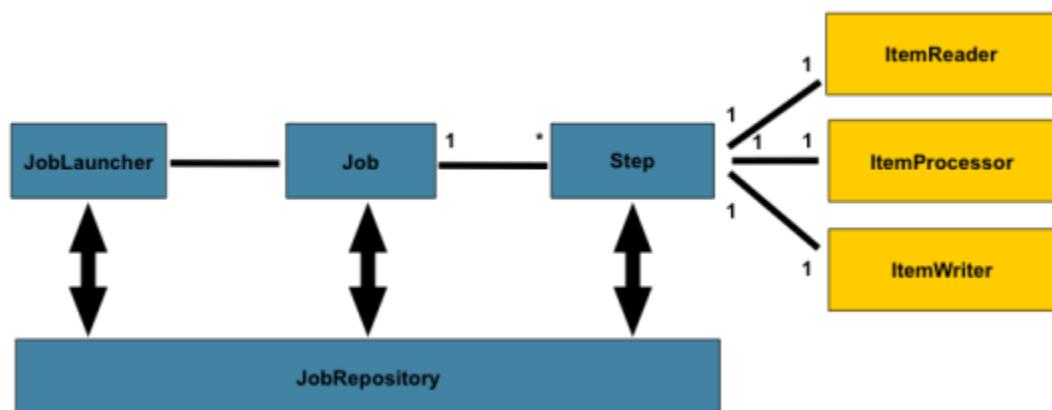
    <step id="readEmployeeData" next="writeEmployeeData"></step>

    <step id="writeEmployeeData" next="employeeDataProcess"></step>

    <step id="employeeDataProcess"></step>

</job>
```

The Domain Language of Batch

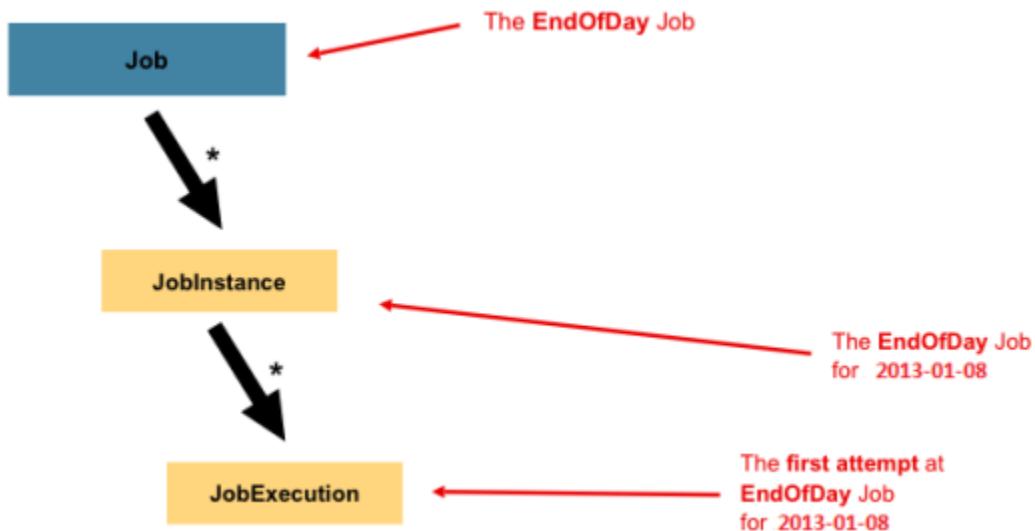


The diagram above highlights the key concepts that make up the domain language of batch. A **Job** has one to many steps, which has exactly one **ItemReader**, **ItemProcessor**, and **ItemWriter**. A

job needs to be launched (*JobLauncher*), and meta data about the currently running process needs to be stored (*JobRepository*).

Job-

A Job is an entity that encapsulates an entire batch process. As is common with other Spring projects, a Job will be wired together via an XML configuration file. This file may be referred to as the “job configuration”. However, Job is just the top of an overall hierarchy:



In Spring Batch, a Job is simply a container for Steps. It combines multiple steps that belong logically together in a flow and allows for configuration of properties global to all steps, such as restartability. The job configuration contains:

- The simple name of the job
- Definition and ordering of Steps
- Whether or not the job is restartable

A default simple implementation of the *Job* interface is provided by Spring Batch in the form of the *SimpleJob* class which creates some standard functionality on top of Job, however the batch namespace abstracts away the need to instantiate it directly. Instead, the `<job>` tag can be used:

```
<job id="myEmpExpireJob">

    <!-- Step bean details omitted for clarity -->

    <step id="readEmployeeData" next="writeEmployeeData"></step>

    <step id="writeEmployeeData" next="employeeDataProcess"></step>

    <step id="employeeDataProcess"></step>
```

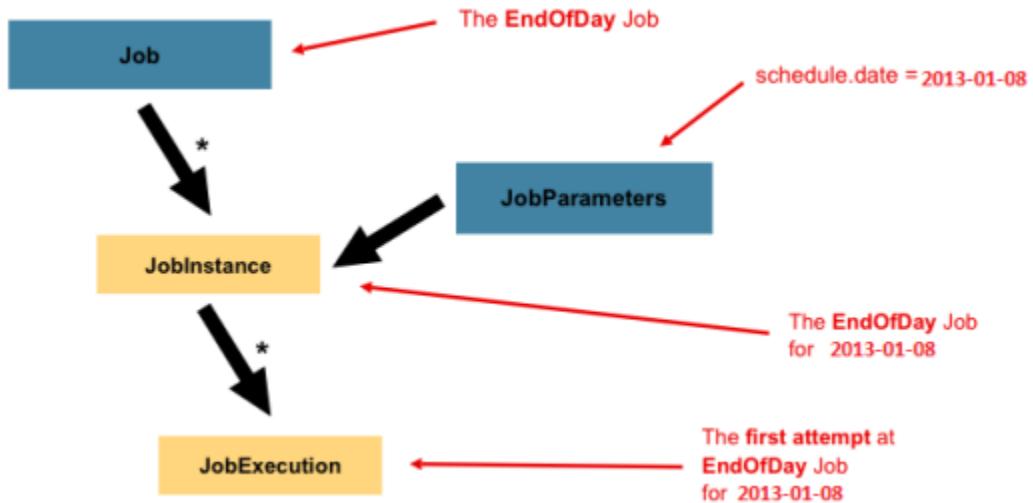
```
</job>
```

JobInstance-

A *JobInstance* refers to the concept of a logical job run. Let's consider a batch job that should be run once at the end of the day, such as the 'EndOfDay' job from the diagram above. There is one 'EndOfDay' Job, but each individual run of the Job must be tracked separately. In the case of this job, there will be one logical *JobInstance* per day. For example, there will be a January 1st run, and a January 2nd run. If the January 1st run fails the first time and is run again the next day, it is still the January 1st run.

JobParameters-

Having discussed *JobInstance* and how it differs from Job, the natural question to ask is: "how is one *JobInstance* distinguished from another?" The answer is: *JobParameters*. *JobParameters* is a set of parameters used to start a batch job. They can be used for identification or even as reference data during the run:



JobExecution-

A *JobExecution* refers to the technical concept of a single attempt to run a Job. An execution may end in failure or success, but the *JobInstance* corresponding to a given execution will not be considered complete unless the execution completes successfully. Using the EndOfDay Job described above as an example, consider a *JobInstance* for 01-01-2013 that failed the first time it was run. If it is run again with the same job parameters as the first run (01-01-2013), a new *JobExecution* will be created. However, there will still be only one *JobInstance*.

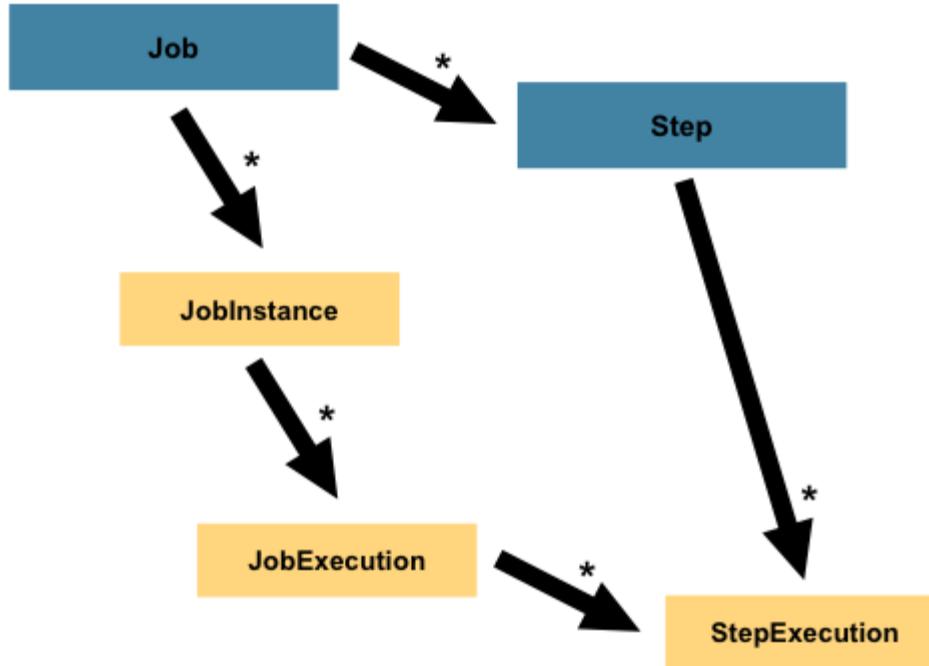
A Job defines what a job is and how it is to be executed, and *JobInstance* is a purely organizational object to group executions together, primarily to enable correct restart semantics. A *JobExecution*, however, is the primary storage mechanism for what actually

happened during a run, and as such contains many more properties that must be controlled and persisted:

status	A BatchStatus object that indicates the status of the execution. While running, it's BatchStatus.STARTED, if it fails, it's BatchStatus.FAILED, and if it finishes successfully it's BatchStatus.COMPLETED
startTime	A java.util.Date representing the current system time when the execution was started.
endTime	A java.util.Date representing the current system time when the execution finished, regardless of whether or not it was successful.
exitStatus	The ExitStatus indicating the result of the run. It is most important because it contains an exitCode that will be returned to the caller. See chapter 5 for more details.
createTime	A java.util.Date representing the current system time when the JobExecution was first persisted. The job may not have been started yet (and thus has no start time), but it will always have a createTime, which is required by the framework for managing job level ExecutionContexts.
lastUpdated	A java.util.Date representing the last time a JobExecution was persisted.
executionContext	The ‘property bag’ containing any user data that needs to be persisted between executions.
failureExceptions	The list of exceptions encountered during the execution of a Job. These can be useful if more than one exception is encountered during the failure of a Job.

Step-

A **Step** is a domain object that encapsulates an independent, sequential phase of a batch job. Therefore, every Job is composed entirely of one or more steps. A Step contains all of the information necessary to define and control the actual batch processing. This is a necessarily vague description because the contents of any given Step are at the discretion of the developer writing a Job. A Step can be as simple or complex as the developer desires. A simple Step might load data from a file into the database, requiring little or no code. (depending upon the implementations used) A more complex Step may have complicated business rules that are applied as part of the processing. As with Job, a Step has an individual **StepExecution** that corresponds with a unique **JobExecution**:



StepExecution-

A *StepExecution* represents a single attempt to execute a Step. A new *StepExecution* will be created each time a Step is run, similar to *JobExecution*. However, if a step fails to execute because the step before it fails, there will be no execution persisted for it. A *StepExecution* will only be created when its Step is actually started.

Step executions are represented by objects of the *StepExecution* class. Each execution contains a reference to its corresponding step and *JobExecution*, and transaction related data such as commit and rollback count and start and end times. Additionally, each step execution will contain an *ExecutionContext*, which contains any data a developer needs persisted across batch runs, such as statistics or state information needed to restart. The following is a listing of the properties for *StepExecution*:

StepExecution Properties

status	A BatchStatus object that indicates the status of the execution. While it's running, the status is BatchStatus.STARTED, if it fails, the status is BatchStatus.FAILED, and if it finishes successfully, the status is BatchStatus.COMPLETED
startTime	A java.util.Date representing the current system time when the execution was started.
endTime	A java.util.Date representing the current system time when the execution finished, regardless of whether or not it was successful.
exitStatus	The ExitStatus indicating the result of the execution. It is most important because it contains an exit code that will be returned to the caller. See chapter 5 for more details.
executionContext	The 'property bag' containing any user data that needs to be persisted between executions

readCount	The number of items that have been successfully read
writeCount	The number of items that have been successfully written
commitCount	The number transactions that have been committed for this execution
rollbackCount	The number of times the business transaction controlled by the Step has been rolled back.
readSkipCount	The number of times read has failed, resulting in a skipped item.
processSkipCount	The number of times process has failed, resulting in a skipped item.
filterCount	The number of items that have been ‘filtered’ by the ItemProcessor.
writeSkipCount	The number of times write has failed, resulting in a skipped item.

JobRepository-

JobRepository is the persistence mechanism for all of the Stereotypes mentioned above. It provides CRUD operations for **JobLauncher**, Job, and Step implementations. When a Job is first launched, a JobExecution is obtained from the repository, and during the course of execution **StepExecution** and **JobExecution** implementations are persisted by passing them to the repository:

```
<job-repository id="jobRepository"/>
```

JobLauncher-

JobLauncher represents a simple interface for launching a Job with a given set of JobParameters:

```
public interface JobLauncher {
    public JobExecution run(Job job, JobParameters jobParameters)
        throws JobExecutionAlreadyRunningException,
        JobRestartException;
}
```

Item Reader-

ItemReader is an abstraction that represents the retrieval of input for a Step, one item at a time. When the **ItemReader** has exhausted the items it can provide, it will indicate this by returning

null. More details about the ***ItemReader*** interface and its various implementations can be found in later Chapter , [ItemReaders and ItemWriters](#).

Item Writer-

ItemWriter is an abstraction that represents the output of a Step, one batch or chunk of items at a time. Generally, an item writer has no knowledge of the input it will receive next, only the item that was passed in its current invocation. More details about the ***ItemWriter*** interface and its various implementations can be found in later Chapter, [ItemReaders and ItemWriters](#).

Item Processor-

ItemProcessor is an abstraction that represents the business processing of an item. While the ***ItemReader*** reads one item, and the ***ItemWriter*** writes them, the ***ItemProcessor*** provides access to transform or apply other business processing. If, while processing the item, it is determined that the item is not valid, returning null indicates that the item should not be written out. More details about the ***ItemProcessor*** interface can be found in later Chapter, [ItemReaders and ItemWriters](#).

Batch Namespace-

Many of the domain concepts listed above need to be configured in a Spring ***ApplicationContext***. While there are implementations of the interfaces above that can be used in a standard bean definition, a namespace has been provided for ease of configuration:

```
<beans:beans      xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.springframework.org/schema/batch" xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-
2.0.xsd
        http://www.springframework.org/schema/batch
        http://www.springframework.org/schema/batch/spring-batch-
2.0.xsd">

<job id="ioSampleJob">
    <step id="step1">
        <tasklet>
```

```

        <chunk      commit-interval="2"      reader="itemReader"
writer="itemWriter"></chunk>

    </tasklet>

</step>

</job>

</beans:beans>

```

ItemReaders and ItemWriters in Spring Batch 2.0

ItemReader-

Although a simple concept, an **ItemReader** is the means for providing data from many different types of input. The most general examples include:

- Flat File- Flat File Item Readers read lines of data from a flat file that typically describe records with fields of data defined by fixed positions in the file or delimited by some special character (e.g. Comma).
- XML – XML **ItemReaders** process XML independently of technologies used for parsing, mapping and validating objects. Input data allows for the validation of an XML file against an XSD schema.
- Database – A database resource is accessed to return resultsets which can be mapped to objects for processing. The default SQL **ItemReaders** invoke a **RowMapper** to return objects, keep track of the current row if restart is required, store basic statistics, and provide some transaction enhancements that will be explained later.

There are many more possibilities, but we'll focus on the basic ones for this chapter. A complete list of all available **ItemReaders** can be found in Appendix A. **ItemReader** is a basic interface for generic input operations:

```

public interface ItemReader<T> {

    T read() throws Exception, UnexpectedInputException, ParseException;
}

```

```
}
```

The read method defines the most essential contract of the **ItemReader**; calling it returns one Item or null if no more items are left. An item might represent a line in a file, a row in a database, or an element in an XML file. It is generally expected that these will be mapped to a usable domain object (i.e. Trade, Foo, etc) but there is no requirement in the contract to do so. It is expected that implementations of the **ItemReader** interface will be forward only. However, if the underlying resource is transactional (such as a JMS queue) then calling read may return the same logical item on subsequent calls in a rollback scenario. It is also worth noting that a lack of items to process by an **ItemReader** will not cause an exception to be thrown. For example, a database **ItemReader** that is configured with a query that returns 0 results will simply return null on the first invocation of read.

ItemWriter-

ItemWriter is similar in functionality to an **ItemReader**, but with inverse operations. Resources still need to be located, opened and closed but they differ in that an **ItemWriter** writes out, rather than reading in. In the case of databases or queues these may be inserts, updates, or sends. The format of the serialization of the output is specific to each batch job.

As with **ItemReader**, **ItemWriter** is a fairly generic interface:

```
public interface ItemWriter<T> {  
  
    void write(List<? extends T> items) throws Exception;  
  
}
```

As with read on **ItemReader**, write provides the basic contract of **ItemWriter**; it will attempt to write out the list of items passed in as long as it is open. Because it is generally expected that items will be 'batched' together into a chunk and then output, the interface accepts a list of items, rather than an item by itself. After writing out the list, any flushing that may be necessary can be performed before returning from the write method. For example, if writing to a Hibernate DAO, multiple calls to write can be made, one for each item. The writer can then call close on the hibernate Session before returning.

ItemProcessor-

The **ItemReader** and **ItemWriter** interfaces are both very useful for their specific tasks, but what if you want to insert business logic before writing? One option for both reading and writing is to

use the composite pattern: create an **ItemWriter** that contains another **ItemWriter**, or an **ItemReader** that contains another **ItemReader**. For example:

```
public class CompositeItemWriter<T> implements ItemWriter<T> {  
  
    ItemWriter<T> itemWriter;  
  
    public CompositeItemWriter(ItemWriter<T> itemWriter) {  
  
        this.itemWriter = itemWriter;  
  
    }  
  
    public void write(List<? extends T> items) throws Exception {  
  
        //Add business logic here  
  
        itemWriter.write(item);  
  
    }  
  
    public void setDelegate(ItemWriter<T> itemWriter) {  
  
        this.itemWriter = itemWriter;  
  
    }  
}
```

The class above contains another **ItemWriter** to which it *delegates* after having provided some business logic. This pattern could easily be used for an **ItemReader** as well, perhaps to obtain more reference data based upon the input that was provided by the main **ItemReader**. It is also useful if you need to control the call to write yourself. However, if you only want to ‘transform’ the item passed in for writing before it is actually written, there isn’t much need to call write yourself: you just want to modify the item. For this scenario, Spring Batch provides the **ItemProcessor** interface:

```
public interface ItemProcessor<I, O> {  
  
    O process(I item) throws Exception;  
  
}
```

An **ItemProcessor** is very simple; given one object, transform it and return another. The provided object may or may not be of the same type. The point is that business logic may be applied within process, and is completely up to the developer to create. An **ItemProcessor** can be wired directly into a step, For example, assuming an **ItemReader** provides a class of type Foo, and it needs to be converted to type Bar before being written out. An **ItemProcessor** can be written that performs the conversion:

```
public class Foo {}  
  
public class Bar {  
  
    public Bar(Foo foo) {}  
  
}  
  
public class FooProcessor implements ItemProcessor<Foo,Bar>{  
  
    public Bar process(Foo foo) throws Exception {  
  
        //Perform simple transformation, convert a Foo to a Bar  
  
        return new Bar(foo);  
  
    }  
  
}  
  
public class BarWriter implements ItemWriter<Bar>{  
  
    public void write(List<? extends Bar> bars) throws Exception {
```

```
//write bars  
  
}  
  
}
```

In the very simple example above, there is a class Foo, a class Bar, and a class **FooProcessor** that adheres to the **ItemProcessor** interface. The transformation is simple, but any type of transformation could be done here. The **BarWriter** will be used to write out Bar objects, throwing an exception if any other type is provided. Similarly, the **FooProcessor** will throw an exception if anything but a Foo is provided. The **FooProcessor** can then be injected into a Step:

```
<job id="ioSampleJob">  
  
    <step name="step1">  
  
        <tasklet>  
  
            &lt;chunk reader="fooReader" processor="fooProcessor"  
writer="barWriter"  
  
                commit-interval="2"/&gt;  
  
        </tasklet>  
  
    </step>  
  
</job>
```

Configuring and Running a Job in Spring Batch

In this chapter we will explain the various configuration options and run time concerns of a Job. While the Job object may seem like a simple container for steps, there are many configuration options of which a developers must be aware . Furthermore, there are many considerations for how a Job will be run and how its meta-data will be stored during that run.

Configuring a Job-

There are multiple implementations of the Job interface, however, the namespace abstracts away the differences in configuration. It has only three required dependencies: a **name**, **JobRepository** , and a list of **Steps**.

```

<job id="myEmpExpireJob">

    <step          id="readEmployeeData"          next="writeEmployeeData"
parent="s1"></step>

    <step          id="writeEmployeeData"         next="employeeDataProcess"
parent="s2"></step>

    <step id="employeeDataProcess" parent="s3"></step>

</job>

```

The namespace defaults to referencing a repository with an id of '*jobRepository*', which is a sensible default. However, this can be overridden explicitly:

```

<job id="myEmpExpireJob" job-repository="specialRepository">

    <step          id="readEmployeeData"          next="writeEmployeeData"
parent="s1"></step>

    <step          id="writeEmployeeData"         next="employeeDataProcess"
parent="s2"></step>

    <step id="employeeDataProcess" parent="s3"></step>

</job>

```

1. Restartability-

One key issue when executing a batch job concerns the behavior of a Job when it is restarted. The launching of a Job is considered to be a 'restart' if a *JobExecution* already exists for the particular *JobInstance*. Ideally, all jobs should be able to start up where they left off, but there are scenarios where this is not possible. It is entirely up to the developer to ensure that a new *JobInstance* is created in this scenario. However, Spring Batch does provide some help. If a Job should never be restarted, but should always be run as part of a new *JobInstance*, then the restartable property may be set to 'false':

```
<job id="myEmpExpireJob" restartable="false">
```

```
...
```

```
</job>
```

another way...

```
Job job = new SimpleJob();

job.setRestartable(false);

JobParameters jobParameters = new JobParameters();

JobExecution firstExecution = jobRepository.createJobExecution(job,
jobParameters);

jobRepository.saveOrUpdate(firstExecution);

try {

    jobRepository.createJobExecution(job, jobParameters);

    fail();

}

catch (JobRestartException e) {

    // expected

}
```

2. Intercepting Job Execution-

During the course of the execution of a Job, it may be useful to be notified of various events in its lifecycle so that custom code may be executed. The *SimpleJob* allows for this by calling a **JobListener** at the appropriate time:

```
public interface JobExecutionListener {
```

```

    void beforeJob(JobExecution jobExecution);

    void afterJob(JobExecution jobExecution);

}

```

JobListeners can be added to a **SimpleJob** via the listeners element on the job:

```

<job id="myEmpExpireJob">

    <step          id="readEmployeeData"          next="writeEmployeeData"
parent="s1"></step>

    <step          id="writeEmployeeData"         next="employeeDataProcess"
parent="s2"></step>

    <step id="employeeDataProcess" parent="s3"></step>

    <listeners>

        <listener ref="sampleListener"></listener>

    </listeners>

</job>

```

It should be noted that **afterJob** will be called regardless of the success or failure of the Job. If success or failure needs to be determined it can be obtained from the **JobExecution**:

```

public void afterJob(JobExecution jobExecution) {

    if( jobExecution.getStatus() == BatchStatus.COMPLETED ) {

        //job success
    }
}

```

```

    }

    else if(jobExecution.getStatus() == BatchStatus.FAILED) {

        //job failure

    }

}

```

The annotations corresponding to this interface are:

- `@BeforeJob`
- `@AfterJob`

3. Inheriting from a Parent Job-

If a group of Jobs share similar, but not identical, configurations, then it may be helpful to define a “parent” Job from which the concrete Jobs may inherit properties. Similar to class inheritance in Java, the “child” Job will combine its elements and attributes with the parent’s.

In the following example, “**baseJob**” is an abstract Job definition that defines only a list of listeners. The Job “**job1**” is a concrete definition that inherits the list of listeners from “**baseJob**” and merges it with its own list of listeners to produce a Job with two listeners and one Step, “**step1**”-

```

<job abstract="true" id="baseJob">

    <listeners>

        <listener ref="listenerOne"/>

    <listeners>

    </listeners></listeners></job>




<job id="job1" parent="baseJob">

    <step id="step1" parent="standaloneStep"/>

    <listeners merge="true">

```

```
<listener ref="listenerTwo"/>

<listeners>

</listeners></listeners></job>
```

4. JobParameters Validator-

A job declared in the XML namespace or using any subclass of *AbstractJob* can optionally declare a validator for the job parameters at runtime. This is useful when for instance you need to assert that a job is started with all its mandatory parameters. There is a *DefaultJobParametersValidator* that can be used to constrain combinations of simple mandatory and optional parameters, and for more complex constraints you can implement the interface yourself. The configuration of a validator is supported through the XML namespace through a child element of the job, e.g:

```
<job id="job1" parent="baseJob3">

    <step id="step1" parent="standaloneStep"/>

    <validator ref="parametersValidator"/>

</job>
```

Configuring a JobRepository-

As described in earlier, the *JobRepository* is used for basic CRUD operations of the various persisted domain objects within Spring Batch, such as *JobExecution* and *StepExecution*. It is required by many of the major framework features, such as the *JobLauncher*, Job, and Step. The batch namespace abstracts away many of the implementation details of the *JobRepository* implementations and their collaborators. However, there are still a few configuration options available:

```
<job-repository id="jobRepository"

    data-source="dataSource"

    transaction-manager="transactionManager"

    isolation-level-for-create="SERIALIZABLE"
```

```

table-prefix="BATCH_"
max-varchar-length="1000"
/>

```

Configuring a JobLauncher-

The most basic implementation of the *JobLauncher* interface is the *SimpleJobLauncher*. Its only required dependency is a *JobRepository*, in order to obtain an execution:

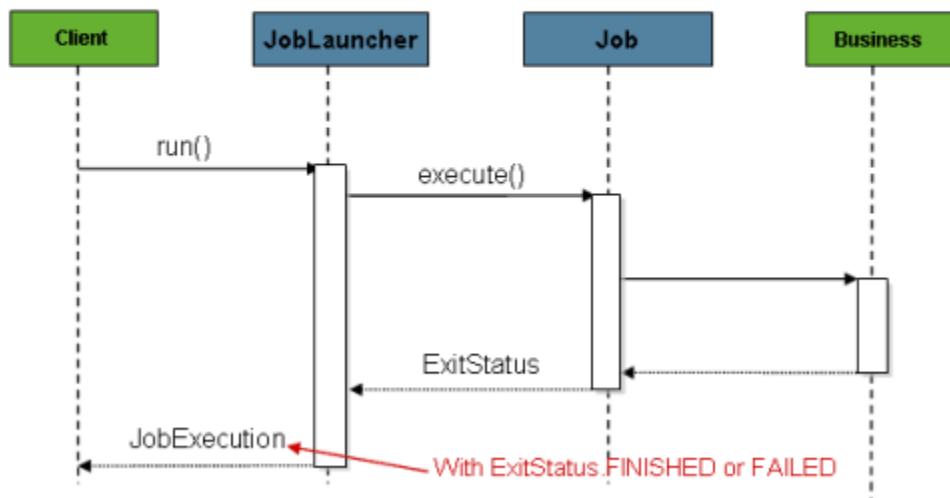
```

<bean
  class="org.springframework.batch.core.launch.support.SimpleJobLauncher"
  id="jobLauncher">

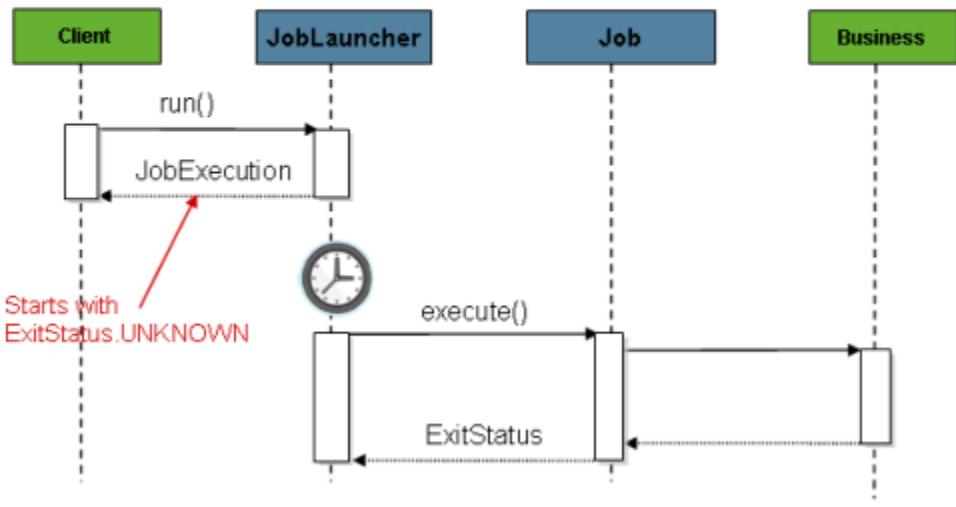
  <property name="jobRepository" ref="jobRepository" />

</bean>

```



The sequence is straightforward and works well when launched from a scheduler. However, issues arise when trying to launch from an HTTP request. In this scenario, the launching needs to be done asynchronously so that the *SimpleJobLauncher* returns immediately to its caller. This is because it is not good practice to keep an HTTP request open for the amount of time needed by long running processes such as batch. An example sequence is below:



30

The *SimpleJobLauncher* can easily be configured to allow for this scenario by configuring a *TaskExecutor*:

```

<bean
    class="org.springframework.batch.core.launch.support.SimpleJobLauncher"
    id="jobLauncher">

    <property name="jobRepository" ref="jobRepository" />

    <property name="taskExecutor">
        <bean
            class="org.springframework.core.task.SimpleAsyncTaskExecutor" />
    </property>
</bean>
    
```

Any implementation of the spring ***TaskExecutor*** interface can be used to control how jobs are asynchronously executed.

Running a Job-

At a minimum, launching a batch job requires two things: the Job to be launched and a ***JobLauncher***. Both can be contained within the same context or different contexts. For example, if launching a job from the command line, a new **JVM** will be instantiated for each Job, and thus every job will have its own ***JobLauncher***. However, if running from within a web container within the scope of an ***HttpRequest***, there will usually be one ***JobLauncher***, configured for asynchronous job launching, that multiple requests will invoke to launch their jobs.

1. Running Jobs from the Command Line-

The CommandLineJobRunner-

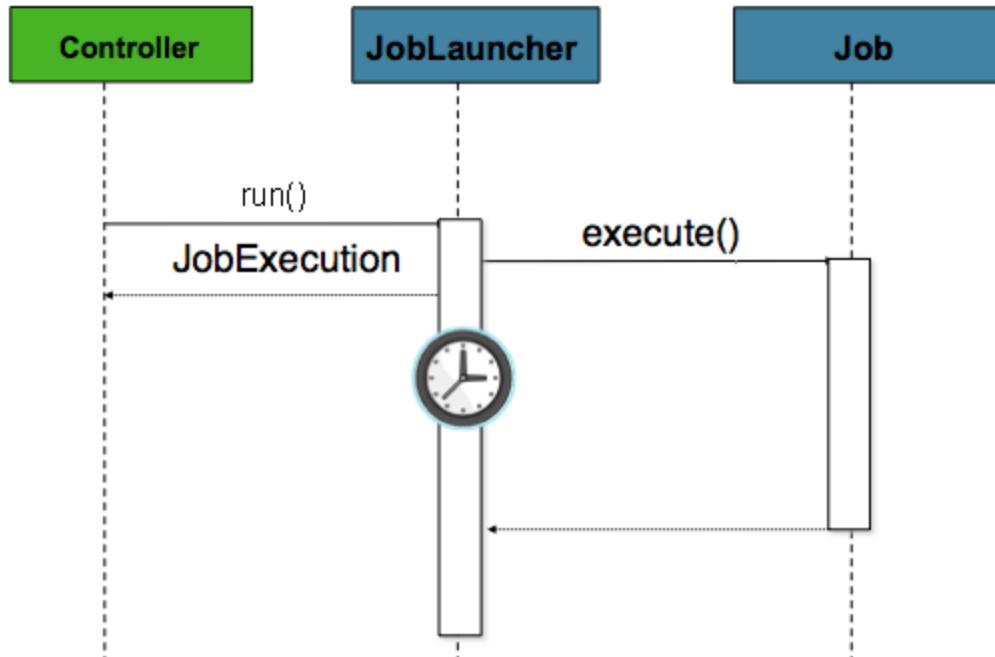
Because the script launching the job must kick off a Java Virtual Machine, there needs to be a class with a main method to act as the primary entry point. Spring Batch provides an implementation that serves just this purpose: ***CommandLineJobRunner***. It's important to note that this is just one way to bootstrap your application, but there are many ways to launch a Java process, and this class should in no way be viewed as definitive. The ***CommandLineJobRunner*** performs four tasks:

- Load the appropriate ***ApplicationContext***
- Parse command line arguments into ***JobParameters***
- Locate the appropriate job based on arguments
- Use the ***JobLauncher*** provided in the application context to launch the job.

```
bash$      java      CommandLineJobRunner      endOfDayJob.xml      endOfDay
schedule.date(date)=2013/01/08
```

2-Running Jobs from within a Web Container-

Historically, offline processing such as batch jobs have been launched from the command-line, as described above. However, there are many cases where launching from an ***HttpRequest*** is a better option. Many such use cases include reporting, ad-hoc job running, and web application support. Because a batch job by definition is long running, the most important concern is ensuring to launch the job asynchronously:



The controller in this case is a Spring MVC controller.

```

@Controller

public class JobLauncherController {

    @Autowired
    JobLauncher jobLauncher;

    @Autowired
    Job job;

    @RequestMapping("/jobLauncher.html")
    public void handle() throws Exception{
        jobLauncher.run(job, new JobParameters());
    }
}
  
```

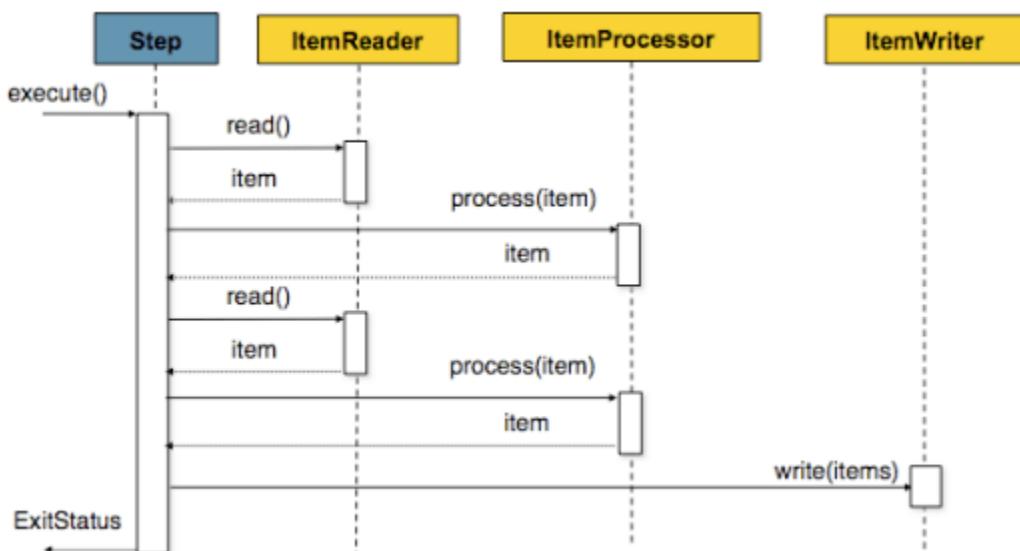
```
}
```

Configuring a Step in Spring Batch 2

A **Step** is a domain object that encapsulates an independent, sequential phase of a batch job and contains all of the information necessary to define and control the actual batch processing. This is a necessarily vague description because the contents of any given **Step** are at the discretion of the developer writing a Job. A Step can be as simple or complex as the developer desires. A simple Step might load data from a file into the database, requiring little or no code. (depending upon the implementations used) A more complex **Step** may have complicated business rules that are applied as part of the processing.

Chunk-Oriented Processing-

Spring Batch uses a 'Chunk Oriented' processing style within its most common implementation. Chunk oriented processing refers to reading the data one at a time, and creating 'chunks' that will be written out, within a transaction boundary. One item is read in from an **ItemReader**, handed to an **ItemProcessor**, and aggregated. Once the number of items read equals the commit interval, the entire chunk is written out via the **ItemWriter**, and then the transaction is committed.



Below is a code representation of the same concepts shown above:

```
List items = new ArrayList();  
  
for(int i = 0; i < commitInterval; i++) {  
  
    Object item = itemReader.read()  
}
```

```

Object processedItem = itemProcessor.process(item);

items.add(processedItem);

}

itemWriter.write(items);

```

1. Configuring a Step-

```

<job id="sampleJob" job-repository="jobRepository">

    <step id="step1">

        <tasklet transaction-manager="transactionManager">

            &lt;chunk    reader="itemReader"    writer="itemWriter"    commit-
interval="10"/&gt;

        </tasklet>

    </step>

</job>

```

The configuration above represents the only required dependencies to create a item-oriented step:

- reader – The ItemReader that provides items for processing.
- writer – The ItemWriter that processes the items provided by the ItemReader.
- transaction-manager – Spring’s PlatformTransactionManager that will be used to begin and commit transactions during processing.
- job-repository – The JobRepository that will be used to periodically store the StepExecution and ExecutionContext during processing (just before committing). For an in-line `<step/>` (one defined within a `<job/>`) it is an attribute on the `<job/>` element; for a standalone step, it is defined as an attribute of the `<tasklet/>`.
- commit-interval – The number of items that will be processed before the transaction is committed.

It should be noted that, job-repository defaults to “jobRepository” and transaction-manager defaults to “transactionManger”. Furthermore, the ItemProcessor is optional, not required, since the item could be directly passed from the reader to the writer.

2. Inheriting from a Parent Step-

If a group of Steps share similar configurations, then it may be helpful to define a “parent” Step from which the concrete Steps may inherit properties. Similar to class inheritance in Java, the “child” Step will combine its elements and attributes with the parent’s. The child will also override any of the parent’s Steps. In the following example, the Step *“concreteStep1”* will inherit from *“parentStep”*. It will be instantiated with ‘*itemReader*’, ‘*itemProcessor*’, ‘*itemWriter*’, *startLimit=5*, and *allowStartIfComplete=true*. Additionally, the *commitInterval* will be ‘5’ since it is overridden by the *“concreteStep1”*:

```
<step id="parentStep">

    <tasklet allow-start-if-complete="true">

        <chunk reader="itemReader" writer="itemWriter" commit-
interval="10"/>

    </tasklet>

<step id="concreteStep1" parent="parentStep">

    <tasklet start-limit="5">

        <chunk processor="itemProcessor" commit-interval="5"/>

    </tasklet>

</step>
```

▪ Abstract Step-

Sometimes it may be necessary to define a parent Step that is not a complete Step configuration. If, for instance, the reader, writer, and tasklet attributes are left off of a Step configuration, then initialization will fail. If a parent must be defined without these properties, then the “abstract” attribute should be used. An “abstract” Step will not be instantiated; it is used only for extending.

In the following example, the Step *“abstractParentStep”* would not instantiate if it were not declared to be abstract. The Step *“concreteStep2”* will have ‘*itemReader*’, ‘*itemWriter*’, and *commitInterval=10*.

```
<step abstract="true" id="abstractParentStep">
```

```

<tasklet>

    &lt;chunk commit-interval="10"/&gt;

</tasklet>

</step>

<step id="concreteStep2" parent="abstractParentStep">

    <tasklet>

        <chunk reader="itemReader" writer="itemWriter"/>

    </tasklet>

</step>

```

3. The Commit Interval-

As mentioned above, a step reads in and writes out items, periodically committing using the supplied *PlatformTransactionManager*. With a commit-interval of 1, it will commit after writing each individual item. This is less than ideal in many situations, since beginning and committing a transaction is expensive. Ideally, it is preferable to process as many items as possible in each transaction, which is completely dependent upon the type of data being processed and the resources with which the step is interacting. For this reason, the number of items that are processed within a commit can be configured.

```

<job id="sampleJob">

    <step id="step1">

        <tasklet>

            <chunk     reader="itemReader"     writer="itemWriter"     commit-
interval="10"/>

        </tasklet>

    </step>

```

```
</job>
```

In the example above, 10 items will be processed within each transaction. At the beginning of processing a transaction is begun, and each time `read` is called on the *ItemReader*, a counter is incremented. When it reaches 10, the list of aggregated items is passed to the *ItemWriter*, and the transaction will be committed.

4. Configuring a Step for Restart-

- *Setting a StartLimit-*

There are many scenarios where you may want to control the number of times a Step may be started. For example, a particular Step might need to be configured so that it only runs once because it invalidates some resource that must be fixed manually before it can be run again.

```
<step id="step1">

    <tasklet start-limit="1">

        <chunk      reader="itemReader"      writer="itemWriter"      commit-
interval="10"/>

    </tasklet>

</step>
```

The simple step above can be run only once. Attempting to run it again will cause an exception to be thrown. It should be noted that the default value for the `start-limit` is `Integer.MAX_VALUE`.

- *Restarting a completed step-*

```
<step id="step1">

    <tasklet allow-start-if-complete="true">

        <chunk      reader="itemReader"      writer="itemWriter"      commit-
interval="10"/>

    </tasklet>

</step>
```

In the case of a restartable job, there may be one or more steps that should always be run, regardless of whether or not they were successful the first time. An example might be a validation step, or a Step that cleans up resources before processing. During normal processing of a restarted job, any step with a status of 'COMPLETED', meaning it has already been completed successfully, will be skipped. Setting allow-start-if-complete to "true" overrides this so that the step will always run.

5. Configuring Skip Logic-

There are many scenarios where errors encountered while processing should not result in Step failure, but should be skipped instead. This is usually a decision that must be made by someone who understands the data itself and what meaning it has. Financial data, for example, may not be skippable because it results in money being transferred, which needs to be completely accurate. Loading a list of vendors, on the other hand, might allow for skips. If a vendor is not loaded because it was formatted incorrectly or was missing necessary information, then there probably won't be issues. Usually these bad records are logged as well, which will be covered later when discussing listeners.

```
<step id="step1">

    <tasklet>

        <chunk    commit-interval="10"    reader="flatFileItemReader"    skip-
limit="10" writer="itemWriter">

            <skippable-exception-classes>

                <include
class="org.springframework.batch.item.file.FlatFileParseException"/>

            </skippable-exception-classes>

        </chunk>

    </tasklet>

</step>
```

In this example, a **FlatFileItemReader** is used, and if at any point a **FlatFileParseException** is thrown, it will be skipped and counted against the total skip limit of 10. Separate counts are made of skips on read, process and write inside the step execution, and the limit applies across all. Once the skip limit is reached, the next exception found will cause the step to fail.

```

<step id="step1">

    <tasklet>

        <chunk commit-interval="10" reader="flatFileItemReader" skip-limit="10" writer="itemWriter">

            <skippable-exception-classes>

                <include class="java.lang.Exception"/>

                <exclude class="java.io.FileNotFoundException"/>

            </skippable-exception-classes>

        </chunk>

    </tasklet>

</step>

```

By 'including' `java.lang.Exception` as a skippable exception class, the configuration indicates that all Exceptions are skippable. However, by 'excluding' `java.io.FileNotFoundException`, the configuration refines the list of skippable exception classes to be all Exceptions except `FileNotFoundException`. Any excluded exception classes will be fatal if encountered (i.e. not skipped).

6. Configuring Retry Logic-

In most cases you want an exception to cause either a skip or Step failure. However, not all exceptions are deterministic. If a `FlatFileParseException` is encountered while reading, it will always be thrown for that record; resetting the `ItemReader` will not help. However, for other exceptions, such as a `DeadlockLoserDataAccessException`, which indicates that the current process has attempted to update a record that another process holds a lock on, waiting and trying again might result in success. In this case, retry should be configured:

```

<step id="step1">

    <tasklet>

        <chunk commit-interval="2" reader="itemReader" retry-limit="3"
writer="itemWriter">

```

```

<retryable-exception-classes>

    <include
class="org.springframework.dao.DeadlockLoserDataAccessException"/>

</retryable-exception-classes>

</chunk>

</tasklet>

</step>

```

7. Controlling Rollback-

By default, regardless of retry or skip, any exceptions thrown from the **ItemWriter** will cause the transaction controlled by the Step to rollback. If skip is configured as described above, exceptions thrown from the **ItemReader** will not cause a rollback. However, there are many scenarios in which exceptions thrown from the **ItemWriter** should not cause a rollback because no action has taken place to invalidate the transaction. For this reason, the Step can be configured with a list of exceptions that should not cause rollback.

```

<step id="step1">

    <tasklet>

        <chunk commit-interval="2" reader="itemReader" writer="itemWriter">

            <no-rollback-exception-classes>

                <include
class="org.springframework.batch.item.validator.ValidationException"/>

            </no-rollback-exception-classes>

        </chunk></tasklet>

    </step>

```

8. Transaction Attributes-

```

<tasklet>

    <chunk      reader="itemReader"      writer="itemWriter"      commit-
interval="2"/>

        <transaction-attributes isolation="DEFAULT"
                                     propagation="REQUIRED"
                                     timeout="30"/>

</tasklet>

```

9. Registering ItemStreams with the Step-

```

<step id="step1">

    <tasklet>

        <chunk      commit-interval="2"      reader="itemReader"
writer="compositeWriter">

            <streams>

                <stream ref="fileItemWriter1"/>

                <stream ref="fileItemWriter2"/>

            </streams>

        </chunk>

    </tasklet>

</step>

<beans:bean
class="org.springframework.batch.item.support.CompositeItemWriter"
id="compositeWriter">

```

```
<beans:property name="delegates">

    <beans:list>

        <beans:ref bean="fileItemWriter1" />

        <beans:ref bean="fileItemWriter2" />

    </beans:list>

</beans:property>

</beans:bean>
```

10. Intercepting Step Execution-

```
<step id="step1">

    <tasklet>

        <chunk reader="reader" writer="writer" commit-interval="10"/>

        <listeners>

            <listener ref="chunkListener"/>

        </listeners>

    </tasklet>

</step>
```

StepExecutionListener

```
public interface StepExecutionListener extends StepListener {

    void beforeStep(StepExecution stepExecution);
```

```
    ExitStatus afterStep(StepExecution stepExecution);
```

```
}
```

ChunkListener

```
public interface ChunkListener extends StepListener {
```

```
    void beforeChunk();
```

```
    void afterChunk();
```

```
}
```

ItemReadListener

```
public interface ItemReadListener<T> extends StepListener {
```

```
    void beforeRead();
```

```
    void afterRead(T item);
```

```
    void onReadError(Exception ex);
```

```
}
```

ItemProcessListener

```
public interface ItemProcessListener<T, S> extends StepListener {  
  
    void beforeProcess(T item);  
  
    void afterProcess(T item, S result);  
  
    void onProcessError(T item, Exception e);  
  
}
```

ItemWriteListener

```
public interface ItemWriteListener<S> extends StepListener {  
  
    void beforeWrite(List<? extends S> items);  
  
    void afterWrite(List<? extends S> items);  
  
    void onWriteError(Exception exception, List<? extends S> items);  
  
}
```

SkipListener

```

public interface SkipListener<T,S> extends StepListener {

    void onSkipInRead(Throwable t);

    void onSkipInProcess(T item, Throwable t);

    void onSkipInWrite(S item, Throwable t);

}

```

TaskletStep—The **Tasklet** is a simple interface that has one method, execute, which will be called repeatedly by the **TaskletStep** until it either returns **RepeatStatus.FINISHED** or throws an exception to signal a failure. Each call to the **Tasklet** is wrapped in a transaction. Tasklet implementors might call a stored procedure, a script, or a simple SQL update statement. To create a TaskletStep, the ‘ref’ attribute of the **<tasklet>** element should reference a bean defining a Tasklet object; no **<chunk>** element should be used within the **<tasklet>**:

```

<step id="step1">

    <tasklet ref="myTasklet"/>

</step>

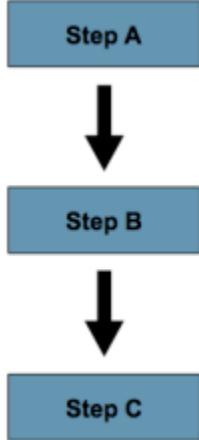
```

Controlling Step Flow-

With the ability to group steps together within an owning job comes the need to be able to control how the job ‘flows’ from one step to another. The failure of a Step doesn’t necessarily mean that the Job should fail. Furthermore, there may be more than one type of ‘success’ which determines which Step should be executed next. Depending upon how a group of Steps is configured, certain steps may not even be processed at all.

1. Sequential Flow-

The simplest flow scenario is a job where all of the steps execute sequentially:



This can be achieved using the 'next' attribute of the step element:

```

<job id="job">

    <step id="stepA" parent="s1" next="stepB" />
    <step id="stepB" parent="s2" next="stepC"/>
    <step id="stepC" parent="s3" />

</job>

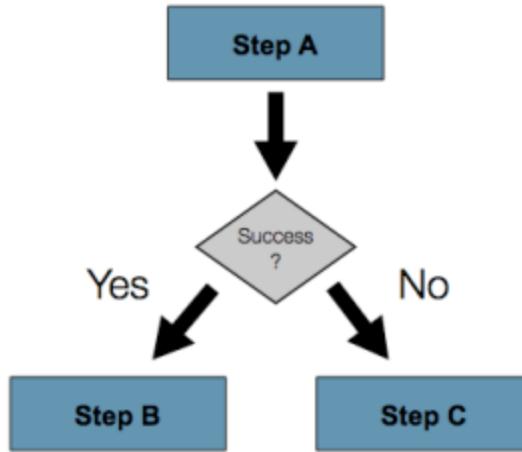
```

2. Conditional Flow-

In the example above, there are only two possibilities:

1. The Step is successful and the next Step should be executed.
2. The Step failed and thus the Job should fail.

In many cases, this may be sufficient. However, what about a scenario in which the failure of a Step should trigger a different Step, rather than causing failure?



In order to handle more complex scenarios, the Spring Batch namespace allows transition elements to be defined within the step element. One such transition is the “next” element. Like the “next” attribute, the “next” element will tell the Job which Step to execute next. However, unlike the attribute, any number of “next” elements are allowed on a given Step, and there is no default behavior the case of failure. This means that if transition elements are used, then all of the behavior for the Step’s transitions must be defined explicitly. Note also that a single step cannot have both a “next” attribute and a transition element. The next element specifies a pattern to match and the step to execute next:

```

<job id="job">

    <step id="stepA" parent="s1">

        <next on="*" to="stepB" />

        <next on="FAILED" to="stepC" />

    </step>

    <step id="stepB" parent="s2" next="stepC" />

    <step id="stepC" parent="s3" />

</job>
  
```

The “on” attribute of a transition element uses a simple pattern-matching scheme to match the ExitStatus that results from the execution of the Step. Only two special characters are allowed in the pattern:

- “*” will zero or more characters
- “?” will match exactly one character

For example, "c*t" will match "cat" and "count", while "c?t" will match "cat" but not "count". While there is no limit to the number of transition elements on a Step, if the Step's execution results in an ExitStatus that is not covered by an element, then the framework will throw an exception and the Job will fail. The framework will automatically order transitions from most specific to least specific. This means that even the elements were swapped for "stepA" in the example above, an ExitStatus of "FAILED" would still go to "stepC".

Spring Batch Glossary

Batch

An accumulation of business transactions over time.

Batch Application Style

Term used to designate batch as an application style in its own right similar to online, Web or SOA. It has standard elements of input, validation, transformation of information to business model, business processing and output. In addition, it requires monitoring at a macro level.

Batch Processing

The handling of a batch of many business transactions that have accumulated over a period of time (e.g. an hour, day, week, month, or year). It is the application of a process, or set of processes, to many data entities or objects in a repetitive and predictable fashion with either no manual element, or a separate manual element for error processing.

Batch Window

The time frame within which a batch job must complete. This can be constrained by other systems coming online, other dependent jobs needing to execute or other factors specific to the batch environment.

Step

It is the main batch task or unit of work controller. It initializes the business logic, and controls the transaction environment based on commit interval setting, etc.

Tasklet

A component created by application developer to process the business logic for a Step.

Batch Job Type

Job Types describe application of jobs for particular type of processing. Common areas are interface processing (typically flat files), forms processing (either for online pdf generation or print formats), report processing.

Job

A job represents entire batch work. E.g. End Of Day (EOD) job in bank which consists of multiple steps, each step representing single unit of work.

JobRepository

Repository is responsible for persistence of batch meta-data information. SimpleJobRepository is an implementation of JobRepository that stores JobInstances, JobExecutions, and StepExecutions information using the DAOs injected via constructor arguments. Spring Batch supports two implementations of these DAOs: Map based (in-memory) and Jdbc based. In real enterprise application the Jdbc variants are preferred but we will use simpler in-memory alternatives (MapJobInstanceDao, MapJobExecutionDao, MapStepExecutionDao, MapExecutionContextDao).

JobLauncher

As name suggests it is responsible for launching batch job. We are using SimpleJobLauncher

implementation which requires only one dependency, a JobRepository. JobRepository is used to obtain a valid JobExecution. Repository must be used because the provided Job could be a restart of an existing JobInstance, and only the Repository can reliably recreate it.

JobInstanceDao, JobExecutionDao, StepExecutionDao

These data access objects are used by SimpleJobRepository to store execution related information. Two sets of implementations are provided by Spring Batch: Map based (in-memory) and Jdbc based. In a real application the Jdbc variants are more suitable but we will use the simpler in-memory alternative in this example.

Driving Query

A driving query identifies the set of work for a job to do; the job then breaks that work into individual units of work. For instance, identify all financial transactions that have a status of "pending transmission" and send them to our partner system. The driving query returns a set of record IDs to process; each record ID then becomes a unit of work. A driving query may involve a join (if the criteria for selection falls across two or more tables) or it may work with a single table.

Item

An item represents the smallest amount of complete data for processing. In the simplest terms, this might mean a line in a file, a row in a database table, or a particular element in an XML file.

Logical Unit of Work (LUW)

A batch job iterates through a driving query (or another input source such as a file) to perform the set of work that the job must accomplish. Each iteration of work performed is a unit of work.

Commit Interval

A set of LUWs processed within a single transaction.

Partitioning

Splitting a job into multiple threads where each thread is responsible for a subset of the overall data to be processed. The threads of execution may be within the same JVM or they may span JVMs in a clustered environment that supports workload balancing.

Staging Table

A table that holds temporary data while it is being processed.

Restartable

A job that can be executed again and will assume the same identity as when run initially. In other words, it has the same job instance id.

Rerunnable

A job that is restartable and manages its own state in terms of previous run's record processing. An example of a rerunnable step is one based on a driving query. If the driving query can be formed so that it will limit the processed rows when the job is restarted than it is re-runnable. This is managed by the application logic. Often times a condition is added to the where statement to limit the rows returned by the driving query with something like "and processedFlag != true".

Repeat

One of the most basic units of batch processing, that defines repeatability calling a portion of code until it is finished, and while there is no error. Typically a batch process would be repeatable as long as there is input.

Retry

Simplifies the execution of operations with retry semantics most frequently associated with

handling transactional output exceptions. Retry is slightly different from repeat, rather than continually calling a block of code, retry is stateful, and continually calls the same block of code with the same input, until it either succeeds, or some type of retry limit has been exceeded. It is only generally useful if a subsequent invocation of the operation might succeed because something in the environment has improved.

Recover

Recover operations handle an exception in such a way that a repeat process is able to continue.

Skip

Skip is a recovery strategy often used on file input sources as the strategy for ignoring bad input records that failed validation.

Scaling and Parallel Processing in Spring Batch 2

Many batch processing problems can be solved with single threaded, single process jobs, so it is always a good idea to properly check if that meets your needs before thinking about more complex implementations. Measure the performance of a realistic job and see if the simplest implementation meets your needs first: you can read and write a file of several hundred megabytes in well under a minute, even with standard hardware.

When you are ready to start implementing a job with some parallel processing, Spring Batch offers a range of options, which are described in this chapter, although some features are covered elsewhere. At a high level there are two modes of parallel processing: single process, multi-threaded; and multi-process. These break down into categories as well, as follows:

- Multi-threaded Step (single process)
- Parallel Steps (single process)
- Remote Chunking of Step (multi process)
- Partitioning a Step (single or multi process)

Next we review the single-process options first, and then the multi-process options.

1. Multi-threaded Step-

```
<step id="loading">  
    <tasklet task-executor="taskExecutor">...</tasklet>  
</step>
```

The simplest way to start parallel processing is to add a TaskExecutor to your Step configuration.

In this example the taskExecutor is a reference to another bean definition, implementing

the **TaskExecutor** interface. **TaskExecutor** is a standard Spring interface, so consult the Spring User Guide for details of available implementations. The simplest multi-threaded **TaskExecutor** is a **SimpleAsyncTaskExecutor**.

The result of the above configuration will be that the Step executes by reading, processing and writing each chunk of items (each commit interval) in a separate thread of execution. Note that this means there is no fixed order for the items to be processed, and a chunk might contain items that are non-consecutive compared to the single-threaded case. In addition to any limits placed by the task executor (e.g. if it is backed by a thread pool), there is a throttle limit in the tasklet configuration which defaults to 4. You may need to increase this to ensure that a thread pool is fully utilised, e.g.

```
<step id="loading">    <tasklet task-executor="taskExecutor" throttle-limit="20">...</tasklet>

</step>
```

Note also that there may be limits placed on concurrency by any pooled resources used in your step, such as a **DataSource**. Be sure to make the pool in those resources at least as large as the desired number of concurrent threads in the step.

2. Parallel Steps-

As long as the application logic that needs to be parallelized can be split into distinct responsibilities, and assigned to individual steps then it can be parallelized in a single process. Parallel Step execution is easy to configure and use, for example, to execute steps (step1,step2) in parallel with step3, you could configure a flow like this:

```
<job id="job1">

<split id="split1" next="step4" task-executor="taskExecutor">

    <flow>

        <step id="step1" parent="s1" next="step2"/>

        <step id="step2" parent="s2"/>

    </flow>

    <flow>

        <step id="step3" parent="s3"/>

    </flow>

</split>
```

```

</flow>

</split>

<step id="step4" parent="s4"/>

</job>

<beans:bean id="taskExecutor" class="org.spr...SimpleAsyncTaskExecutor"/>

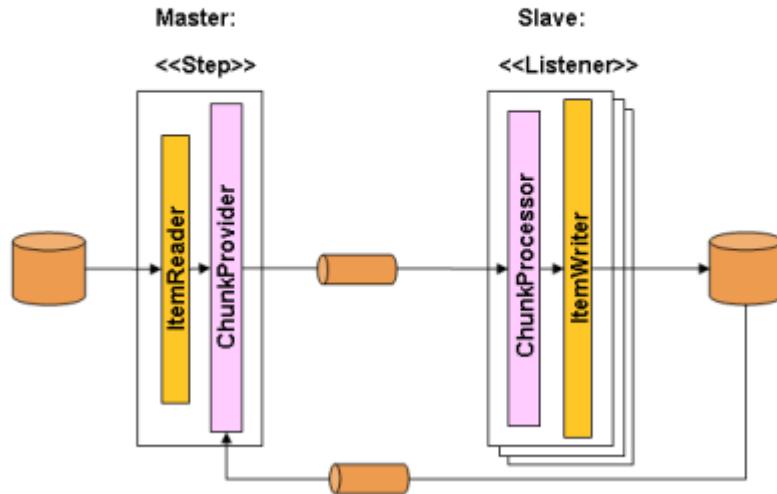
```

The configurable “task-executor” attribute is used to specify which **TaskExecutor** implementation should be used to execute the individual flows. The default is **SyncTaskExecutor**, but an asynchronous **TaskExecutor** is required to run the steps in parallel. Note that the job will ensure that every flow in the split completes before aggregating the exit statuses and transitioning.

3. Remote Chunking-

In Remote **Chunking** the Step processing is split across multiple processes, communicating with each other through some middleware. Here is a picture of the pattern in action:

Remote Chunking



The Master component is a single process, and the Slaves are multiple remote processes. Clearly this pattern works best if the Master is not a bottleneck, so the processing must be more expensive than the reading of items (this is often the case in practice).

The Master is just an implementation of a Spring Batch Step, with the ItemWriter replaced with a generic version that knows how to send chunks of items to the middleware as messages. The Slaves are standard listeners for whatever middleware is being used (e.g. with JMS they would be MessageListeners), and their role is to process the chunks of items using a standard ItemWriter or ItemProcessor plus ItemWriter, through the ChunkProcessor interface. One of the advantages of using this pattern is that the reader, processor and writer components are off-the-shelf (the same as would be used for a local execution of the step). The items are divided up dynamically and work is shared through the middleware, so if the listeners are all eager consumers, then load balancing is automatic.

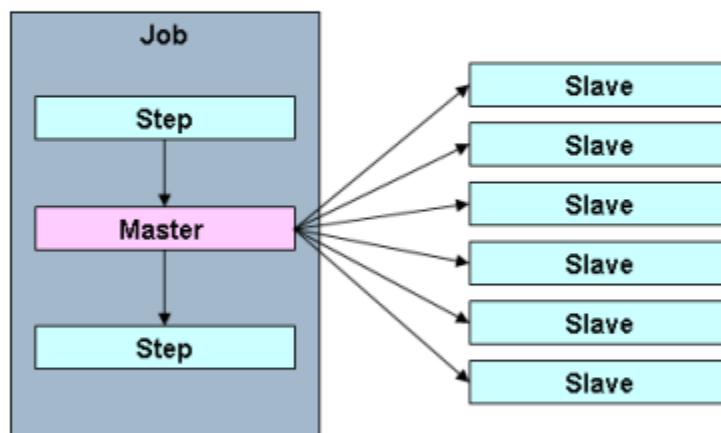
The middleware has to be durable, with guaranteed delivery and single consumer for each message. JMS is the obvious candidate, but other options exist in the grid computing and shared memory product space (e.g. Java Spaces).

Spring Batch has a sister project Spring Batch Admin, which provides(amongst other things) implementations of various patterns like this one using Spring Integration. These are implemented in a module called Spring Batch Integration.

4. Partitioning-

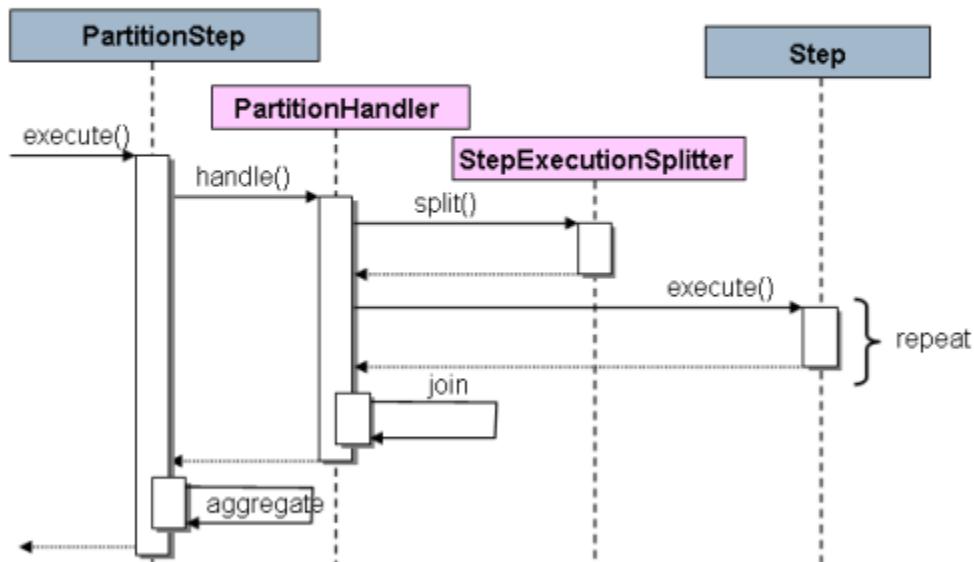
Spring Batch also provides an SPI for partitioning a Step execution and executing it remotely. In this case the remote participants are simply Step instances that could just as easily have been configured and used for local processing. Here is a picture of the pattern in action:

Partitioning Overview



The Job is executing on the left hand side as a sequence of Steps, and one of the Steps is labelled as a Master. The Slaves in this picture are all identical instances of a Step, which could in fact take the place of the Master resulting in the same outcome for the Job. The Slaves are typically going to be remote services, but could also be local threads of execution. The messages sent by the Master to the Slaves in this pattern do not need to be durable, or have guaranteed delivery: Spring Batch meta-data in the JobRepository will ensure that each Slave is executed once and only once for each Job execution.

The SPI in Spring Batch consists of a special implementation of Step (the PartitionStep), and two strategy interfaces that need to be implemented for the specific environment. The strategy interfaces are PartitionHandler and StepExecutionSplitter, and their role is show in the sequence diagram below:



The Step on the right in this case is the “remote” Slave, so potentially there are many objects and or processes playing this role, and the PartitionStep is shown driving the execution. The PartitionStep configuration looks like this:

```

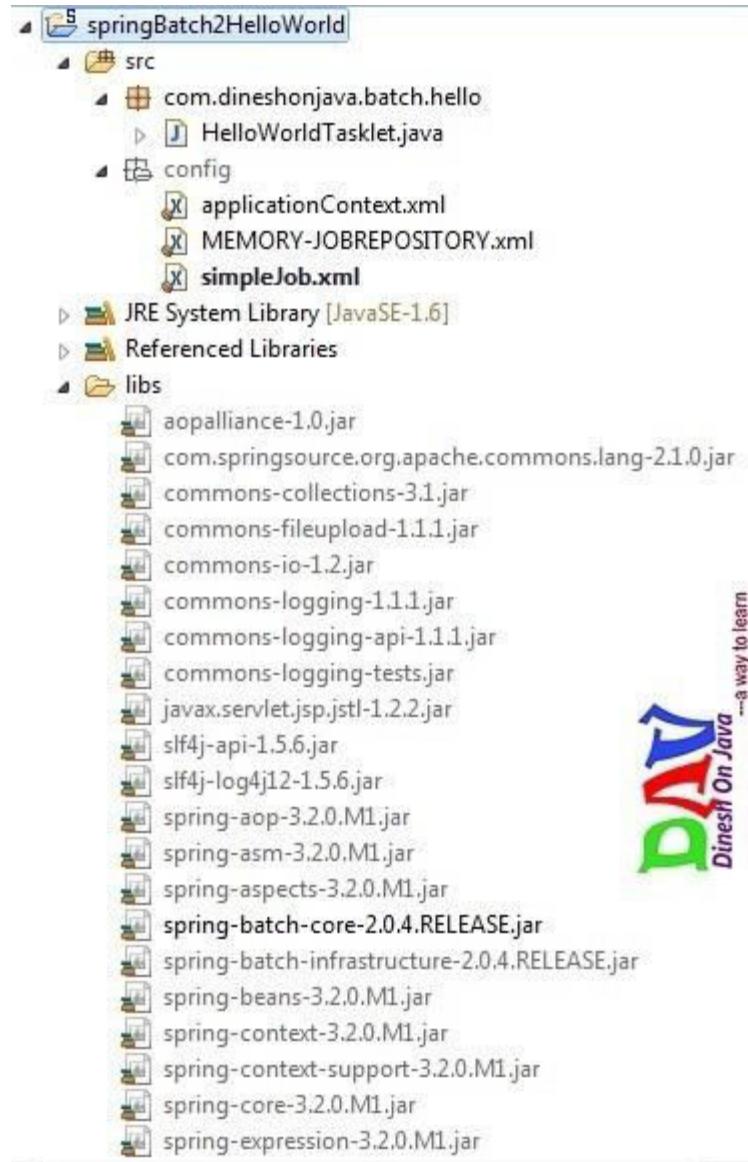
<step id="step1.master">
    <partition partitioner="partitioner" step="step1">
        <handler grid-size="10" task-executor="taskExecutor"/>
    </partition>
</step>

```

```
</partition>  
  
</step>
```

Hello World With Spring Batch 2.0.x

In this example, we will be explaining Spring Batch classes and interfaces using a simple Hello World example. We will create a job which consists of two steps. One prints “**Hello World!**” and second prints “**Created By Dinesh On Java**”. Aim is to keep the example simple and still understand the power of Spring Batch Framework.



Create Application Context XML-

In order to run Spring Batch, we need to configuration Spring Batch Infrastructure. This includes creation of ***JobRepository***, ***JobLauncher*** and ***TransactionManager***.

For each job, we will use a separate xml context definition file. However there is a number of common objects that we will need recurrently. I will group them in an ***applicationContext.xml*** which will be imported from within job definitions. Let's go through these common objects:

JobLauncher-

JobLaunchers are responsible for starting a Job with a given job parameters. The provided implementation, ***SimpleJobLauncher***, relies on a ***TaskExecutor*** to launch the jobs. If no specific ***TaskExecutor*** is set then a ***SyncTaskExecutor*** is used.

JobRepository-

We will use the ***SimpleJobRepository*** implementation which requires a set of execution ***Daos*** to store its information.

JobInstanceDao, JobExecutionDao, StepExecutionDao-

These data access objects are used by ***SimpleJobRepository*** to store execution related information. Two sets of implementations are provided by Spring Batch: Map based (in-memory) and Jdbc based. In a real application the Jdbc variants are more suitable but we will use the simpler in-memory alternative in this example.

Here's our ***applicationContext.xml***:

```
<beans xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="http://www.springframework.org/schema/beans" xsi:schemaLocation="

http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd
http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">

    <import resource="MEMORY-JOBREPOSITORY.xml"/>

</beans>
```

Here's our ***MEMORY-JOBREPOSITORY.xml***:

```
<beans xmlns:batch="http://www.springframework.org/schema/batch"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xmlns:p="http://www.springframework.org/schema/p"
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://www.springframework.org/schema/beans" xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd
http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">

<bean
class="org.springframework.batch.core.repository.support.SimpleJobRepository" id="jobRepository">

    <constructor-arg><bean
class="org.springframework.batch.core.repository.dao.MapJobInstanceDao"/>

    </constructor-arg><constructor-arg>bean
class="org.springframework.batch.core.repository.dao.MapJobExecutionDao"
/>

    </constructor-arg><constructor-arg><bean
class="org.springframework.batch.core.repository.dao.MapStepExecutionDao"/>
    </constructor-arg><constructor-arg><bean
class="org.springframework.batch.core.repository.dao.MapExecutionContextDa
o"/>

    </constructor-arg></bean>          <bean id="transactionManager"
class="org.springframework.batch.support.transaction.ResourcelessTransacti
onManager"/>          <bean
class="org.springframework.batch.core.launch.support.SimpleJobLauncher"
id="jobLauncher">      <property name="jobRepository" ref="jobRepository"/>
</bean>      </beans>
```

Create HelloWorldTasklet.java class-

A tasklet is a class containing custom logic to be run as a part of a job. **HelloWorldTasklet** is our custom tasklet which implements **Tasklet** interface and overrides the execute() method. It is a simple tasklet that simply prints a message.

Note that the execute method returns **RepeatStatus** to indicate execution status of tasklet. **RepeatStatus.FINISHED** indicates that processing is finished (either successful or unsuccessful) **RepeatStatus.CONTINUABLE** indicates that processing can continue.

```
package com.dineshonjava.batch.hello;

import org.springframework.batch.core.StepContribution;
import org.springframework.batch.core.scope.context.ChunkContext;
import org.springframework.batch.core.step.tasklet.Tasklet;
import org.springframework.batch.repeat.RepeatStatus;

/**
 *
 * @author Dinesh Rajput
 *
 */
public class HelloWorldTasklet implements Tasklet {

    private String message;

    public void setMessage(String message) {
        this.message = message;
    }
}
```

```

@Override

public RepeatStatus execute(StepContribution contribution, ChunkContext
context)

throws Exception {

System.out.println(message);

return RepeatStatus.FINISHED;

}

}

```

Create Job Configuration XML-

In this section we will see configuration to fit the custom tasklet into the Spring Batch framework.

```

<beans
    xmlns:batch="http://www.springframework.org/schema/batch"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.springframework.org/schema/beans" xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">

```

```
<import resource="applicationContext.xml"/>

<bean abstract="true"
      class="org.springframework.batch.core.step.tasklet.TaskletStep"
      id="taskletStep">

    <property name="jobRepository" ref="jobRepository"/>

    <property name="transactionManager" ref="transactionManager"/>

</bean>

<bean class="com.dineshonjava.batch.hello.HelloWorldTasklet"
      id="helloTasklet">

    <property name="message" value="Hello World!"/>

</bean>

<bean class="com.dineshonjava.batch.hello.HelloWorldTasklet"
      id="createdByTasklet">

    <property name="message" value="Created By Dinesh On Java"/>

</bean>

<bean class="org.springframework.batch.core.job.SimpleJob"
      id="mySimpleJob">

    <property name="name" value="mySimpleJob" />

    <property name="steps">
```

```

<list>

    <bean parent="taskletStep">
        <property name="tasklet" ref="helloTasklet"/>
    </bean>

    <bean parent="taskletStep">
        <property name="tasklet" ref="createdByTasklet"/>
    </bean>

</list>

</property>

<property name="jobRepository" ref="jobRepository">
</bean>

</beans>

```

First we created an abstract bean ("*taskletStep*") using class *TaskletStep*. We injected job repository and transaction manager into this tasklet. Later we created two instance of our *HelloWorldTasklet* class, one of which takes "**Hello World!**" as message and another takes "**Created By Dinesh On Java**" as message.

Finally we defined our batch job ("*mySimpleJob*") consisting of two steps using class *SimpleJob*. Property "steps" takes list of tasklet steps which will be executed sequentially one after another. As the two tasklet instances which we created earlier represent a step in our batch application, they use taskletStep abstract bean as parent.

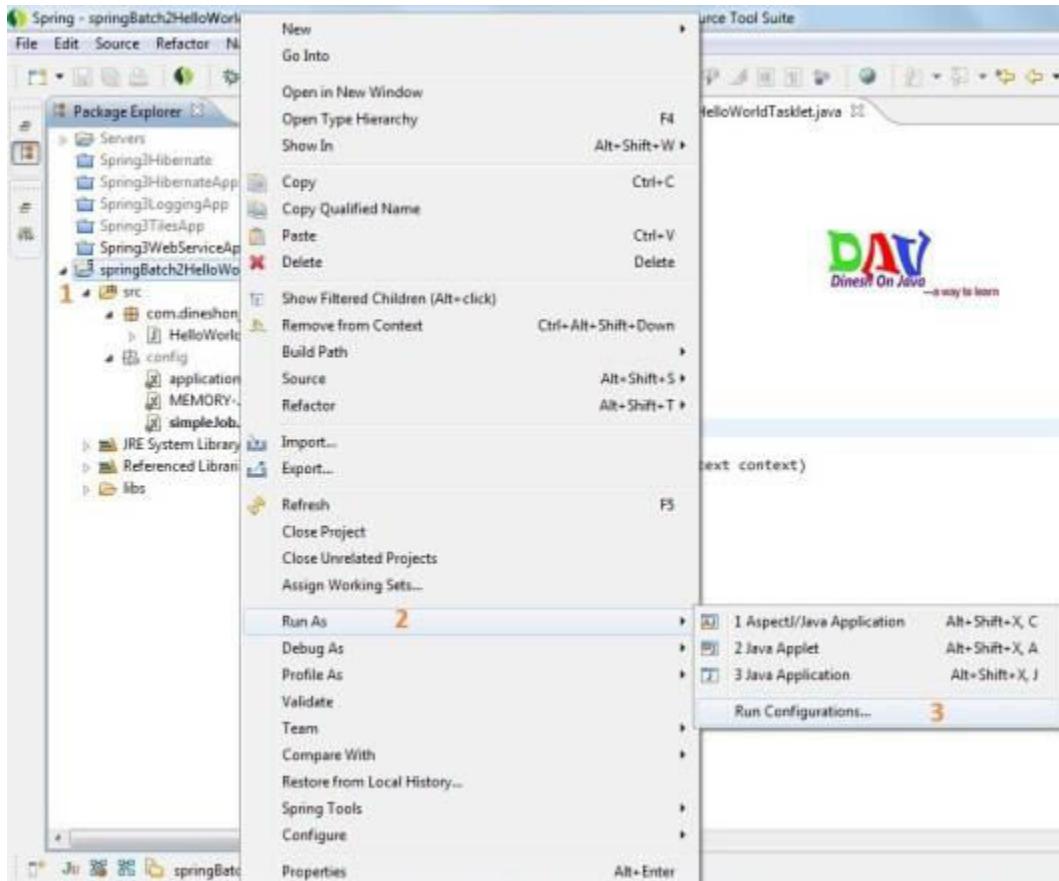
Running the Job-

Now we need something to kick-start the execution of our jobs. Spring Batch provides a convenient class to achieve that from the command line: *CommandLineJobRunner*. In its simplest form this class takes 2 arguments:

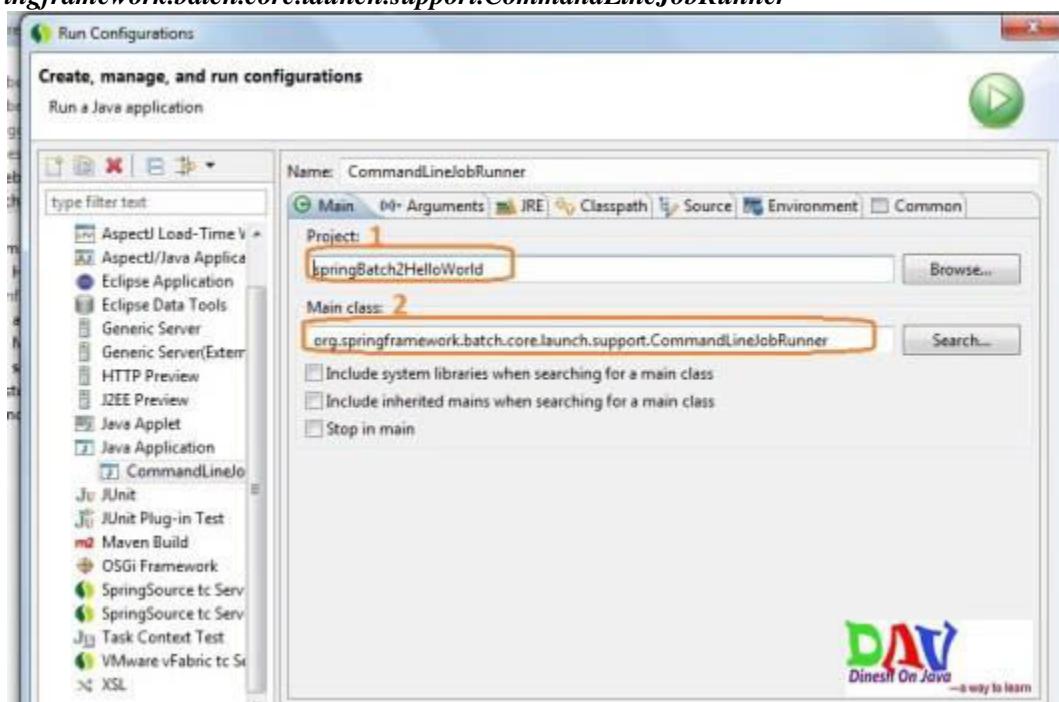
1. the xml application context containing the job to launch(*simpleJob.xml*) and
2. the bean id of that job(*mySimpleJob*).

It naturally requires a *JobLauncher* to be configured in the application context.

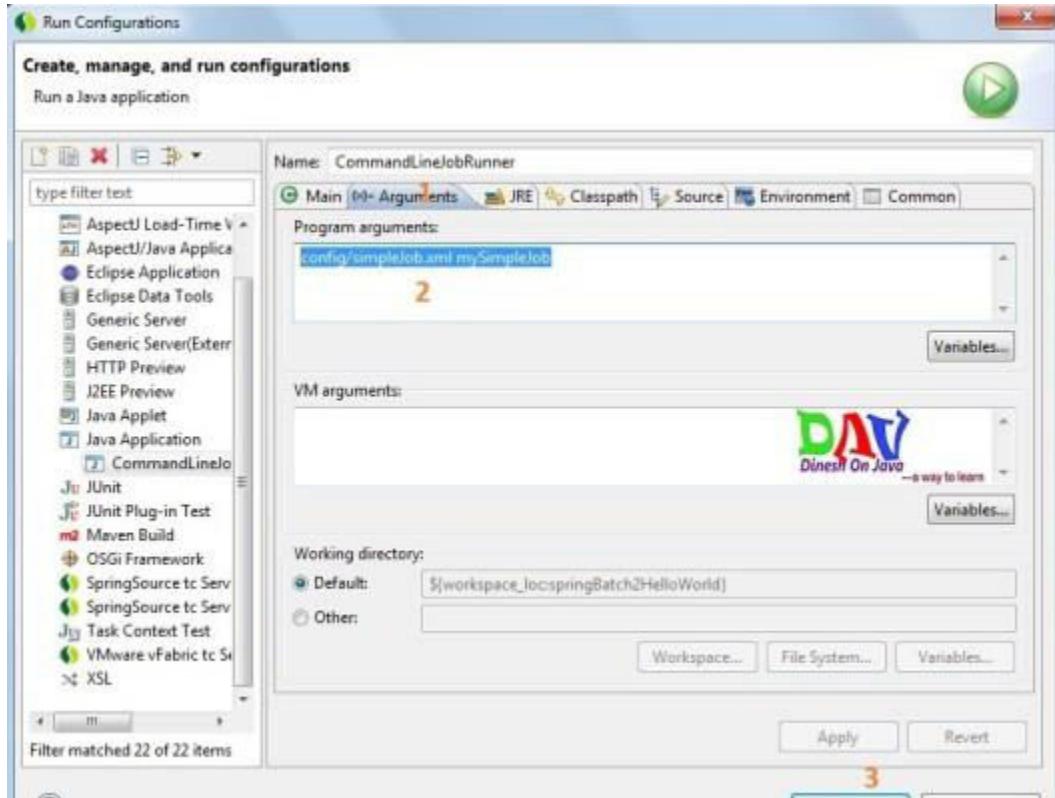
First right click to project "*springBatch2HelloWorld*" and click on **Run As** and go to **Run Configurations...** as follows...



After clicking on the ***Run Configurations...*** we get the following window and put the information as on ***Project*** field with value "*springBatch2HelloWorld*" and ***Main Class*** field with value "*org.springframework.batch.core.launch.support.CommandLineJobRunner*"



Now switch to Arguments tabs on the above window we get the following window and put the value on the **Program arguments** as “`config/simpleJob.xml mySimpleJob`”. and click on the **Run** button.



After clicking Run button if every thing is Ok then we will get the following output on the console.

OUTPUT:

```
Jan 15, 2013 12:00:27 AM org.springframework.context.support.AbstractApplicationContext
prepareRefresh
INFO:                                                 Refreshing
org.springframework.context.support.ClassPathXmlApplicationContext@1100d7a: startup date
[Tue Jan 15 00:00:27 IST 2013]; root of context hierarchy
Jan 15, 2013 12:00:27 AM org.springframework.beans.factory.xml.XmlBeanDefinitionReader
loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [config/simpleJob.xml]
Jan 15, 2013 12:00:27 AM org.springframework.beans.factory.xml.XmlBeanDefinitionReader
loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [config/applicationContext.xml]
Jan 15, 2013 12:00:27 AM org.springframework.beans.factory.xml.XmlBeanDefinitionReader
loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [config/MEMORY-
JOBREPOSITORY.xml]
```

```
Jan           15,          2013          12:00:27      AM
org.springframework.beans.factory.support.DefaultListableBeanFactory
preInstantiateSingletons
INFO:           Pre-instantiating           singletons       in
org.springframework.beans.factory.support.DefaultListableBeanFactory@1d85f79:   defining
beans
[jobRepository,transactionManager,jobLauncher,taskletStep,helloTasklet,createdByTasklet,mySi
mpleJob];           root           of           factory       hierarchy
Jan 15, 2013 12:00:27 AM org.springframework.batch.core.launch.support.SimpleJobLauncher
afterPropertiesSet
INFO: No TaskExecutor has been set, defaulting to synchronous executor.
Jan           15,          2013          12:00:27      AM
org.springframework.batch.core.launch.support.SimpleJobLauncher$1           run
INFO: Job: [SimpleJob: [name=mySimpleJob]] launched with the following parameters: []
Jan 15, 2013 12:00:27 AM org.springframework.batch.core.job.AbstractJob handleStep
INFO: Executing step: [TaskletStep: [name=taskletStep$child#1ea5671]]
Hello                               World!
Jan 15, 2013 12:00:27 AM org.springframework.batch.core.job.AbstractJob handleStep
INFO: Executing step: [TaskletStep: [name=taskletStep$child#1d15445]]
Created           By           Dinesh           On           Java
Jan           15,          2013          12:00:27      AM
org.springframework.batch.core.launch.support.SimpleJobLauncher$1           run
INFO: Job: [SimpleJob: [name=mySimpleJob]] completed with the following parameters: [] and
the           following           status:           [COMPLETED]
Jan 15, 2013 12:00:27 AM org.springframework.context.support.AbstractApplicationContext
doClose
INFO:                           Closing
org.springframework.context.support.ClassPathXmlApplicationContext@1100d7a: startup date
[Tue Jan 15 00:00:27 IST 2013];   root   of   context   hierarchy
Jan           15,          2013          12:00:27      AM
org.springframework.beans.factory.support.DefaultSingletonBeanRegistry destroySingletons
INFO:           Destroying           singletons       in
org.springframework.beans.factory.support.DefaultListableBeanFactory@1d85f79:   defining
beans
[jobRepository,transactionManager,jobLauncher,taskletStep,helloTasklet,createdByTasklet,mySi
mpleJob];   root of factory hierarchy
```


Part IV. Spring Boot features

23. Spring Application

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);  
}
```

When your application starts, you should see something similar to the following output:

By default, **INFO** logging messages are shown, including some relevant startup details, such as the user that launched the application. If you need a

log level other than `INFO`, you can set it, as described in [Section 26.4, “Log Levels”](#),

23.1 Startup Failure

If your application fails to start, registered `FailureAnalyzers` get a chance to provide a dedicated error message and a concrete action to fix the problem. For instance, if you start a web application on port `8080` and that port is already in use, you should see something similar to the following message:

```
*****
APPLICATION FAILED TO START
*****
```

Description:

Embedded servlet container failed to start. Port 8080 was already in use.

Action:

Identify and stop the process that's listening on port 8080 or configure this application to listen on another port.



Spring Boot provides numerous `FailureAnalyzer` implementations, and you can [add your own](#).

If no failure analyzers are able to handle the exception, you can still display the full conditions report to better understand what went wrong. To do so, you need to [enable](#)

[the `debug` property](#) or [enable DEBUG logging](#) for `org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingListener`.

For instance, if you are running your application by using `java -jar`, you can enable the `debug` property as follows:

```
$ java -jar myproject-0.0.1-SNAPSHOT.jar --debug
```

23.2 Customizing the Banner

The banner that is printed on start up can be changed by adding a `banner.txt` file to your classpath or by setting

the `spring.banner.location` property to the location of such a file. If the file has an encoding other than UTF-8, you can set `spring.banner.charset`. In addition to a text file, you can also add a `banner.gif`, `banner.jpg`, or `banner.png` image file to your classpath or set the `spring.banner.image.location` property. Images are converted into an ASCII art representation and printed above any text banner.

Inside your `banner.txt` file, you can use any of the following placeholders:

Table 23.1. Banner variables

	Description
<code>ion}</code>	The version number of your application, as declared in <code>MANIFEST.MF</code> . For example, <code>Implementation-Version: 1.0</code> is printed as <code>1.0</code> .
<code>atted-version}</code>	The version number of your application, as declared in <code>MANIFEST.MF</code> and formatted for display (surrounded with brackets and prefixed with <code>v</code>). For example <code>(v1.0)</code> .
<code>ion}</code>	The Spring Boot version that you are using. For example <code>2.1.4.RELEASE</code> .
<code>atted-version}</code>	The Spring Boot version that you are using, for example <code>(v2.1.4.RELEASE)</code> .
<code>AnsiColor.NAME}, \${AnsiBackground.NAME}, \${AnsiStyle.NAME})</code>	Where <code>NAME</code> is the name of an ANSI escape code. See AnsiPropertySource for details.
<code>e}</code>	The title of your application, as declared in <code>Implementation-Title: MyApp</code> .



The `SpringApplication.setBanner(...)` method can be used if you want to generate a banner programmatically. Use the `org.springframework.boot.Banner` interface and implement your own `printBanner()` method.

You can also use the `spring.main.banner-mode` property to determine if the banner has to be printed on `System.out` (`console`), sent to the configured logger (`log`), or not produced at all (`off`).

The printed banner is registered as a singleton bean under the following name: `springBootBanner`.



YAML maps `off` to `false`, so be sure to add quotes if you want to disable the banner in your application, as shown in the following example:

```
spring:  
  main:  
    banner-mode: "off"
```

23.3 Customizing SpringApplication

If the `SpringApplication` defaults are not to your taste, you can instead create a local instance and customize it. For example, to turn off the banner, you could write:

```
public static void main(String[] args) {  
    SpringApplication app = new  
    SpringApplication(MySpringConfiguration.class);  
    app.setBannerMode(Banner.Mode.OFF);  
    app.run(args);  
}
```



The constructor arguments passed to `SpringApplication` are configuration sources for Spring beans. In most cases, these are references to `@Configuration` classes, but they could also be references to XML configuration or to packages that should be scanned.

It is also possible to configure the `SpringApplication` by using an `application.properties` file. See [Chapter 24, Externalized Configuration](#) for details.

For a complete list of the configuration options, see the [SpringApplication Javadoc](#).

23.4 Fluent Builder API

If you need to build an `ApplicationContext` hierarchy (multiple contexts with a parent/child relationship) or if you prefer using a “fluent” builder API, you can use the `SpringApplicationBuilder`.

The `SpringApplicationBuilder` lets you chain together multiple method calls and includes `parent` and `child` methods that let you create a hierarchy, as shown in the following example:

```
new SpringApplicationBuilder()
    .sources(Parent.class)
    .child(Application.class)
    .bannerMode(Banner.Mode.OFF)
    .run(args);
```



There are some restrictions when creating an `ApplicationContext` hierarchy. For example, Web components **must** be contained within the child context, and the same `Environment` is used for both parent and child contexts. See the [SpringApplicationBuilder Javadoc](#) for full details.

23.5 Application Events and Listeners

In addition to the usual Spring Framework events, such as `ContextRefreshedEvent`, a `SpringApplication` sends some additional application events.



Some events are actually triggered before the `ApplicationContext` is created, so you cannot register a listener on those as a `@Bean`. You can register them with the `SpringApplication.addListeners(...)` method or the `SpringApplicationBuilder.listeners(...)` method.

If you want those listeners to be registered automatically, regardless of the way the application is created, you can add a `META-INF/spring.factories` file to your project and reference your listener(s) by using the `org.springframework.context.ApplicationListener` key, as shown in the following example:

```
org.springframework.context.ApplicationListener=com.example.project.MyListener
```

Application events are sent in the following order, as your application runs:

1. An `ApplicationStartingEvent` is sent at the start of a run but before any processing, except for the registration of listeners and initializers.
2. An `ApplicationEnvironmentPreparedEvent` is sent when the `Environment` to be used in the context is known but before the context is created.
3. An `ApplicationPreparedEvent` is sent just before the refresh is started but after bean definitions have been loaded.
4. An `ApplicationStartedEvent` is sent after the context has been refreshed but before any application and command-line runners have been called.
5. An `ApplicationReadyEvent` is sent after any application and command-line runners have been called. It indicates that the application is ready to service requests.
6. An `ApplicationFailedEvent` is sent if there is an exception on startup.



You often need not use application events, but it can be handy to know that they exist. Internally, Spring Boot uses events to handle a variety of tasks.

Application events are sent by using Spring Framework's event publishing mechanism. Part of this mechanism ensures that an event published to the listeners in a child context is also published to the listeners in any ancestor contexts. As a result of this, if your application uses a hierarchy of `SpringApplication` instances, a listener may receive multiple instances of the same type of application event.

To allow your listener to distinguish between an event for its context and an event for a descendant context, it should request that its application context is injected and then compare the injected context with the context of the event. The context can be injected by implementing `ApplicationContextAware` or, if the listener is a bean, by using `@Autowired`.

23.6 Web Environment

A `SpringApplication` attempts to create the right type of `ApplicationContext` on your behalf. The algorithm used to determine a `WebApplicationType` is fairly simple:

- If Spring MVC is present, an `AnnotationConfigServletWebServerApplicationContext` is used
- If Spring MVC is not present and Spring WebFlux is present, an `AnnotationConfigReactiveWebServerApplicationContext` is used
- Otherwise, `AnnotationConfigApplicationContext` is used

This means that if you are using Spring MVC and the new `WebClient` from Spring WebFlux in the same application, Spring MVC will be used by default. You can override that easily by calling `setWebApplicationType(WebApplicationType)`.

It is also possible to take complete control of the `ApplicationContext` type that is used by calling `setApplicationContextClass(...)`.



It is often desirable to call `setWebApplicationType(WebApplicationType.NONE)` when using `SpringApplication` within a JUnit test.

23.7 Accessing Application Arguments

If you need to access the application arguments that were passed to `SpringApplication.run(...)`, you can inject a `org.springframework.boot.ApplicationArguments` bean.

The `ApplicationArguments` interface provides access to both the raw `String[]` arguments as well as parsed `option` and `non-option` arguments, as shown in the following example:

```
import org.springframework.boot.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.stereotype.*;

@Component
public class MyBean {

    @Autowired
    public MyBean(ApplicationArguments args) {
        boolean debug = args.containsOption("debug");
        List<String> files = args.getNonOptionArgs();
        // if run with "--debug Logfile.txt" debug=true,
        files=[ "Logfile.txt" ]
    }
}
```

```
}
```



Spring Boot also registers a `CommandLinePropertySource` with the Spring `Environment`. This lets you also inject single application arguments by using the `@Value` annotation.

23.8 Using the ApplicationRunner or CommandLineRunner

If you need to run some specific code once the `SpringApplication` has started, you can implement the `ApplicationRunner` or `CommandLineRunner` interfaces. Both interfaces work in the same way and offer a single `run` method, which is called just before `SpringApplication.run(...)` completes.

The `CommandLineRunner` interface provides access to application arguments as a simple string array, whereas the `ApplicationRunner` uses the `ApplicationArguments` interface discussed earlier. The following example shows a `CommandLineRunner` with a `run` method:

```
import org.springframework.boot.*;
import org.springframework.stereotype.*;

@Component
public class MyBean implements CommandLineRunner {

    public void run(String... args) {
        // Do something...
    }
}
```

If several `CommandLineRunner` or `ApplicationRunner` beans are defined that must be called in a specific order, you can additionally implement the `org.springframework.core.Ordered` interface or use the `org.springframework.core.annotation.Order` annotation.

23.9 Application Exit

Each `SpringApplication` registers a shutdown hook with the JVM to ensure that the `ApplicationContext` closes gracefully on exit. All the standard Spring lifecycle callbacks (such as the `DisposableBean` interface or the `@PreDestroy` annotation) can be used.

In addition, beans may implement the `org.springframework.boot.ExitCodeGenerator` interface if they wish to return a specific exit code when `SpringApplication.exit()` is called. This exit code can then be passed to `System.exit()` to return it as a status code, as shown in the following example:

```
@SpringBootApplication
public class ExitCodeApplication {

    @Bean
    public ExitCodeGenerator exitCodeGenerator() {
        return () -> 42;
    }

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

Also, the `ExitCodeGenerator` interface may be implemented by exceptions. When such an exception is encountered, Spring Boot returns the exit code provided by the implemented `getExitCode()` method.

23.10 Admin Features

It is possible to enable admin-related features for the application by specifying the `spring.application.admin.enabled` property. This exposes the `SpringApplicationAdminMXBean` on the platform `MBeanServer`. You could use this feature to administer your Spring Boot application remotely. This feature could also be useful for any service wrapper implementation.

24. Externalized Configuration

Spring Boot lets you externalize your configuration so that you can work with the same application code in different environments. You can use properties files, YAML files, environment variables, and command-line arguments to externalize configuration. Property values can be injected directly into your beans by using the `@Value` annotation, accessed through Spring's `Environment` abstraction, or be [bound to structured objects](#) through `@ConfigurationProperties`.

Spring Boot uses a very particular `PropertySource` order that is designed to allow sensible overriding of values. Properties are considered in the following order:

1. [Devtools global settings properties](#) on your home directory (`~/.spring-boot-devtools.properties` when devtools is active).
2. `@TestPropertySource` annotations on your tests.
3. `properties` attribute on your tests. Available on `@SpringBootTest` and the [test annotations for testing a particular slice of your application](#).
4. Command line arguments.
5. Properties from `SPRING_APPLICATION_JSON` (inline JSON embedded in an environment variable or system property).
6. `ServletConfig` init parameters.
7. `ServletContext` init parameters.
8. JNDI attributes from `java:comp/env`.
9. Java System properties (`System.getProperties()`).
10. OS environment variables.
11. A `RandomValuePropertySource` that has properties only in `random.*`.
12. [Profile-specific application properties](#) outside of your packaged jar (`application-{profile}.properties` and YAML variants).
13. [Profile-specific application properties](#) packaged inside your jar (`application-{profile}.properties` and YAML variants).
14. Application properties outside of your packaged jar (`application.properties` and YAML variants).
15. Application properties packaged inside your jar (`application.properties` and YAML variants).
16. `@PropertySource` annotations on your `@Configuration` classes.

17. Default properties (specified by setting `SpringApplication.setDefaultProperties()`).

To provide a concrete example, suppose you develop a `@Component` that uses a `name` property, as shown in the following example:

```
import org.springframework.stereotype.*;
import org.springframework.beans.factory.annotation.*;

@Component
public class MyBean {

    @Value("${name}")
    private String name;

    // ...

}
```

On your application classpath (for example, inside your jar) you can have an `application.properties` file that provides a sensible default property value for `name`. When running in a new environment, an `application.properties` file can be provided outside of your jar that overrides the `name`. For one-off testing, you can launch with a specific command line switch (for example, `java -jar app.jar --name="Spring"`).



The `SPRING_APPLICATION_JSON` properties can be supplied on the command line with an environment variable. For example, you could use the following line in a UN*X shell:

```
$ SPRING_APPLICATION_JSON='{"acme":{"name":"test"}}' java -jar myapp.jar
```

In the preceding example, you end up with `acme.name=test` in the Spring `Environment`. You can also supply the JSON as `spring.application.json` in a System property, as shown in the following example:

```
$ java -Dspring.application.json='{"name":"test"}' -jar myapp.jar
```

You can also supply the JSON by using a command line argument, as shown in the following example:

```
$ java -jar myapp.jar --spring.application.json='{"name":"test"}'
```

You can also supply the JSON as a JNDI variable, as

follows: `java:comp/env/spring.application.json`.

24.1 Configuring Random Values

The `RandomValuePropertySource` is useful for injecting random values (for example, into secrets or test cases). It can produce integers, longs, uuids, or strings, as shown in the following example:

```
my.secret=${random.value}
my.number=${random.int}
my.bignumber=${random.long}
my.uuid=${random.uuid}
my.number.less.than.ten=${random.int(10)}
my.number.in.range=${random.int[1024,65536]}
```

The `random.int*` syntax is `OPEN value (,max) CLOSE` where the `OPEN,CLOSE` are any character and `value,max` are integers. If `max` is provided, then `value` is the minimum value and `max` is the maximum value (exclusive).

24.2 Accessing Command Line Properties

By default, `SpringApplication` converts any command line option arguments (that is, arguments starting with `--`, such as `--server.port=9000`) to a `property` and adds them to the Spring `Environment`. As mentioned previously, command line properties always take precedence over other property sources.

If you do not want command line properties to be added to the `Environment`, you can disable them by using `SpringApplication.setAddCommandLineProperties(false)`.

24.3 Application Property Files

`SpringApplication` loads properties from `application.properties` files in the following locations and adds them to the Spring `Environment`:

1. A `/config` subdirectory of the current directory
2. The current directory
3. A classpath `/config` package
4. The classpath root

The list is ordered by precedence (properties defined in locations higher in the list override those defined in lower locations).



You can also [use YAML \('.yml'\) files](#) as an alternative to '.properties'.

If you do not like `application.properties` as the configuration file name, you can switch to another file name by specifying a `spring.config.name` environment property. You can also refer to an explicit location by using the `spring.config.location` environment property (which is a comma-separated list of directory locations or file paths). The following example shows how to specify a different file name:

```
$ java -jar myproject.jar --spring.config.name=myproject
```

The following example shows how to specify two locations:

```
$ java -jar myproject.jar --  
spring.config.location=classpath:/default.properties,classpath:/override.p  
roperties
```



`spring.config.name` and `spring.config.location` are used very early to determine which files have to be loaded, so they must be defined as an environment property (typically an OS environment variable, a system property, or a command-line argument).

If `spring.config.location` contains directories (as opposed to files), they should end in `/` (and, at runtime, be appended with the names generated from `spring.config.name` before being loaded, including profile-specific file names). Files specified in `spring.config.location` are used as-is, with no support for profile-specific variants, and are overridden by any profile-specific properties.

Config locations are searched in reverse order. By default, the configured locations are `classpath:/,classpath:/config/,file:./,file:./config/`. The resulting search order is the following:

1. `file:./config/`
2. `file:./`
3. `classpath:/config/`
4. `classpath:/`

When custom config locations are configured by using `spring.config.location`, they replace the default locations. For example, if `spring.config.location` is configured with the value `classpath:/custom-config/,file:./custom-config/`, the search order becomes the following:

1. `file:./custom-config/`
2. `classpath:custom-config/`

Alternatively, when custom config locations are configured by using `spring.config.additional-location`, they are used in addition to the default locations. Additional locations are searched before the default locations. For example, if additional locations of `classpath:/custom-config/,file:./custom-config/` are configured, the search order becomes the following:

1. `file:./custom-config/`
2. `classpath:custom-config/`
3. `file:./config/`
4. `file:./`
5. `classpath:/config/`
6. `classpath:/`

This search ordering lets you specify default values in one configuration file and then selectively override those values in another. You can provide default values for your application in `application.properties` (or whatever other basename you choose with `spring.config.name`) in one of the default locations. These default values can then be overridden at runtime with a different file located in one of the custom locations.



If you use environment variables rather than system properties, most operating systems disallow period-separated key names, but you can use underscores instead (for example, `SPRING_CONFIG_NAME` instead of `spring.config.name`).



If your application runs in a container, then JNDI properties (in `java:comp/env`) or servlet context initialization parameters can be used instead of, or as well as, environment variables or system properties.

24.4 Profile-specific Properties

In addition to `application.properties` files, profile-specific properties can also be defined by using the following naming convention: `application-{profile}.properties`. The `Environment` has a set of default profiles (by default, `[default]`) that are used if no active profiles are set. In other words, if no profiles are explicitly activated, then properties from `application-default.properties` are loaded.

Profile-specific properties are loaded from the same locations as standard `application.properties`, with profile-specific files always overriding the non-specific ones, whether or not the profile-specific files are inside or outside your packaged jar.

If several profiles are specified, a last-wins strategy applies. For example, profiles specified by the `spring.profiles.active` property are added after those configured through the `SpringApplication` API and therefore take precedence.



If you have specified any files in `spring.config.location`, profile-specific variants of those files are not considered. Use directories in `spring.config.location` if you want to also use profile-specific properties.

24.5 Placeholders in Properties

The values in `application.properties` are filtered through the existing `Environment` when they are used, so you can refer back to previously defined values (for example, from System properties).

```
app.name=MyApp
```

```
app.description=${app.name} is a Spring Boot application
```



You can also use this technique to create “short” variants of existing Spring Boot properties. See the [Section 77.4, “Use ‘Short’ Command Line Arguments”](#) how-to for details.

24.6 Encrypting Properties

Spring Boot does not provide any built in support for encrypting property values, however, it does provide the hook points necessary to modify values contained in the Spring [Environment](#).

The [EnvironmentPostProcessor](#) interface allows you to manipulate the [Environment](#) before the application starts. See [Section 76.3, “Customize the Environment or ApplicationContext Before It Starts”](#) for details.

If you’re looking for a secure way to store credentials and passwords, the [Spring Cloud Vault](#) project provides support for storing externalized configuration in [HashiCorp Vault](#).

24.7 Using YAML Instead of Properties

[YAML](#) is a superset of JSON and, as such, is a convenient format for specifying hierarchical configuration data. The [SpringApplication](#) class automatically supports YAML as an alternative to properties whenever you have the [SnakeYAML](#) library on your classpath.



If you use “Starters”, SnakeYAML is automatically provided by [spring-boot-starter](#).

24.7.1 Loading YAML

Spring Framework provides two convenient classes that can be used to load YAML documents. The [YamlPropertiesFactoryBean](#) loads YAML as [Properties](#) and the [YamlMapFactoryBean](#) loads YAML as a [Map](#).

For example, consider the following YAML document:

```
environments:  
  dev:
```

```
    url: https://dev.example.com
    name: Developer Setup
prod:
    url: https://another.example.com
    name: My Cool App
```

The preceding example would be transformed into the following properties:

```
environments.dev.url=https://dev.example.com
environments.dev.name=Developer Setup
environments.prod.url=https://another.example.com
environments.prod.name=My Cool App
```

YAML lists are represented as property keys with [\[index\]](#) dereferencers. For example, consider the following YAML:

```
my:
servers:
  - dev.example.com
  - another.example.com
```

The preceding example would be transformed into these properties:

```
my.servers[0]=dev.example.com
my.servers[1]=another.example.com
```

To bind to properties like that by using Spring Boot's [Binder](#) utilities (which is what [@ConfigurationProperties](#) does), you need to have a property in the target bean of type [java.util.List](#) (or [Set](#)) and you either need to provide a setter or initialize it with a mutable value. For example, the following example binds to the properties shown previously:

```
@ConfigurationProperties(prefix="my")
public class Config {

    private List<String> servers = new ArrayList<String>();

    public List<String> getServers() {
        return this.servers;
    }
}
```

24.7.2 Exposing YAML as Properties in the Spring Environment

The `YamlPropertySourceLoader` class can be used to expose YAML as a `PropertySource` in the Spring `Environment`. Doing so lets you use the `@Value` annotation with placeholders syntax to access YAML properties.

24.7.3 Multi-profile YAML Documents

You can specify multiple profile-specific YAML documents in a single file by using a `spring.profiles` key to indicate when the document applies, as shown in the following example:

```
server:
    address: 192.168.1.100
---
spring:
    profiles: development
server:
    address: 127.0.0.1
---
spring:
    profiles: production & eu-central
server:
    address: 192.168.1.120
```

In the preceding example, if the `development` profile is active, the `server.address` property is `127.0.0.1`. Similarly, if the `production` and `eu-central` profiles are active, the `server.address` property is `192.168.1.120`. If the `development`, `production` and `eu-central` profiles are **not** enabled, then the value for the property is `192.168.1.100`.



`spring.profiles` can therefore contain a simple profile name (for example `production`) or a profile expression. A profile expression allows for more complicated profile logic to be expressed, for example `production & (eu-central | eu-west)`. Check the [reference guide](#) for more details.

If none are explicitly active when the application context starts, the default profiles are activated. So, in the following YAML, we set a value

for `spring.security.user.password` that is available **only** in the "default" profile:

```
server:
  port: 8000
---
spring:
  profiles: default
  security:
    user:
      password: weak
```

Whereas, in the following example, the password is always set because it is not attached to any profile, and it would have to be explicitly reset in all other profiles as necessary:

```
server:
  port: 8000
spring:
  security:
    user:
      password: weak
```

Spring profiles designated by using the `spring.profiles` element may optionally be negated by using the `!` character. If both negated and non-negated profiles are specified for a single document, at least one non-negated profile must match, and no negated profiles may match.

24.7.4 YAML Shortcomings

YAML files cannot be loaded by using the `@PropertySource` annotation. So, in the case that you need to load values that way, you need to use a properties file.

Using the multi YAML document syntax in profile-specific YAML files can lead to unexpected behavior. For example, consider the following config in a file called `application-dev.yml`, with the `dev` profile being active:

```
server:
  port: 8000
---
spring:
  profiles: !test
  security:
```

```
user:  
password: weak
```

In the example above, profile negation and profile expressions will not behave as expected. We recommend that you don't combine profile-specific YAML files and multiple YAML documents and stick to using only one of them.

24.8 Type-safe Configuration Properties

Using the `@Value("${property}")` annotation to inject configuration properties can sometimes be cumbersome, especially if you are working with multiple properties or your data is hierarchical in nature. Spring Boot provides an alternative method of working with properties that lets strongly typed beans govern and validate the configuration of your application, as shown in the following example:

```
package com.example;  
  
import java.net.InetAddress;  
import java.util.ArrayList;  
import java.util.Collections;  
import java.util.List;  
  
import org.springframework.boot.context.properties.ConfigurationProperties;  
  
@ConfigurationProperties("acme")  
public class AcmeProperties {  
  
    private boolean enabled;  
  
    private InetAddress remoteAddress;  
  
    private final Security security = new Security();  
  
    public boolean isEnabled() { ... }  
  
    public void setEnabled(boolean enabled) { ... }  
  
    public InetAddress getRemoteAddress() { ... }  
  
    public void setRemoteAddress(InetAddress remoteAddress) { ... }  
  
    public Security getSecurity() { ... }
```

```

public static class Security {

    private String username;

    private String password;

    private List<String> roles = new
ArrayList<>(Collections.singleton("USER"));

    public String getUsername() { ... }

    public void setUsername(String username) { ... }

    public String getPassword() { ... }

    public void setPassword(String password) { ... }

    public List<String> getRoles() { ... }

    public void setRoles(List<String> roles) { ... }

}
}

```

The preceding POJO defines the following properties:

- `acme.enabled`, with a value of `false` by default.
- `acme.remote-address`, with a type that can be coerced from `String`.
- `acme.security.username`, with a nested "security" object whose name is determined by the name of the property. In particular, the return type is not used at all there and could have been `SecurityProperties`.
- `acme.security.password`.
- `acme.security.roles`, with a collection of `String`.



Getters and setters are usually mandatory, since binding is through standard Java Beans property descriptors, just like in Spring MVC. A setter may be omitted in the following cases:

Maps, as long as they are initialized, need a getter but not necessarily a setter, since they can be mutated by the binder.

Collections and arrays can be accessed either through an index (typically with YAML) or by using

a single comma-separated value (properties). In the latter case, a setter is mandatory. We recommend to always add a setter for such types. If you initialize a collection, make sure it is not immutable (as in the preceding example).

If nested POJO properties are initialized (like the `Security` field in the preceding example), a setter is not required. If you want the binder to create the instance on the fly by using its default constructor, you need a setter.

Some people use Project Lombok to add getters and setters automatically. Make sure that Lombok does not generate any particular constructor for such a type, as it is used automatically by the container to instantiate the object.

Finally, only standard Java Bean properties are considered and binding on static properties is not supported.



See also the [differences between `@Value` and `@ConfigurationProperties`](#).

You also need to list the properties classes to register in the `@EnableConfigurationProperties` annotation, as shown in the following example:

```
@Configuration  
@EnableConfigurationProperties(AcmeProperties.class)  
public class MyConfiguration {  
}
```



When the `@ConfigurationProperties` bean is registered that way, the bean has a conventional name: `<prefix>-<fqn>`, where `<prefix>` is the environment key prefix specified in the `@ConfigurationProperties` annotation and `<fqn>` is the fully qualified name of the bean. If the annotation does not provide any prefix, only the fully qualified name of the bean is used.

The bean name in the example above is `acme-com.example.AcmeProperties`.

The preceding configuration creates a regular bean for `AcmeProperties`. We recommend that `@ConfigurationProperties` only deal with the environment and, in particular, does not inject other beans from the context. Keep in mind that the `@EnableConfigurationProperties` annotation is *also* automatically applied to your project so that any *existing* bean annotated with `@ConfigurationProperties` is configured from the `Environment`. Instead of annotating `MyConfiguration` with `@EnableConfigurationProperties(AcmePrope`

`rties.class)`, you could make `AcmeProperties` a bean, as shown in the following example:

```
@Component
@ConfigurationProperties(prefix="acme")
public class AcmeProperties {

    // ... see the preceding example

}
```

This style of configuration works particularly well with the `SpringApplication` external YAML configuration, as shown in the following example:

```
# application.yml

acme:
    remote-address: 192.168.1.1
    security:
        username: admin
        roles:
            - USER
            - ADMIN

# additional configuration as required
```

To work with `@ConfigurationProperties` beans, you can inject them in the same way as any other bean, as shown in the following example:

```
@Service
public class MyService {

    private final AcmeProperties properties;

    @Autowired
    public MyService(AcmeProperties properties) {
        this.properties = properties;
    }

    //...

    @PostConstruct
    public void openConnection() {
```

```
        Server server = new
Server(this.properties.getRemoteAddress());
        // ...
}
}
```



Using `@ConfigurationProperties` also lets you generate metadata files that can be used by IDEs to offer auto-completion for your own keys. See the [Appendix B, Configuration Metadata](#) appendix for details.

24.8.1 Third-party Configuration

As well as using `@ConfigurationProperties` to annotate a class, you can also use it on public `@Bean` methods. Doing so can be particularly useful when you want to bind properties to third-party components that are outside of your control.

To configure a bean from the `Environment` properties, add `@ConfigurationProperties` to its bean registration, as shown in the following example:

```
@ConfigurationProperties(prefix = "another")
@Bean
public AnotherComponent anotherComponent() {
    ...
}
```

Any property defined with the `another` prefix is mapped onto that `AnotherComponent` bean in manner similar to the preceding `AcmeProperties` example.

24.8.2 Relaxed Binding

Spring Boot uses some relaxed rules for binding `Environment` properties to `@ConfigurationProperties` beans, so there does not need to be an exact match between the `Environment` property name and the bean property name. Common examples where this is useful include dash-separated environment properties (for example, `context-path` binds to `contextPath`), and capitalized environment properties (for example, `PORT` binds to `port`).

For example, consider the following `@ConfigurationProperties` class:

```
@ConfigurationProperties(prefix="acme.my-project.person")
public class OwnerProperties {

    private String firstName;

    public String getFirstName() {
        return this.firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

}
```

In the preceding example, the following properties names can all be used:

Table 24.1. relaxed binding

	Note
son.firstName	Kebab case, which is recommended for use in <code>.properties</code> and <code>.yml</code> files.
son.firstName	Standard camel case syntax.
rson.first_name	Underscore notation, which is an alternative format for use in <code>.properties</code> and <code>.yml</code> files.
SON_FIRSTNAME	Upper case format, which is recommended when using system environment variables.



The `prefix` value for the annotation *must* be in kebab case (lowercase and separated by `-`, such as `acme.my-project.person`).

Table 24.2. relaxed binding rules per property source

Example	List
camel case, kebab case, or underscore notation	Standard list syntax using <code>[]</code> or comma-separated

Example	List
Camel case, kebab case, or underscore notation	Standard YAML list syntax or comma-separated values
Upper case format with underscore as the delimiter. <code>_</code> should not be used within a property name	Numeric values surrounded by underscores, such as <code>my.acme[1].other</code>
Camel case, kebab case, or underscore notation	Standard list syntax using <code>[]</code> or comma-separated values



We recommend that, when possible, properties are stored in lower-case kebab format, such as `my.property-name=acme`.

When binding to `Map` properties, if the `key` contains anything other than lowercase alpha-numeric characters or `-`, you need to use the bracket notation so that the original value is preserved. If the key is not surrounded by `[]`, any characters that are not alpha-numeric or `-` are removed. For example, consider binding the following properties to a `Map`:

```
acme:
  map:
    "[/key1]": value1
    "[/key2]": value2
    /key3: value3
```

The properties above will bind to a `Map` with `/key1`, `/key2` and `key3` as the keys in the map.

24.8.3 Merging Complex Types

When lists are configured in more than one place, overriding works by replacing the entire list.

For example, assume a `MyPojo` object with `name` and `description` attributes that are `null` by default. The following example exposes a list of `MyPojo` objects from `AcmeProperties`:

```
@ConfigurationProperties("acme")
public class AcmeProperties {
```

```

private final List<MyPojo> list = new ArrayList<>();

public List<MyPojo> getList() {
    return this.list;
}

}

```

Consider the following configuration:

```

acme:
  list:
    - name: my name
      description: my description
---
spring:
  profiles: dev
acme:
  list:
    - name: my another name

```

If the `dev` profile is not active, `AcmeProperties.list` contains one `MyPojo` entry, as previously defined. If the `dev` profile is enabled, however, the `list` still contains only one entry (with a name of `my another name` and a description of `null`). This configuration *does not* add a second `MyPojo` instance to the list, and it does not merge the items.

When a `List` is specified in multiple profiles, the one with the highest priority (and only that one) is used. Consider the following example:

```

acme:
  list:
    - name: my name
      description: my description
    - name: another name
      description: another description
---
spring:
  profiles: dev
acme:
  list:
    - name: my another name

```

In the preceding example, if the `dev` profile is active, `AcmeProperties.list` contains *one* `MyPojo` entry (with a name of `my another name` and a description of `null`). For YAML, both comma-separated lists and YAML lists can be used for completely overriding the contents of the list.

For `Map` properties, you can bind with property values drawn from multiple sources. However, for the same property in multiple sources, the one with the highest priority is used. The following example exposes a `Map<String, MyPojo>` from `AcmeProperties`:

```
@ConfigurationProperties("acme")
public class AcmeProperties {

    private final Map<String, MyPojo> map = new HashMap<>();

    public Map<String, MyPojo> getMap() {
        return this.map;
    }

}
```

Consider the following configuration:

```
acme:
  map:
    key1:
      name: my name 1
      description: my description 1
---
spring:
  profiles: dev
acme:
  map:
    key1:
      name: dev name 1
    key2:
      name: dev name 2
      description: dev description 2
```

If the `dev` profile is not active, `AcmeProperties.map` contains one entry with key `key1` (with a name of `my name 1` and a description of `my description 1`). If the `dev` profile is enabled, however, `map` contains two entries with

keys `key1` (with a name of `dev.name.1` and a description of `my.description.1`) and `key2` (with a name of `dev.name.2` and a description of `dev.description.2`).



The preceding merging rules apply to properties from all property sources and not just YAML files.

24.8.4 Properties Conversion

Spring Boot attempts to coerce the external application properties to the right type when it binds to the `@ConfigurationProperties` beans. If you need custom type conversion, you can provide a `ConversionService` bean (with a bean named `conversionService`) or custom property editors (through a `CustomEditorConfigurer` bean) or custom `Converters` (with bean definitions annotated as `@ConfigurationPropertiesBinding`).



As this bean is requested very early during the application lifecycle, make sure to limit the dependencies that your `ConversionService` is using. Typically, any dependency that you require may not be fully initialized at creation time. You may want to rename your custom `ConversionService` if it is not required for configuration keys coercion and only rely on custom converters qualified with `@ConfigurationPropertiesBinding`.

Converting durations

Spring Boot has dedicated support for expressing durations. If you expose a `java.time.Duration` property, the following formats in application properties are available:

- A regular `long` representation (using milliseconds as the default unit unless a `@DurationUnit` has been specified)
- The standard ISO-8601 format `used by java.util.Duration`
- A more readable format where the value and the unit are coupled (e.g. `10s` means 10 seconds)

Consider the following example:

```
@ConfigurationProperties("app.system")
```

```

public class AppSystemProperties {

    @DurationUnit(ChronoUnit.SECONDS)
    private Duration sessionTimeout = Duration.ofSeconds(30);

    private Duration readTimeout = Duration.ofMillis(1000);

    public Duration getSessionTimeout() {
        return this.sessionTimeout;
    }

    public void setSessionTimeout(Duration sessionTimeout) {
        this.sessionTimeout = sessionTimeout;
    }

    public Duration getReadTimeout() {
        return this.readTimeout;
    }

    public void setReadTimeout(Duration readTimeout) {
        this.readTimeout = readTimeout;
    }

}

```

To specify a session timeout of 30 seconds, `30`, `PT30S` and `30s` are all equivalent. A read timeout of 500ms can be specified in any of the following form: `500`, `PT0.5S` and `500ms`.

You can also use any of the supported units. These are:

- `ns` for nanoseconds
- `us` for microseconds
- `ms` for milliseconds
- `s` for seconds
- `m` for minutes
- `h` for hours
- `d` for days

The default unit is milliseconds and can be overridden using `@DurationUnit` as illustrated in the sample above.



If you are upgrading from a previous version that is simply using `Long` to express the duration, make sure to define the unit (using `@DurationUnit`) if it isn't milliseconds alongside the switch to `Duration`. Doing so gives a transparent upgrade path while supporting a much richer format.

Converting Data Sizes

Spring Framework has a `DataSize` value type that allows to express size in bytes. If you expose a `DataSize` property, the following formats in application properties are available:

- A regular `long` representation (using bytes as the default unit unless a `@DataSizeUnit` has been specified)
- A more readable format where the value and the unit are coupled (e.g. `10MB` means 10 megabytes)

Consider the following example:

```
@ConfigurationProperties("app.io")
public class AppIoProperties {

    @DataSizeUnit(DataUnit.MEGABYTES)
    private DataSize bufferSize = DataSize.ofMegabytes(2);

    private DataSize sizeThreshold = DataSize.ofBytes(512);

    public DataSize getBufferSize() {
        return this.bufferSize;
    }

    public void setBufferSize(DataSize bufferSize) {
        this.bufferSize = bufferSize;
    }

    public DataSize getSizeThreshold() {
        return this.sizeThreshold;
    }

    public void setSizeThreshold(DataSize sizeThreshold) {
        this.sizeThreshold = sizeThreshold;
    }
}
```

```
}
```

To specify a buffer size of 10 megabytes, `10` and `10MB` are equivalent. A size threshold of 256 bytes can be specified as `256` or `256B`.

You can also use any of the supported units. These are:

- `B` for bytes
- `KB` for kilobytes
- `MB` for megabytes
- `GB` for gigabytes
- `TB` for terabytes

The default unit is bytes and can be overridden using `@DataSizeUnit` as illustrated in the sample above.



If you are upgrading from a previous version that is simply using `Long` to express the size, make sure to define the unit (using `@DataSizeUnit`) if it isn't bytes alongside the switch to `DataSize`. Doing so gives a transparent upgrade path while supporting a much richer format.

24.8.5 @ConfigurationProperties Validation

Spring Boot attempts to validate `@ConfigurationProperties` classes whenever they are annotated with Spring's `@Validated` annotation. You can use JSR-303 `javax.validation` constraint annotations directly on your configuration class. To do so, ensure that a compliant JSR-303 implementation is on your classpath and then add constraint annotations to your fields, as shown in the following example:

```
@ConfigurationProperties(prefix="acme")
@Validated
public class AcmeProperties {

    @NotNull
    private InetAddress remoteAddress;

    // ... getters and setters

}
```



You can also trigger validation by annotating the `@Bean` method that creates the configuration properties with `@Validated`.

Although nested properties will also be validated when bound, it's good practice to also annotate the associated field as `@Valid`. This ensures that validation is triggered even if no nested properties are found. The following example builds on the preceding `AcmeProperties` example:

```
@ConfigurationProperties(prefix="acme")
@Validated
public class AcmeProperties {

    @NotNull
    private InetAddress remoteAddress;

    @Valid
    private final Security security = new Security();

    // ... getters and setters

    public static class Security {

        @NotEmpty
        public String username;

        // ... getters and setters
    }
}
```

You can also add a custom Spring `Validator` by creating a bean definition called `configurationPropertiesValidator`. The `@Bean` method should be declared `static`. The configuration properties validator is created very early in the application's lifecycle, and declaring the `@Bean` method as static lets the bean be created without having to instantiate the `@Configuration` class. Doing so avoids any problems that may be caused by early instantiation. There is a [property validation sample](#) that shows how to set things up.



The `spring-boot-actuator` module includes an endpoint that exposes all `@ConfigurationProperties` beans. Point your web browser to `/actuator/configprops` or use the equivalent JMX endpoint. See the "[Production ready features](#)" section for details.

24.8.6 `@ConfigurationProperties` vs. `@Value`

The `@Value` annotation is a core container feature, and it does not provide the same features as type-safe configuration properties. The following table summarizes the features that are supported by `@ConfigurationProperties` and `@Value`:

Feature	<code>@ConfigurationProperties</code>
Relaxed binding	Yes
Meta-data support	Yes
<code>SpEL</code> evaluation	No

If you define a set of configuration keys for your own components, we recommend you group them in a POJO annotated with `@ConfigurationProperties`. You should also be aware that, since `@Value` does not support relaxed binding, it is not a good candidate if you need to provide the value by using environment variables.

Finally, while you can write a `SpEL` expression in `@Value`, such expressions are not processed from [application property files](#).

25. Profiles

Spring Profiles provide a way to segregate parts of your application configuration and make it be available only in certain environments. Any `@Component` or `@Configuration` can be marked with `@Profile` to limit when it is loaded, as shown in the following example:

```
@Configuration  
@Profile("production")  
public class ProductionConfiguration {  
  
    // ...  
  
}
```

You can use a `spring.profiles.active` `Environment` property to specify which profiles are active. You can specify the property in any of the ways described earlier in this chapter. For example, you could include it in your `application.properties`, as shown in the following example:

```
spring.profiles.active=dev,hsqldb
```

You could also specify it on the command line by using the following switch: `-spring.profiles.active=dev,hsqldb`.

25.1 Adding Active Profiles

The `spring.profiles.active` property follows the same ordering rules as other properties: The highest `PropertySource` wins. This means that you can specify active profiles in `application.properties` and then **replace** them by using the command line switch.

Sometimes, it is useful to have profile-specific properties that **add** to the active profiles rather than replace them. The `spring.profiles.include` property can be used to unconditionally add active profiles. The `SpringApplication` entry point also has a Java API for setting additional profiles (that is, on top of those activated by the `spring.profiles.active` property). See the `setAdditionalProfiles()` method in [SpringApplication](#).

For example, when an application with the following properties is run by using the switch, `--spring.profiles.active=prod`, the `proddb` and `prodmq` profiles are also activated:

```
---  
my.property: fromyamlfile  
---  
spring.profiles: prod  
spring.profiles.include:  
  - proddb
```

- prodmq

25.2 Programmatically Setting Profiles

You can programmatically set active profiles by calling `SpringApplication.setAdditionalProfiles(...)` before your application runs. It is also possible to activate profiles by using Spring's `ConfigurableEnvironment` interface.

25.3 Profile-specific Configuration Files

Profile-specific variants of both `application.properties` (or `application.yml`) and files referenced through `@ConfigurationProperties` are considered as files and loaded. See "[Section 24.4, “Profile-specific Properties”](#)" for details.

26. Logging

Spring Boot uses [Commons Logging](#) for all internal logging but leaves the underlying log implementation open. Default configurations are provided for [Java Util Logging](#), [Log4J2](#), and [Logback](#). In each case, loggers are pre-configured to use console output with optional file output also available.

By default, if you use the “Starters”, Logback is used for logging. Appropriate Logback routing is also included to ensure that dependent libraries that use Java Util Logging, Commons Logging, Log4J, or SLF4J all work correctly.



There are a lot of logging frameworks available for Java. Do not worry if the above list seems confusing. Generally, you do not need to change your logging dependencies and the Spring Boot defaults work just fine.

26.1 Log Format

The default log output from Spring Boot resembles the following example:

```
2014-03-05 10:57:51.112  INFO 45469 --- [           main]
org.apache.catalina.core.StandardEngine : Starting Servlet Engine: Apache
Tomcat/7.0.52
2014-03-05 10:57:51.253  INFO 45469 --- [ost-startStop-1]
o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded
WebApplicationContext
2014-03-05 10:57:51.253  INFO 45469 --- [ost-startStop-1]
o.s.web.context.ContextLoader          : Root WebApplicationContext:
initialization completed in 1358 ms
2014-03-05 10:57:51.698  INFO 45469 --- [ost-startStop-1]
o.s.b.c.e.ServletRegistrationBean     : Mapping servlet:
'dispatcherServlet' to [/]
2014-03-05 10:57:51.702  INFO 45469 --- [ost-startStop-1]
o.s.b.c.embedded.FilterRegistrationBean : Mapping filter:
'hiddenHttpMethodFilter' to: [/*]
```

The following items are output:

- Date and Time: Millisecond precision and easily sortable.
- Log Level: `ERROR`, `WARN`, `INFO`, `DEBUG`, or `TRACE`.
- Process ID.
- A `---` separator to distinguish the start of actual log messages.
- Thread name: Enclosed in square brackets (may be truncated for console output).
- Logger name: This is usually the source class name (often abbreviated).
- The log message.



Logback does not have a `FATAL` level. It is mapped to `ERROR`.

26.2 Console Output

The default log configuration echoes messages to the console as they are written. By default, `ERROR`-level, `WARN`-level, and `INFO`-level messages are logged. You can also enable a “debug” mode by starting your application with a `--debug` flag.

```
$ java -jar myapp.jar --debug
```



You can also specify `debug=true` in your `application.properties`.

When the debug mode is enabled, a selection of core loggers (embedded container, Hibernate, and Spring Boot) are configured to output more information. Enabling the debug mode does *not* configure your application to log all messages with `DEBUG` level.

Alternatively, you can enable a “trace” mode by starting your application with a `--trace` flag (or `trace=true` in your `application.properties`). Doing so enables trace logging for a selection of core loggers (embedded container, Hibernate schema generation, and the whole Spring portfolio).

26.2.1 Color-coded Output

If your terminal supports ANSI, color output is used to aid readability. You can set `spring.output.ansi.enabled` to a [supported value](#) to override the auto detection.

Color coding is configured by using the `%clr` conversion word. In its simplest form, the converter colors the output according to the log level, as shown in the following example:

```
%clr(%5p)
```

The following table describes the mapping of log levels to colors:

Level	Color
<code>FATAL</code>	Red
<code>ERROR</code>	Red
<code>WARN</code>	Yellow
<code>INFO</code>	Green
<code>DEBUG</code>	Green
<code>TRACE</code>	Green

Alternatively, you can specify the color or style that should be used by providing it as an option to the conversion. For example, to make the text yellow, use the following setting:

```
%clr(%d{yyyy-MM-dd HH:mm:ss.SSS}){yellow}
```

The following colors and styles are supported:

- `blue`
- `cyan`
- `faint`
- `green`
- `magenta`
- `red`
- `yellow`

26.3 File Output

By default, Spring Boot logs only to the console and does not write log files. If you want to write log files in addition to the console output, you need to set a `logging.file` or `logging.path` property (for example, in your `application.properties`).

The following table shows how the `logging.*` properties can be used together:

Table 26.1. Logging properties

<code>logging.path</code>	Example	Description
<code>none</code>		Console only logging.
<code>none</code>	<code>my.log</code>	Writes to the specified log file. Names can be an exact location or relative to the current directory.
specific directory	<code>/var/log</code>	Writes <code>spring.log</code> to the specified directory. Names can be an exact location or relative to the current directory.

Log files rotate when they reach 10 MB and, as with console output, `ERROR`-level, `WARN`-level, and `INFO`-level messages are logged by default. Size limits can be changed using the `logging.file.max-size` property. Previously rotated

files are archived indefinitely unless the `logging.file.max-history` property has been set.



The logging system is initialized early in the application lifecycle. Consequently, logging properties are not found in property files loaded through `@PropertySource` annotations.



Logging properties are independent of the actual logging infrastructure. As a result, specific configuration keys (such as `logback.configurationFile` for Logback) are not managed by Spring Boot.

26.4 Log Levels

All the supported logging systems can have the logger levels set in the Spring `Environment` (for example, in `application.properties`) by using `logging.level.<logger-name>=<level>` where `level` is one of TRACE, DEBUG, INFO, WARN, ERROR, FATAL, or OFF. The `root` logger can be configured by using `logging.level.root`.

The following example shows potential logging settings in `application.properties`:

```
logging.level.root=WARN
logging.level.org.springframework.web=DEBUG
logging.level.org.hibernate=ERROR
```

26.5 Log Groups

It's often useful to be able to group related loggers together so that they can all be configured at the same time. For example, you might commonly change the logging levels for *all* Tomcat related loggers, but you can't easily remember top level packages.

To help with this, Spring Boot allows you to define logging groups in your Spring `Environment`. For example, here's how you could define a "tomcat" group by adding it to your `application.properties`:

```
logging.group.tomcat=org.apache.catalina, org.apache.coyote,  
org.apache.tomcat
```

Once defined, you can change the level for all the loggers in the group with a single line:

```
logging.level.tomcat=TRACE
```

Spring Boot includes the following pre-defined logging groups that can be used out-of-the-box:

Name	Loggers
web	<code>org.springframework.core.codec</code> , <code>org.springframework.http</code> , <code>org.springframework.web</code>
sql	<code>org.springframework.jdbc.core</code> , <code>org.hibernate.SQL</code>

26.6 Custom Log Configuration

The various logging systems can be activated by including the appropriate libraries on the classpath and can be further customized by providing a suitable configuration file in the root of the classpath or in a location specified by the following Spring `Environment` property: `logging.config`.

You can force Spring Boot to use a particular logging system by using the `org.springframework.boot.logging.LoggingSystem` system property. The value should be the fully qualified class name of a `LoggingSystem` implementation. You can also disable Spring Boot's logging configuration entirely by using a value of `none`.



Since logging is initialized **before** the `ApplicationContext` is created, it is not possible to control logging from `@PropertySources` in Spring `@Configuration` files. The only way to change the logging system or disable it entirely is via System properties.

Depending on your logging system, the following files are loaded:

Logging System	Customization

Logging System	Customization
Logback	<code>logback-spring.xml</code> , <code>logback-spring.groovy</code> , <code>logback.xml</code> , or <code>logback.groovy</code>
Log4j2	<code>log4j2-spring.xml</code> or <code>log4j2.xml</code>
JDK (Java Util Logging)	<code>logging.properties</code>



When possible, we recommend that you use the `-spring` variants for your logging configuration (other than `logback.xml`). If you use standard configuration locations, Spring cannot correctly map them.



There are known classloading issues with Java Util Logging that cause problems when running from an 'executable jar'. We recommend that you avoid it when running from an 'executable jar' if at all possible.

To help with the customization, some other properties are transferred from the Spring `Environment` to System properties, as described in the following table:

Spring Environment	System Property	Comments
<code>logging.exception-conversion-word</code>	<code>LOG_EXCEPTION_CONVERSION_WORD</code>	The conversion word used when
<code>logging.file</code>	<code>LOG_FILE</code>	If defined, it is used in the default
<code>logging.file.max-size</code>	<code>LOG_FILE_MAX_SIZE</code>	Maximum log file size (if <code>LOG_</code>
<code>logging.file.max-history</code>	<code>LOG_FILE_MAX_HISTORY</code>	Maximum number of archive logs (in default Logback setup.)
<code>logging.path</code>	<code>LOG_PATH</code>	If defined, it is used in the default
<code>logging.pattern.console</code>	<code>CONSOLE_LOG_PATTERN</code>	The log pattern to use on the console

Spring Environment	System Property	Comments
<code>logging.pattern.dateformat</code>	<code>LOG_DATEFORMAT_PATTERN</code>	Appender pattern for log date format.
<code>logging.pattern.file</code>	<code>FILE_LOG_PATTERN</code>	The log pattern to use in a file (if setup.)
<code>logging.pattern.level</code>	<code>LOG_LEVEL_PATTERN</code>	The format to use when rendering Logback setup.)
<code>PID</code>	<code>PID</code>	The current process ID (discovered variable).

All the supported logging systems can consult System properties when parsing their configuration files. See the default configurations in `spring-boot.jar` for examples:

- [Logback](#)
- [Log4j 2](#)
- [Java Util logging](#)



If you want to use a placeholder in a logging property, you should use [Spring Boot's syntax](#) and not the syntax of the underlying framework. Notably, if you use Logback, you should use `:` as the delimiter between a property name and its default value and not use `:-`.



You can add MDC and other ad-hoc content to log lines by overriding only the `LOG_LEVEL_PATTERN` (or `logging.pattern.level` with Logback). For example, if you use `logging.pattern.level=user:%X{user} %5p`, then the default log format contains an MDC entry for "user", if it exists, as shown in the following example.

```
2015-09-30 12:30:04.031 user:someone INFO 22174 --- [nio-8080-exec-0] demo.Controller
Handling authenticated request
```

26.7 Logback Extensions

Spring Boot includes a number of extensions to Logback that can help with advanced configuration. You can use these extensions in your `logback-spring.xml` configuration file.



Because the standard `logback.xml` configuration file is loaded too early, you cannot use extensions in it. You need to either use `logback-spring.xml` or define a `logging.config` property.



The extensions cannot be used with Logback's [configuration scanning](#). If you attempt to do so, making changes to the configuration file results in an error similar to one of the following being logged:

```
ERROR in ch.qos.logback.core.joran.spi.Interpreter@4:71 - no applicable action for [springProperty], current ElementPath is [[configuration][springProperty]]  
ERROR in ch.qos.logback.core.joran.spi.Interpreter@4:71 - no applicable action for [springProfile], current ElementPath is [[configuration][springProfile]]
```

26.7.1 Profile-specific Configuration

The `<springProfile>` tag lets you optionally include or exclude sections of configuration based on the active Spring profiles. Profile sections are supported anywhere within the `<configuration>` element. Use the `name` attribute to specify which profile accepts the configuration. The `<springProfile>` tag can contain a simple profile name (for example `staging`) or a profile expression. A profile expression allows for more complicated profile logic to be expressed, for example `production & (eu-central | eu-west)`. Check the [reference guide](#) for more details. The following listing shows three sample profiles:

```
<springProfile name="staging">  
    <!-- configuration to be enabled when the "staging" profile is active --&gt;<br/></springProfile>  
  
<springProfile name="dev | staging">  
    <!-- configuration to be enabled when the "dev" or "staging" profiles are active --&gt;<br/></springProfile>
```

```
<springProfile name="!production">
    <!-- configuration to be enabled when the "production" profile is
not active -->
</springProfile>
```

26.7.2 Environment Properties

The `<springProperty>` tag lets you expose properties from the Spring `Environment` for use within Logback. Doing so can be useful if you want to access values from your `application.properties` file in your Logback configuration. The tag works in a similar way to Logback's standard `<property>` tag. However, rather than specifying a direct `value`, you specify the `source` of the property (from the `Environment`). If you need to store the property somewhere other than in `local` scope, you can use the `scope` attribute. If you need a fallback value (in case the property is not set in the `Environment`), you can use the `defaultValue` attribute. The following example shows how to expose properties for use within Logback:

```
<springProperty scope="context" name="fluentHost"
source="myapp.fluentd.host"
        defaultValue="localhost"/>
<appender name="FLUENT"
class="ch.qos.logback.more.appenders.DataFluentAppender">
    <remoteHost>${fluentHost}</remoteHost>
    ...
</appender>
```

27. Internationalization

Spring Boot supports localized messages so that your application can cater to users of different language preferences. By default, Spring Boot looks for the presence of a `messages` resource bundle at the root of the classpath.



The auto-configuration applies when the default properties file for the configured resource bundle is available (i.e. `messages.properties` by default). If your resource bundle contains only language-specific properties files, you are required to add the default.

The basename of the resource bundle as well as several other attributes can be configured using the `spring.messages` namespace, as shown in the following example:

```
spring.messages.basename=messages,config.i18n.messages
spring.messages.fallback-to-system-locale=false
```

`spring.messages.basename` supports comma-separated list of locations, either a package qualifier or a resource resolved from the classpath root.

28. JSON

Spring Boot provides integration with three JSON mapping libraries:

- Gson
- Jackson
- JSON-B

Jackson is the preferred and default library.

28.1 Jackson

Auto-configuration for Jackson is provided and Jackson is part of `spring-boot-starter-json`. When Jackson is on the classpath an `ObjectMapper` bean is automatically configured. Several configuration properties are provided for [customizing the configuration of the `ObjectMapper`](#).

28.2 Gson

Auto-configuration for Gson is provided. When Gson is on the classpath a `Gson` bean is automatically configured. Several `spring.gson.*` configuration properties are provided for customizing the configuration. To take more control, one or more `GsonBuilderCustomizer` beans can be used.

28.3 JSON-B

Auto-configuration for JSON-B is provided. When the JSON-B API and an

implementation are on the classpath a `Jsonb` bean will be automatically configured. The preferred JSON-B implementation is Apache Johnzon for which dependency management is provided.

29. Developing Web Applications

Spring Boot is well suited for web application development. You can create a self-contained HTTP server by using embedded Tomcat, Jetty, Undertow, or Netty. Most web applications use the `spring-boot-starter-web` module to get up and running quickly. You can also choose to build reactive web applications by using the `spring-boot-starter-webflux` module.

If you have not yet developed a Spring Boot web application, you can follow the "Hello World!" example in the [Getting started](#) section.

29.1 The “Spring Web MVC Framework”

The [Spring Web MVC framework](#) (often referred to as simply “Spring MVC”) is a rich “model view controller” web framework. Spring MVC lets you create special `@Controller` or `@RestController` beans to handle incoming HTTP requests. Methods in your controller are mapped to HTTP by using `@RequestMapping` annotations.

The following code shows a typical `@RestController` that serves JSON data:

```
@RestController
@RequestMapping(value="/users")
public class MyRestController {

    @RequestMapping(value="/{user}", method=RequestMethod.GET)
    public User getUser(@PathVariable Long user) {
        // ...
    }

    @RequestMapping(value="/{user}/customers",
method=RequestMethod.GET)
    List<Customer> getUserCustomers(@PathVariable Long user) {
```

```

        // ...
    }

    @RequestMapping(value="/{user}", method=RequestMethod.DELETE)
    public User deleteUser(@PathVariable Long user) {
        // ...
    }

}

```

Spring MVC is part of the core Spring Framework, and detailed information is available in the [reference documentation](#). There are also several guides that cover Spring MVC available at [spring.io/guides](#).

29.1.1 Spring MVC Auto-configuration

Spring Boot provides auto-configuration for Spring MVC that works well with most applications.

The auto-configuration adds the following features on top of Spring's defaults:

- Inclusion of `ContentNegotiatingViewResolver` and `BeanNameViewResolver` beans.
- Support for serving static resources, including support for WebJars (covered [later in this document](#)).
- Automatic registration of `Converter`, `GenericConverter`, and `Formatter` beans.
- Support for `HttpMessageConverters` (covered [later in this document](#)).
- Automatic registration of `MessageCodesResolver` (covered [later in this document](#)).
- Static `index.html` support.
- Custom `Favicon` support (covered [later in this document](#)).
- Automatic use of a `ConfigurableWebBindingInitializer` bean (covered [later in this document](#)).

If you want to keep Spring Boot MVC features and you want to add additional [MVC configuration](#) (interceptors, formatters, view controllers, and other features), you can add your own `@Configuration` class of type `WebMvcConfigurer` but **without** `@EnableWebMvc`. If you wish to provide

custom instances of `RequestMappingHandlerMapping`, `RequestMappingHandlerAdapter`, or `ExceptionHandlerExceptionResolver`, you can declare a `WebMvcRegistrationsAdapter` instance to provide such components.

If you want to take complete control of Spring MVC, you can add your own `@Configuration` annotated with `@EnableWebMvc`.

29.1.2 HttpMessageConverters

Spring MVC uses the `HttpMessageConverter` interface to convert HTTP requests and responses. Sensible defaults are included out of the box. For example, objects can be automatically converted to JSON (by using the Jackson library) or XML (by using the Jackson XML extension, if available, or by using JAXB if the Jackson XML extension is not available). By default, strings are encoded in `UTF-8`.

If you need to add or customize converters, you can use Spring Boot's `HttpMessageConverters` class, as shown in the following listing:

```
import org.springframework.boot.autoconfigure.http.HttpMessageConverters;
import org.springframework.context.annotation.*;
import org.springframework.http.converter.*;

@Configuration
public class MyConfiguration {

    @Bean
    public HttpMessageConverters customConverters() {
        HttpMessageConverter<?> additional = ...
        HttpMessageConverter<?> another = ...
        return new HttpMessageConverters(additional, another);
    }
}
```

Any `HttpMessageConverter` bean that is present in the context is added to the list of converters. You can also override default converters in the same way.

29.1.3 Custom JSON Serializers and Deserializers

If you use Jackson to serialize and deserialize JSON data, you might want to write your own `JsonSerializer` and `JsonDeserializer` classes. Custom serializers are usually [registered with Jackson through a module](#), but Spring Boot provides an alternative `@JsonComponent` annotation that makes it easier to directly register Spring Beans.

You can use the `@JsonComponent` annotation directly on `JsonSerializer` or `JsonDeserializer` implementations. You can also use it on classes that contain serializers/deserializers as inner classes, as shown in the following example:

```
import java.io.*;
import com.fasterxml.jackson.core.*;
import com.fasterxml.jackson.databind.*;
import org.springframework.boot.jackson.*;

@JsonComponent
public class Example {

    public static class Serializer extends
JsonSerializer<SomeObject> {
        // ...
    }

    public static class Deserializer extends
JsonDeserializer<SomeObject> {
        // ...
    }
}
```

All `@JsonComponent` beans in the `ApplicationContext` are automatically registered with Jackson. Because `@JsonComponent` is meta-annotated with `@Component`, the usual component-scanning rules apply.

Spring Boot also provides `JsonObjectSerializer` and `JsonObjectDeserializer` base classes that provide useful alternatives to the standard Jackson versions when serializing objects. See `JsonObjectSerializer` and `JsonObjectDeserializer` in the Javadoc for details.

29.1.4 MessageCodesResolver

Spring MVC has a strategy for generating error codes for rendering error messages from binding errors: `MessageCodesResolver`. If you set the `spring.mvc.message-codes-resolver.format` property `PREFIX_ERROR_CODE` or `POSTFIX_ERROR_CODE`, Spring Boot creates one for you (see the enumeration in `DefaultMessageCodesResolver.Format`).

29.1.5 Static Content

By default, Spring Boot serves static content from a directory called `/static` (or `/public` or `/resources` or `/META-INF/resources`) in the classpath or from the root of the `ServletContext`. It uses the `ResourceHttpRequestHandler` from Spring MVC so that you can modify that behavior by adding your own `WebMvcConfigurer` and overriding the `addResourceHandlers` method.

In a stand-alone web application, the default servlet from the container is also enabled and acts as a fallback, serving content from the root of the `ServletContext` if Spring decides not to handle it. Most of the time, this does not happen (unless you modify the default MVC configuration), because Spring can always handle requests through the `DispatcherServlet`.

By default, resources are mapped on `/**`, but you can tune that with the `spring.mvc.static-path-pattern` property. For instance, relocating all resources to `/resources/**` can be achieved as follows:

```
spring.mvc.static-path-pattern=/resources/**
```

You can also customize the static resource locations by using the `spring.resources.static-locations` property (replacing the default values with a list of directory locations). The root Servlet context path, `"/"`, is automatically added as a location as well.

In addition to the “standard” static resource locations mentioned earlier, a special case is made for [Webjars content](#). Any resources with a path in `/webjars/**` are served from jar files if they are packaged in the

Webjars format.



Do not use the `src/main/webapp` directory if your application is packaged as a jar. Although this directory is a common standard, it works **only** with war packaging, and it is silently ignored by most build tools if you generate a jar.

Spring Boot also supports the advanced resource handling features provided by Spring MVC, allowing use cases such as cache-busting static resources or using version agnostic URLs for Webjars.

To use version agnostic URLs for Webjars, add the `webjars-locator-core` dependency. Then declare your Webjar. Using jQuery as an example, adding `"/webjars/jquery/jquery.min.js"` results in `"/webjars/jquery/x.y.z/jquery.min.js"`. where `x.y.z` is the Webjar version.



If you use JBoss, you need to declare the `webjars-locator-jboss vfs` dependency instead of the `webjars-locator-core`. Otherwise, all Webjars resolve as a **404**.

To use cache busting, the following configuration configures a cache busting solution for all static resources, effectively adding a content hash, such as `<link href="/css/spring-2a2d595e6ed9a0b24f027f2b63b134d6.css" />`, in URLs:

```
spring.resources.chain.strategy.content.enabled=true  
spring.resources.chain.strategy.content.paths=/**
```



Links to resources are rewritten in templates at runtime, thanks to

a `ResourceUrlEncodingFilter` that is auto-configured for Thymeleaf and FreeMarker. You should manually declare this filter when using JSPs. Other template engines are currently not automatically supported but can be with custom template macros/helpers and the use of the `ResourceUrlProvider`.

When loading resources dynamically with, for example, a JavaScript module loader, renaming files is not an option. That is why other strategies are also supported and can be combined. A "fixed" strategy adds a static version string in the URL without changing the file name, as shown in the following example:

```
spring.resources.chain.strategy.content.enabled=true
spring.resources.chain.strategy.content.paths=/*
spring.resources.chain.strategy.fixed.enabled=true
spring.resources.chain.strategy.fixed.paths=/js/lib/
spring.resources.chain.strategy.fixed.version=v12
```

With this configuration, JavaScript modules located under `"/js/lib/"` use a fixed versioning strategy (`"/v12/js/lib/mymodule.js"`), while other resources still use the content one (`<link href="/css/spring-2a2d595e6ed9a0b24f027f2b63b134d6.css"/>`).

See [ResourceProperties](#) for more supported options.



This feature has been thoroughly described in a dedicated [blog post](#) and in Spring Framework's [reference documentation](#).

29.1.6 Welcome Page

Spring Boot supports both static and templated welcome pages. It first looks for an `index.html` file in the configured static content locations. If one is not found, it then looks for an `index` template. If either is found, it is automatically used as the welcome page of the application.

29.1.7 Custom Favicon

Spring Boot looks for a `favicon.ico` in the configured static content locations and the root of the classpath (in that order). If such a file is

present, it is automatically used as the favicon of the application.

29.1.8 Path Matching and Content Negotiation

Spring MVC can map incoming HTTP requests to handlers by looking at the request path and matching it to the mappings defined in your application (for example, `@GetMapping` annotations on Controller methods).

Spring Boot chooses to disable suffix pattern matching by default, which means that requests like `"GET /projects/spring-boot.json"` won't be matched to `@GetMapping("/projects/spring-boot")` mappings. This is considered as a [best practice for Spring MVC applications](#). This feature was mainly useful in the past for HTTP clients which did not send proper "Accept" request headers; we needed to make sure to send the correct Content Type to the client. Nowadays, Content Negotiation is much more reliable.

There are other ways to deal with HTTP clients that don't consistently send proper "Accept" request headers. Instead of using suffix matching, we can use a query parameter to ensure that requests like `"GET /projects/spring-boot?format=json"` will be mapped to `@GetMapping("/projects/spring-boot")`:

```
spring.mvc.contentnegotiation.favor-parameter=true

# We can change the parameter name, which is "format" by default:
# spring.mvc.contentnegotiation.parameter-name=myparam

# We can also register additional file extensions/media types with:
spring.mvc.contentnegotiation.media-types.markdown=text/markdown
```

If you understand the caveats and would still like your application to use suffix pattern matching, the following configuration is required:

```
spring.mvc.contentnegotiation.favor-path-extension=true
spring.mvc.pathmatch.use-suffix-pattern=true
```

Alternatively, rather than open all suffix patterns, it's more secure to just support registered suffix patterns:

```
spring.mvc.contentnegotiation.favor-path-extension=true
spring.mvc.pathmatch.use-registered-suffix-pattern=true
```

```
# You can also register additional file extensions/media types with:  
# spring.mvc.contentnegotiation.media-types.adoc=text/asciidoc
```

29.1.9 ConfigurableWebBindingInitializer

Spring MVC uses a `WebBindingInitializer` to initialize a `WebDataBinder` for a particular request. If you create your own `ConfigurableWebBindingInitializer@Bean`, Spring Boot automatically configures Spring MVC to use it.

29.1.10 Template Engines

As well as REST web services, you can also use Spring MVC to serve dynamic HTML content. Spring MVC supports a variety of templating technologies, including Thymeleaf, FreeMarker, and JSPs. Also, many other templating engines include their own Spring MVC integrations.

Spring Boot includes auto-configuration support for the following templating engines:

- [FreeMarker](#)
- [Groovy](#)
- [Thymeleaf](#)
- [Mustache](#)



If possible, JSPs should be avoided. There are several [known limitations](#) when using them with embedded servlet containers.

When you use one of these templating engines with the default configuration, your templates are picked up automatically from `src/main/resources/templates`.



Depending on how you run your application, IntelliJ IDEA orders the classpath differently. Running your application in the IDE from its main method results

in a different ordering than when you run your application by using Maven or Gradle or from its packaged jar. This can cause Spring Boot to fail to find the templates on the classpath. If you have this problem, you can reorder the classpath in the IDE to place the module's classes and resources first. Alternatively, you can configure the template prefix to search every `templates` directory on the classpath, as follows: `classpath*:/templates/`.

29.1.11 Error Handling

By default, Spring Boot provides an `/error` mapping that handles all errors in a sensible way, and it is registered as a “global” error page in the servlet container. For machine clients, it produces a JSON response with details of the error, the HTTP status, and the exception message. For browser clients, there is a “whitelabel” error view that renders the same data in HTML format (to customize it, add a `View` that resolves to `error`). To replace the default behavior completely, you can implement `ErrorController` and register a bean definition of that type or add a bean of type `ErrorAttributes` to use the existing mechanism but replace the contents.



The `BasicErrorController` can be used as a base class for a custom `ErrorController`. This is particularly useful if you want to add a handler for a new content type (the default is to handle `text/html` specifically and provide a fallback for everything else). To do so, extend `BasicErrorController`, add a public method with a `@RequestMapping` that has a `produces` attribute, and create a bean of your new type.

You can also define a class annotated with `@ControllerAdvice` to customize the JSON document to return for a particular controller and/or exception type, as shown in the following example:

```
@ControllerAdvice(basePackageClasses = AcmeController.class)
public class AcmeControllerAdvice extends
 ResponseEntityExceptionHandler {

    @ExceptionHandler(YourException.class)
    @ResponseBody
```

```

        ResponseEntity<?> handleControllerException(HttpServletRequest request, Throwable ex) {
            HttpStatus status = getStatus(request);
            return new ResponseEntity<?>(new CustomErrorResponse(status.value(), ex.getMessage()), status);
        }

    private HttpStatus getStatus(HttpServletRequest request) {
        Integer statusCode = (Integer)
request.getAttribute("javax.servlet.error.status_code");
        if (statusCode == null) {
            return HttpStatus.INTERNAL_SERVER_ERROR;
        }
        return HttpStatus.valueOf(statusCode);
    }

}

```

In the preceding example, if `YourException` is thrown by a controller defined in the same package as `AcmeController`, a JSON representation of the `CustomErrorResponse` POJO is used instead of the `ErrorAttributes` representation.

Custom Error Pages

If you want to display a custom HTML error page for a given status code, you can add a file to an `/error` folder. Error pages can either be static HTML (that is, added under any of the static resource folders) or be built by using templates. The name of the file should be the exact status code or a series mask.

For example, to map `404` to a static HTML file, your folder structure would be as follows:

```

src/
+- main/
  +- java/
  |   + <source code>
  +- resources/
  |   +- public/
  |       +- error/
  |           +- 404.html
  |           +- <other public assets>

```

To map all **5xx** errors by using a FreeMarker template, your folder structure would be as follows:

```
src/
+- main/
  +- java/
    |   + <source code>
  +- resources/
    +- templates/
      +- error/
        |   +- 5xx.ftl
      +- <other templates>
```

For more complex mappings, you can also add beans that implement the **ErrorViewResolver** interface, as shown in the following example:

```
public class MyErrorViewResolver implements ErrorViewResolver {

    @Override
    public ModelAndView resolveErrorView(HttpServletRequest request,
                                         HttpStatus status, Map<String, Object> model) {
        // Use the request or status to optionally return a
        ModelAndView
        return ...
    }

}
```

You can also use regular Spring MVC features such as **@ExceptionHandler** methods and **@ControllerAdvice**. The **ErrorController** then picks up any unhandled exceptions.

Mapping Error Pages outside of Spring MVC

For applications that do not use Spring MVC, you can use the **ErrorPageRegistrar** interface to directly register **ErrorPages**. This abstraction works directly with the underlying embedded servlet container and works even if you do not have a Spring MVC **DispatcherServlet**.

```
@Bean
public ErrorPageRegistrar errorPageRegistrar(){
    return new MyErrorPageRegistrar();
```

```

}

// ...

private static class MyErrorHandlerRegistrar implements
ErrorHandlerRegistrar {

    @Override
    public void registerErrorPages(ErrorPageRegistry registry) {
        registry.addErrorPages(new
ErrorPage(HttpStatus.BAD_REQUEST, "/400"));
    }

}

```



If you register an `ErrorPage` with a path that ends up being handled by a `Filter` (as is common with some non-Spring web frameworks, like Jersey and Wicket), then the `Filter` has to be explicitly registered as an `ERROR` dispatcher, as shown in the following example:

```

@Bean
public FilterRegistrationBean myFilter() {
    FilterRegistrationBean registration = new
FilterRegistrationBean();
    registration.setFilter(new MyFilter());
    ...
    registration.setDispatcherTypes(EnumSet.allOf(DispatcherType.c1
ass));
    return registration;
}

```

Note that the default `FilterRegistrationBean` does not include the `ERROR` dispatcher type.

CAUTION:When deployed to a servlet container, Spring Boot uses its error page filter to forward a request with an error status to the appropriate error page. The request can only be forwarded to the correct error page if the response has not already been committed. By default, WebSphere Application Server 8.0 and later commits the response upon successful completion of a servlet's service method. You should disable this behavior by setting `com.ibm.ws.webcontainer.invokeFlushAfterService` to `false`.

29.1.12 Spring HATEOAS

If you develop a RESTful API that makes use of hypermedia, Spring Boot provides auto-configuration for Spring HATEOAS that works well with most applications. The auto-configuration replaces the need to use `@EnableHypermediaSupport` and registers a number of beans to ease building hypermedia-based applications, including a `LinkDiscoverers` (for client side support) and an `ObjectMapper` configured to correctly marshal responses into the desired representation. The `ObjectMapper` is customized by setting the various `spring.jackson.*` properties or, if one exists, by a `Jackson2ObjectMapperBuilder` bean.

You can take control of Spring HATEOAS's configuration by using `@EnableHypermediaSupport`. Note that doing so disables the `ObjectMapper` customization described earlier.

29.1.13 CORS Support

[Cross-origin resource sharing](#) (CORS) is a [W3C specification](#) implemented by [most browsers](#) that lets you specify in a flexible way what kind of cross-domain requests are authorized, instead of using some less secure and less powerful approaches such as IFRAME or JSONP.

As of version 4.2, Spring MVC [supports CORS](#). Using [controller method CORS configuration](#) with `@CrossOrigin` annotations in your Spring Boot application does not require any specific configuration. [Global CORS configuration](#) can be defined by registering a `WebMvcConfigurer` bean with a customized `addCorsMappings(CorsRegistry)` method, as shown in the following example:

```
@Configuration
public class MyConfiguration {

    @Bean
    public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurer() {
            @Override
            public void addCorsMappings(CorsRegistry
registry) {
                registry.addMapping("/api/**");
            }
        };
    }
}
```

```
}
```

29.2 The “Spring WebFlux Framework”

Spring WebFlux is the new reactive web framework introduced in Spring Framework 5.0. Unlike Spring MVC, it does not require the Servlet API, is fully asynchronous and non-blocking, and implements the [Reactive Streams](#) specification through [the Reactor project](#).

Spring WebFlux comes in two flavors: functional and annotation-based. The annotation-based one is quite close to the Spring MVC model, as shown in the following example:

```
@RestController
@RequestMapping("/users")
public class MyRestController {

    @GetMapping("/{user}")
    public Mono<User> getUser(@PathVariable Long user) {
        // ...
    }

    @GetMapping("/{user}/customers")
    public Flux<Customer> getUserCustomers(@PathVariable Long user)
    {
        // ...
    }

    @DeleteMapping("/{user}")
    public Mono<User> deleteUser(@PathVariable Long user) {
        // ...
    }

}
```

“WebFlux.fn”, the functional variant, separates the routing configuration from the actual handling of the requests, as shown in the following example:

```
@Configuration
public class RoutingConfiguration {

    @Bean
    public RouterFunction<ServerResponse>
```

```
monoRouterFunction(UserHandler userHandler) {
    return
route(GET("/{user}").and(accept(APPLICATION_JSON)),
userHandler::getUser)

    .andRoute(GET("/{user}/customers").and(accept(APPLICATION_JSON))
), userHandler::getUserCustomers)

    .andRoute(DELETE("/{user}").and(accept(APPLICATION_JSON)),
userHandler::deleteUser);
}

}

@Component
public class UserHandler {

    public Mono<ServerResponse> getUser(ServerRequest request) {
        // ...
    }

    public Mono<ServerResponse> getUserCustomers(ServerRequest
request) {
        // ...
    }

    public Mono<ServerResponse> deleteUser(ServerRequest request) {
        // ...
    }
}
```

WebFlux is part of the Spring Framework and detailed information is available in its [reference documentation](#).



You can define as many **RouterFunction** beans as you like to modularize the definition of the router. Beans can be ordered if you need to apply a precedence.

To get started, add the `spring-boot-starter-webflux` module to your application.



Adding both `spring-boot-starter-web` and `spring-boot-starter-webflux` modules in your application results in Spring Boot auto-configuring Spring MVC, not WebFlux. This behavior has been chosen because many Spring developers add `spring-boot-starter-webflux` to their Spring MVC application to use the reactive `WebClient`. You can still enforce your choice by setting the chosen application type `toSpringApplication.setWebApplicationType(WebApplicationType.REACTIVE)`.

29.2.1 Spring WebFlux Auto-configuration

Spring Boot provides auto-configuration for Spring WebFlux that works well with most applications.

The auto-configuration adds the following features on top of Spring's defaults:

- Configuring codecs for `HttpMessageReader` and `HttpMessageWriter` instances (described [later in this document](#)).
- Support for serving static resources, including support for WebJars (described [later in this document](#)).

If you want to keep Spring Boot WebFlux features and you want to add additional [WebFlux configuration](#), you can add your own `@Configuration` class of type `WebFluxConfigurer` but **without** `@EnableWebFlux`.

If you want to take complete control of Spring WebFlux, you can add your own `@Configuration` annotated with `@EnableWebFlux`.

29.2.2 HTTP Codecs with HttpMessageReaders and HttpMessageWriters

Spring WebFlux uses the `HttpMessageReader` and `HttpMessageWriter` interfaces to convert HTTP requests and responses. They are configured with `CodecConfigurer` to have sensible defaults by looking at the libraries

available in your classpath.

Spring Boot applies further customization by using `CodecCustomizer` instances. For example, `spring.jackson.*` configuration keys are applied to the Jackson codec.

If you need to add or customize codecs, you can create a custom `CodecCustomizer` component, as shown in the following example:

```
import org.springframework.boot.web.codec.CodecCustomizer;

@Configuration
public class MyConfiguration {

    @Bean
    public CodecCustomizer myCodecCustomizer() {
        return codecConfigurer -> {
            // ...
        }
    }
}
```

You can also leverage [Boot's custom JSON serializers and deserializers](#).

29.2.3 Static Content

By default, Spring Boot serves static content from a directory called `/static` (or `/public` or `/resources` or `/META-INF/resources`) in the classpath. It uses the `ResourceWebHandler` from Spring WebFlux so that you can modify that behavior by adding your own `WebFluxConfigurer` and overriding the `addResourceHandlers` method.

By default, resources are mapped on `/**`, but you can tune that by setting the `spring.webflux.static-path-pattern` property. For instance, relocating all resources to `/resources/**` can be achieved as follows:

```
spring.webflux.static-path-pattern=/resources/**
```

You can also customize the static resource locations by using `spring.resources.static-locations`. Doing so replaces the default

values with a list of directory locations. If you do so, the default welcome page detection switches to your custom locations. So, if there is an `index.html` in any of your locations on startup, it is the home page of the application.

In addition to the “standard” static resource locations listed earlier, a special case is made for [Webjars content](#). Any resources with a path in `/webjars/**` are served from jar files if they are packaged in the Webjars format.



Spring WebFlux applications do not strictly depend on the Servlet API, so they cannot be deployed as war files and do not use the `src/main/webapp` directory.

29.2.4 Template Engines

As well as REST web services, you can also use Spring WebFlux to serve dynamic HTML content. Spring WebFlux supports a variety of templating technologies, including Thymeleaf, FreeMarker, and Mustache.

Spring Boot includes auto-configuration support for the following templating engines:

- [FreeMarker](#)
- [Thymeleaf](#)
- [Mustache](#)

When you use one of these templating engines with the default configuration, your templates are picked up automatically from `src/main/resources/templates`.

29.2.5 Error Handling

Spring Boot provides a `WebExceptionHandler` that handles all errors in a sensible way. Its position in the processing order is immediately before the handlers provided by WebFlux, which are considered last. For machine clients, it produces a JSON response with details of the error, the HTTP status, and the exception message. For browser clients, there is a “whitelabel” error handler that renders the same data in HTML format. You

can also provide your own HTML templates to display errors (see the [next section](#)).

The first step to customizing this feature often involves using the existing mechanism but replacing or augmenting the error contents. For that, you can add a bean of type `ErrorAttributes`.

To change the error handling behavior, you can implement `ErrorWebExceptionHandler` and register a bean definition of that type. Because a `WebExceptionHandler` is quite low-level, Spring Boot also provides a convenient `AbstractErrorWebExceptionHandler` to let you handle errors in a WebFlux functional way, as shown in the following example:

```
public class CustomErrorWebExceptionHandler extends AbstractErrorWebExceptionHandler {

    // Define constructor here

    @Override
    protected RouterFunction<ServerResponse>
    getRoutingFunction(ErrorAttributes errorAttributes) {

        return RouterFunctions
            .route(aPredicate, aHandler)
            .andRoute(anotherPredicate,
anotherHandler);
    }

}
```

For a more complete picture, you can also subclass `DefaultErrorWebExceptionHandler` directly and override specific methods.

Custom Error Pages

If you want to display a custom HTML error page for a given status code, you can add a file to an `/error` folder. Error pages can either be static HTML (that is, added under any of the static resource folders) or built with templates. The name of the file should be the exact status code or a series mask.

For example, to map `404` to a static HTML file, your folder structure would be as follows:

```
src/
+- main/
  +- java/
    |  + <source code>
  +- resources/
    +- public/
      +- error/
        |  +- 404.html
      +- <other public assets>
```

To map all `5xx` errors by using a Mustache template, your folder structure would be as follows:

```
src/
+- main/
  +- java/
    |  + <source code>
  +- resources/
    +- templates/
      +- error/
        |  +- 5xx.mustache
      +- <other templates>
```

29.2.6 Web Filters

Spring WebFlux provides a `WebFilter` interface that can be implemented to filter HTTP request-response exchanges. `WebFilter` beans found in the application context will be automatically used to filter each exchange.

Where the order of the filters is important they can implement `Ordered` or be annotated with `@Order`. Spring Boot auto-configuration may configure web filters for you. When it does so, the orders shown in the following table will be used:

Web Filter
MetricsWebFilter

`WebFilterChainProxy` (Spring Security)

-100

`HttpTraceWebFilter`

Ordered

29.3 JAX-RS and Jersey

If you prefer the JAX-RS programming model for REST endpoints, you can use one of the available implementations instead of Spring MVC. [Jersey](#) and [Apache CXF](#) work quite well out of the box. CXF requires you to register its `Servlet` or `Filter` as a `@Bean` in your application context. Jersey has some native Spring support, so we also provide auto-configuration support for it in Spring Boot, together with a starter.

To get started with Jersey, include the `spring-boot-starter-jersey` as a dependency and then you need one `@Bean` of type `ResourceConfig` in which you register all the endpoints, as shown in the following example:

```
@Component
public class JerseyConfig extends ResourceConfig {

    public JerseyConfig() {
        register(Endpoint.class);
    }

}
```



Jersey's support for scanning executable archives is rather limited. For example, it cannot scan for endpoints in a package found in a [fully executable jar file](#) or in `WEB-INF/classes` when running an executable war file. To avoid this limitation, the `packages` method should not be used, and endpoints should be registered individually by using the `register` method, as shown in the preceding example.

For more advanced customizations, you can also register an arbitrary number of beans that implement `ResourceConfigCustomizer`.

All the registered endpoints should be `@Components` with HTTP resource annotations (`@GET` and others), as shown in the following example:

```

@Component
@Path("/hello")
public class Endpoint {

    @GET
    public String message() {
        return "Hello";
    }

}

```

Since the `Endpoint` is a Spring `@Component`, its lifecycle is managed by Spring and you can use the `@Autowired` annotation to inject dependencies and use the `@Value` annotation to inject external configuration. By default, the Jersey servlet is registered and mapped to `/*`. You can change the mapping by adding `@ApplicationPath` to your `ResourceConfig`.

By default, Jersey is set up as a Servlet in a `@Bean` of type `ServletRegistrationBean` named `jerseyServletRegistration`. By default, the servlet is initialized lazily, but you can customize that behavior by setting `spring.jersey.servlet.load-on-startup`. You can disable or override that bean by creating one of your own with the same name. You can also use a filter instead of a servlet by setting `spring.jersey.type=filter` (in which case, the `@Bean` to replace or override is `jerseyFilterRegistration`). The filter has an `@Order`, which you can set with `spring.jersey.filter.order`. Both the servlet and the filter registrations can be given init parameters by using `spring.jersey.init.*` to specify a map of properties.

There is a [Jersey sample](#) so that you can see how to set things up.

29.4 Embedded Servlet Container Support

Spring Boot includes support for embedded [Tomcat](#), [Jetty](#), and [Undertow](#) servers. Most developers use the appropriate “Starter” to obtain a fully configured instance. By default, the embedded server listens for HTTP requests on port `8080`.

29.4.1 Servlets, Filters, and listeners

When using an embedded servlet container, you can register servlets,

filters, and all the listeners (such as `HttpSessionListener`) from the Servlet spec, either by using Spring beans or by scanning for Servlet components.

Registering Servlets, Filters, and Listeners as Spring Beans

Any `Servlet`, `Filter`, or servlet `*Listener` instance that is a Spring bean is registered with the embedded container. This can be particularly convenient if you want to refer to a value from your `application.properties` during configuration.

By default, if the context contains only a single Servlet, it is mapped to `/`. In the case of multiple servlet beans, the bean name is used as a path prefix. Filters map to `/*`.

If convention-based mapping is not flexible enough, you can use the `ServletRegistrationBean`, `FilterRegistrationBean`, and `ServletListenerRegistrationBean` classes for complete control.

Spring Boot ships with many auto-configurations that may define Filter beans. Here are a few examples of Filters and their respective order (lower order value means higher precedence):

Servlet Filter
<code>OrderedCharacterEncodingFilter</code>
<code>WebMvcMetricsFilter</code>
<code>ErrorPageFilter</code>
<code>HttpTraceFilter</code>

It is usually safe to leave Filter beans unordered.

If a specific order is required, you should avoid configuring a Filter that reads the request body at `Ordered.HIGHEST_PRECEDENCE`, since it might go against the character encoding configuration of your application. If a Servlet filter wraps the request, it should be configured with an order that

is less than or equal to `OrderedFilter.REQUEST_WRAPPER_FILTER_MAX_ORDER`.

29.4.2 Servlet Context Initialization

Embedded servlet containers do not directly execute the Servlet 3.0+ `javax.servlet.ServletContainerInitializer` interface or Spring's `org.springframework.web.WebApplicationInitializer` interface. This is an intentional design decision intended to reduce the risk that third party libraries designed to run inside a war may break Spring Boot applications.

If you need to perform servlet context initialization in a Spring Boot application, you should register a bean that implements the `org.springframework.boot.web.servlet.ServletContextInitializer` interface. The single `onStartup` method provides access to the `ServletContext` and, if necessary, can easily be used as an adapter to an existing `WebApplicationInitializer`.

Scanning for Servlets, Filters, and listeners

When using an embedded container, automatic registration of classes annotated with `@WebServlet`, `@WebFilter`, and `@WebListener` can be enabled by using `@ServletComponentScan`.



`@ServletComponentScan` has no effect in a standalone container, where the container's built-in discovery mechanisms are used instead.

29.4.3 The ServletWebServerApplicationContext

Under the hood, Spring Boot uses a different type of `ApplicationContext` for embedded servlet container support. The `ServletWebServerApplicationContext` is a special type of `WebApplicationContext` that bootstraps itself by searching for a single `ServletWebServerFactory` bean. Usually a `TomcatServletWebServerFactory`, `JettyServletWebServerFactory`, or `UndertowServletWebServerFactory` has been auto-configured.



You usually do not need to be aware of these implementation classes. Most applications are auto-configured, and the appropriate `ApplicationContext` and `ServletWebServerFactory` are created on your behalf.

29.4.4 Customizing Embedded Servlet Containers

Common servlet container settings can be configured by using Spring `Environment` properties. Usually, you would define the properties in your `application.properties` file.

Common server settings include:

- Network settings: Listen port for incoming HTTP requests (`server.port`), interface address to bind to `server.address`, and so on.
- Session settings: Whether the session is persistent (`server.servlet.session.persistence`), session timeout (`server.servlet.session.timeout`), location of session data (`server.servlet.session.store-dir`), and session-cookie configuration (`server.servlet.session.cookie.*`).
- Error management: Location of the error page (`server.error.path`) and so on.
- [SSL](#)
- [HTTP compression](#)

Spring Boot tries as much as possible to expose common settings, but this is not always possible. For those cases, dedicated namespaces offer server-specific customizations (see `server.tomcat` and `server.undertow`). For instance, [access logs](#) can be configured with specific features of the embedded servlet container.



See the `ServerProperties` class for a complete list.

Programmatic Customization

If you need to programmatically configure your embedded servlet container, you can register a Spring bean that implements the `WebServerFactoryCustomizer` interface. `WebServerFactoryCustomizer` provides access to the `ConfigurableServletWebServerFactory`, which includes numerous customization setter methods. The following example shows programmatically setting the port:

```
import org.springframework.boot.web.server.WebServerFactoryCustomizer;
import
org.springframework.boot.web.servlet.server.ConfigurableServletWebServerFactory;
import org.springframework.stereotype.Component;

@Component
public class CustomizationBean implements
WebServerFactoryCustomizer<ConfigurableServletWebServerFactory> {

    @Override
    public void customize(ConfigurableServletWebServerFactory
server) {
        server.setPort(9000);
    }
}
```



`TomcatServletWebServerFactory`, `JettyServletWebServerFactory` and `UndertowServletWebServerFactory` are dedicated variants of `ConfigurableServletWebServerFactory` that have additional customization setter methods for Tomcat, Jetty and Undertow respectively.

Customizing ConfigurableServletWebServerFactory Directly

If the preceding customization techniques are too limited, you can register the `TomcatServletWebServerFactory`, `JettyServletWebServerFactory`, or `UndertowServletWebServerFactory` bean yourself.

```
@Bean
public ConfigurableServletWebServerFactory webServerFactory() {
    TomcatServletWebServerFactory factory = new
```

```
TomcatServletWebServerFactory();
    factory.setPort(9000);
    factory.setSessionTimeout(10, TimeUnit.MINUTES);
    factory.addErrorPages(new ErrorPage(HttpStatus.NOT_FOUND,
"/notfound.html"));
    return factory;
}
```

Setters are provided for many configuration options. Several protected method “hooks” are also provided should you need to do something more exotic. See the [source code documentation](#) for details.

29.4.5 JSP Limitations

When running a Spring Boot application that uses an embedded servlet container (and is packaged as an executable archive), there are some limitations in the JSP support.

- With Jetty and Tomcat, it should work if you use war packaging. An executable war will work when launched with `java -jar`, and will also be deployable to any standard container. JSPs are not supported when using an executable jar.
- Undertow does not support JSPs.
- Creating a custom `error.jsp` page does not override the default view for [error handling](#). [Custom error pages](#) should be used instead.

There is a [JSP sample](#) so that you can see how to set things up.

29.5 Embedded Reactive Server Support

Spring Boot includes support for the following embedded reactive web servers: Reactor Netty, Tomcat, Jetty, and Undertow. Most developers use the appropriate “Starter” to obtain a fully configured instance. By default, the embedded server listens for HTTP requests on port 8080.

29.6 Reactive Server Resources Configuration

When auto-configuring a Reactor Netty or Jetty server, Spring Boot will create specific beans that will provide HTTP resources to the server instance: `ReactorResourceFactory` or `JettyResourceFactory`.

By default, those resources will be also shared with the Reactor Netty and Jetty clients for optimal performances, given:

- the same technology is used for server and client
- the client instance is built using the `WebClient.Builder` bean auto-configured by Spring Boot

Developers can override the resource configuration for Jetty and Reactor Netty by providing a custom `ReactorResourceFactory` or `JettyResourceFactory` bean - this will be applied to both clients and servers.

You can learn more about the resource configuration on the client side in the [WebClient Runtime section](#).

30. Security

If [Spring Security](#) is on the classpath, then web applications are secured by default. Spring Boot relies on Spring Security's content-negotiation strategy to determine whether to use `httpBasic` or `formLogin`. To add method-level security to a web application, you can also add `@EnableGlobalMethodSecurity` with your desired settings. Additional information can be found in the [Spring Security Reference Guide](#).

The default `UserDetailsService` has a single user. The user name is `user`, and the password is random and is printed at INFO level when the application starts, as shown in the following example:

```
Using generated security password: 78fa095d-3f4c-48b1-ad50-e24c31d5cf35
```



If you fine-tune your logging configuration, ensure that the `org.springframework.boot.autoconfigure.security` category is set to log `INFO`-level messages. Otherwise, the default password is not printed.

You can change the username and password by providing a `spring.security.user.name` and `spring.security.user.password`.

The basic features you get by default in a web application are:

- A `UserDetailsService` (or `ReactiveUserDetailsService` in case of a WebFlux application) bean with in-memory store and a single user with a generated password (see `SecurityProperties.User` for the properties of the user).
- Form-based login or HTTP Basic security (depending on the `Accept` header in the request) for the entire application (including actuator endpoints if actuator is on the classpath).
- A `DefaultAuthenticationEventPublisher` for publishing authentication events.

You can provide a different `AuthenticationEventPublisher` by adding a bean for it.

30.1 MVC Security

The default security configuration is implemented in `SecurityAutoConfiguration` and `UserDetailsServiceAutoConfiguration`. `SecurityAutoConfiguration` imports `SpringBootWebSecurityConfiguration` for web security

and `UserDetailsServiceAutoConfiguration` configures authentication, which is also relevant in non-web applications. To switch off the default web application security configuration completely, you can add a bean of type `WebSecurityConfigurerAdapter` (doing so does not disable the `UserDetailsService` configuration or Actuator's security).

To also switch off the `UserDetailsService` configuration, you can add a bean of type `UserDetailsService`, `AuthenticationProvider`, or `AuthenticationManager`. There are several secure applications in the [Spring Boot samples](#) to get you started with common use cases.

Access rules can be overridden by adding a custom `WebSecurityConfigurerAdapter`. Spring Boot provides convenience methods that can be used to override access rules for actuator endpoints and static resources. `EndpointRequest` can be used to create a `RequestMatcher` that is based on the `management.endpoints.web.base-path` property. `PathRequest` can be used to create a `RequestMatcher` for resources in commonly used

locations.

30.2 WebFlux Security

Similar to Spring MVC applications, you can secure your WebFlux applications by adding the `spring-boot-starter-security` dependency.

The default security configuration is implemented in `ReactiveSecurityAutoConfiguration` and `UserDetailsServiceAutoConfiguration`. `ReactiveSecurityAutoConfiguration` imports `WebFluxSecurityConfiguration` for web security

and `UserDetailsServiceAutoConfiguration` configures authentication, which is also relevant in non-web applications. To switch off the default web application security configuration completely, you can add a bean of type `WebFilterChainProxy` (doing so does not disable the `UserDetailsService` configuration or Actuator's security).

To also switch off the `UserDetailsService` configuration, you can add a bean of

type `ReactiveUserDetailsService` or `ReactiveAuthenticationManager`.

Access rules can be configured by adding a custom `SecurityWebFilterChain`. Spring Boot provides convenience methods that can be used to override access rules for actuator endpoints and static resources. `EndpointRequest` can be used to create a `ServerWebExchangeMatcher` that is based on the `management.endpoints.web.base-path` property.

`PathRequest` can be used to create a `ServerWebExchangeMatcher` for resources in commonly used locations.

For example, you can customize your security configuration by adding something like:

```
@Bean
public SecurityWebFilterChain
springSecurityFilterChain(ServerHttpSecurity http) {
    return http
        .authorizeExchange()
            .matchers(PathRequest.toStaticResources().atCommonLocations())
        .permitAll()
```

```
        .pathMatchers("/foo", "/bar")
            .authenticated().and()
        .formLogin().and()
    .build();
}
```

30.3 OAuth2

[OAuth2](#) is a widely used authorization framework that is supported by Spring.

30.3.1 Client

If you have `spring-security-oauth2-client` on your classpath, you can take advantage of some auto-configuration to make it easy to set up an OAuth2/Open ID Connect clients. This configuration makes use of the properties under `OAuth2ClientProperties`. The same properties are applicable to both servlet and reactive applications.

You can register multiple OAuth2 clients and providers under the `spring.security.oauth2.client` prefix, as shown in the following example:

```
spring.security.oauth2.client.registration.my-client-1.client-id=abcd
spring.security.oauth2.client.registration.my-client-1.client-
secret=password
spring.security.oauth2.client.registration.my-client-1.client-
name=Client for user scope
spring.security.oauth2.client.registration.my-client-1.provider=my-
oauth-provider
spring.security.oauth2.client.registration.my-client-1.scope=user
spring.security.oauth2.client.registration.my-client-1.redirect-uri-
template=https://my-redirect-uri.com
spring.security.oauth2.client.registration.my-client-1.client-
authentication-method=basic
spring.security.oauth2.client.registration.my-client-1.authorization-
grant-type=authorization_code

spring.security.oauth2.client.registration.my-client-2.client-id=abcd
spring.security.oauth2.client.registration.my-client-2.client-
secret=password
spring.security.oauth2.client.registration.my-client-2.client-
name=Client for email scope
spring.security.oauth2.client.registration.my-client-2.provider=my-
oauth-provider
```

```
spring.security.oauth2.client.registration.my-client-2.scope=email
spring.security.oauth2.client.registration.my-client-2.redirect-uri-
template=https://my-redirect-uri.com
spring.security.oauth2.client.registration.my-client-2.client-
authentication-method=basic
spring.security.oauth2.client.registration.my-client-2.authorization-
grant-type=authorization_code

spring.security.oauth2.client.provider.my-oauth-
provider.authorization-uri=http://my-auth-server/oauth/authorize
spring.security.oauth2.client.provider.my-oauth-provider.token-
uri=http://my-auth-server/oauth/token
spring.security.oauth2.client.provider.my-oauth-provider.user-info-
uri=http://my-auth-server/userinfo
spring.security.oauth2.client.provider.my-oauth-provider.user-info-
authentication-method=header
spring.security.oauth2.client.provider.my-oauth-provider.jwk-set-
uri=http://my-auth-server/token_keys
spring.security.oauth2.client.provider.my-oauth-provider.user-name-
attribute=name
```

For OpenID Connect providers that support [OpenID Connect discovery](#), the configuration can be further simplified. The provider needs to be configured with an `issuer-uri` which is the URI that it asserts as its Issuer Identifier. For example, if the `issuer-uri` provided is "https://example.com", then an [OpenID Provider Configuration Request](#) will be made to "https://example.com/.well-known/openid-configuration". The result is expected to be an [OpenID Provider Configuration Response](#). The following example shows how an OpenID Connect Provider can be configured with the `issuer-uri`:

```
spring.security.oauth2.client.provider.oidc-provider.issuer-
uri=https://dev-123456.oktapreview.com/oauth2/default/
```

By default, Spring Security's `OAuth2LoginAuthenticationFilter` only processes URLs matching `/login/oauth2/code/*`. If you want to customize the `redirect-uri` to use a different pattern, you need to provide configuration to process that custom pattern. For example, for servlet applications, you can add your own `WebSecurityConfigurerAdapter` that resembles the following:

```
public class OAuth2LoginSecurityConfig extends
```

```
WebSecurityConfigurerAdapter {  
  
    @Override  
    protected void configure(HttpSecurity http) throws Exception {  
        http  
            .authorizeRequests()  
                .anyRequest().authenticated()  
                .and()  
            .oauth2Login()  
                . redirectionEndpoint()  
                    .baseUri("/custom-callback");  
    }  
}
```

OAuth2 client registration for common providers

For common OAuth2 and OpenID providers, including Google, Github, Facebook, and Okta, we provide a set of provider defaults (`google`, `github`, `facebook`, and `okta`, respectively).

If you do not need to customize these providers, you can set the `provider` attribute to the one for which you need to infer defaults. Also, if the key for the client registration matches a default supported provider, Spring Boot infers that as well.

In other words, the two configurations in the following example use the Google provider:

```
spring.security.oauth2.client.registration.my-client.client-id=abcd  
spring.security.oauth2.client.registration.my-client.client-  
secret=password  
spring.security.oauth2.client.registration.my-client.provider=google  
  
spring.security.oauth2.client.registration.google.client-id=abcd  
spring.security.oauth2.client.registration.google.client-  
secret=password
```

30.3.2 Resource Server

If you have `spring-security-oauth2-resource-server` on your classpath, Spring Boot can set up an OAuth2 Resource Server as long as a JWK Set URI or OIDC Issuer URI is specified, as shown in the following examples:

```
spring.security.oauth2.resourcereserver.jwt.jwk-set-  
uri=https://example.com/oauth2/default/v1/keys
```

```
spring.security.oauth2.resourceserver.jwt.issuer-uri=https://dev-123456.oktapreview.com/oauth2/default/
```

The same properties are applicable for both servlet and reactive applications.

Alternatively, you can define your own `JwtDecoder` bean for servlet applications or a `ReactiveJwtDecoder` for reactive applications.

30.3.3 Authorization Server

Currently, Spring Security does not provide support for implementing an OAuth 2.0 Authorization Server. However, this functionality is available from the [Spring Security OAuth](#) project, which will eventually be superseded by Spring Security completely. Until then, you can use the `spring-security-oauth2-autoconfigure` module to easily set up an OAuth 2.0 authorization server; see its [documentation](#) for instructions.

30.4 Actuator Security

For security purposes, all actuators other than `/health` and `/info` are disabled by default.

The `management.endpoints.web.exposure.include` property can be used to enable the actuators.

If Spring Security is on the classpath and no other `WebSecurityConfigurerAdapter` is present, all actuators other than `/health` and `/info` are secured by Spring Boot auto-configuration. If you define a custom `WebSecurityConfigurerAdapter`, Spring Boot auto-configuration will back off and you will be in full control of actuator access rules.



Before setting the `management.endpoints.web.exposure.include`, ensure that the exposed actuators do not contain sensitive information and/or are secured by placing them behind a firewall or by something like Spring Security.

30.4.1 Cross Site Request Forgery Protection

Since Spring Boot relies on Spring Security's defaults, CSRF protection is turned on by default. This means that the actuator endpoints that require a `POST` (shutdown and loggers endpoints), `PUT` or `DELETE` will get a 403 forbidden error when the default security configuration is in use.

We recommend disabling CSRF protection completely only if you are creating a service that is used by non-browser clients.

Additional information about CSRF protection can be found in the [Spring Security Reference Guide](#).

31. Working with SQL Databases

The [Spring Framework](#) provides extensive support for working with SQL databases, from direct JDBC access using `JdbcTemplate` to complete “object relational mapping” technologies such as Hibernate. [Spring Data](#) provides an additional level of functionality: creating `Repository` implementations directly from interfaces and using conventions to generate queries from your method names.

31.1 Configure a DataSource

Java's `javax.sql.DataSource` interface provides a standard method of working with database connections. Traditionally, a 'DataSource' uses a `URL` along with some credentials to establish a database connection.



See [the “How-to” section](#) for more advanced examples, typically to take full control over the configuration of the `DataSource`.

31.1.1 Embedded Database Support

It is often convenient to develop applications by using an in-memory

embedded database. Obviously, in-memory databases do not provide persistent storage. You need to populate your database when your application starts and be prepared to throw away data when your application ends.



The “How-to” section includes a [section on how to initialize a database](#).

Spring Boot can auto-configure embedded [H2](#), [HSQL](#), and [Derby](#) databases. You need not provide any connection URLs. You need only include a build dependency to the embedded database that you want to use.



If you are using this feature in your tests, you may notice that the same database is reused by your whole test suite regardless of the number of application contexts that you use. If you want to make sure that each context has a separate embedded database, you should set `spring.datasource.generate-unique-name` to `true`.

For example, the typical POM dependencies would be as follows:

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



You need a dependency on `spring-idbc` for an embedded database to be

auto-configured. In this example, it is pulled in transitively through `spring-boot-starter-data-jpa`.



If, for whatever reason, you do configure the connection URL for an embedded database, take care to ensure that the database's automatic shutdown is disabled. If you use H2, you should use `DB_CLOSE_ON_EXIT=FALSE` to do so. If you use HSQLDB, you should ensure that `shutdown=true` is not used. Disabling the database's automatic shutdown lets Spring Boot control when the database is closed, thereby ensuring that it happens once access to the database is no longer needed.

31.1.2 Connection to a Production Database

Production database connections can also be auto-configured by using a pooling `DataSource`. Spring Boot uses the following algorithm for choosing a specific implementation:

1. We prefer [HikariCP](#) for its performance and concurrency. If HikariCP is available, we always choose it.
2. Otherwise, if the Tomcat pooling `DataSource` is available, we use it.
3. If neither HikariCP nor the Tomcat pooling datasource are available and if [Commons DBCP2](#) is available, we use it.

If you use the `spring-boot-starter-jdbc` or `spring-boot-starter-data-jpa` “starters”, you automatically get a dependency to [HikariCP](#).



You can bypass that algorithm completely and specify the connection pool to use by setting the `spring.datasource.type` property. This is especially important if you run your application in a Tomcat container, as `tomcat-jdbc` is provided by default.



Additional connection pools can always be configured manually. If you define your own `DataSource` bean, auto-configuration does not occur.

DataSource configuration is controlled by external configuration properties in `spring.datasource.*`. For example, you might declare the following section in `application.properties`:

```
spring.datasource.url=jdbc:mysql://localhost/test
spring.datasource.username=dbuser
spring.datasource.password=dbpass
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```



You should at least specify the URL by setting the `spring.datasource.url` property. Otherwise, Spring Boot tries to auto-configure an embedded database.



You often do not need to specify the `driver-class-name`, since Spring Boot can deduce it for most databases from the `url`.



For a pooling **DataSource** to be created, we need to be able to verify that a valid **Driver** class is available, so we check for that before doing anything. In other words, if you set `spring.datasource.driver-class-name=com.mysql.jdbc.Driver`, then that class has to be loadable.

See [DataSourceProperties](#) for more of the supported options. These are the standard options that work regardless of the actual implementation. It is also possible to fine-tune implementation-specific settings by using their respective prefix

(`spring.datasource.hikari.*`, `spring.datasource.tomcat.*`, and `spring.datasource.dbcp2.*`). Refer to the documentation of the connection pool implementation you are using for more details.

For instance, if you use the [Tomcat connection pool](#), you could customize many additional settings, as shown in the following example:

```
# Number of ms to wait before throwing an exception if no connection  
# is available.  
spring.datasource.tomcat.max-wait=10000  
  
# Maximum number of active connections that can be allocated from this  
# pool at the same time.  
spring.datasource.tomcat.max-active=50  
  
# Validate the connection before borrowing it from the pool.  
spring.datasource.tomcat.test-on-borrow=true
```

31.1.3 Connection to a JNDI DataSource

If you deploy your Spring Boot application to an Application Server, you might want to configure and manage your DataSource by using your Application Server's built-in features and access it by using JNDI.

The `spring.datasource.jndi-name` property can be used as an alternative to the `spring.datasource.url`, `spring.datasource.username`, and `spring.datasource.password` properties to access the `DataSource` from a specific JNDI location. For example, the following section in `application.properties` shows how you can access a JBoss AS defined `DataSource`:

```
spring.datasource.jndi-name=java:jboss/datasources/customers
```

31.2 Using JdbcTemplate

Spring's `JdbcTemplate` and `NamedParameterJdbcTemplate` classes are auto-configured, and you can `@Autowire` them directly into your own beans, as shown in the following example:

```
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.jdbc.core.JdbcTemplate;  
import org.springframework.stereotype.Component;  
  
@Component  
public class MyBean {  
  
    private final JdbcTemplate jdbcTemplate;  
  
    @Autowired  
    public MyBean(JdbcTemplate jdbcTemplate) {  
        this.jdbcTemplate = jdbcTemplate;
```

```
    }  
  
    // ...  
  
}
```

You can customize some properties of the template by using the `spring.jdbc.template.*` properties, as shown in the following example:

```
spring.jdbc.template.max-rows=500
```



The `NamedParameterJdbcTemplate` reuses the same `JdbcTemplate` instance behind the scenes. If more than one `JdbcTemplate` is defined and no primary candidate exists, the `NamedParameterJdbcTemplate` is not auto-configured.

31.3 JPA and Spring Data JPA

The Java Persistence API is a standard technology that lets you “map” objects to relational databases. The `spring-boot-starter-data-jpa` POM provides a quick way to get started. It provides the following key dependencies:

- Hibernate: One of the most popular JPA implementations.
- Spring Data JPA: Makes it easy to implement JPA-based repositories.
- Spring ORMs: Core ORM support from the Spring Framework.



We do not go into too many details of JPA or [Spring Data](#) here. You can follow the “[Accessing Data with JPA](#)” guide from [spring.io](#) and read the [Spring Data JPA](#) and [Hibernate](#) reference documentation.

31.3.1 Entity Classes

Traditionally, JPA “Entity” classes are specified in a `persistence.xml` file.

With Spring Boot, this file is not necessary and “Entity Scanning” is used instead. By default, all packages below your main configuration class (the one annotated

with `@EnableAutoConfiguration` or `@SpringBootApplication`) are searched.

Any classes annotated with `@Entity`, `@Embeddable`, or `@MappedSuperclass` are considered. A typical entity class resembles the following example:

```
package com.example.myapp.domain;

import java.io.Serializable;
import javax.persistence.*;

@Entity
public class City implements Serializable {

    @Id
    @GeneratedValue
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false)
    private String state;

    // ... additional members, often include @OneToMany mappings

    protected City() {
        // no-args constructor required by JPA spec
        // this one is protected since it shouldn't be used
        // directly
    }

    public City(String name, String state) {
        this.name = name;
        this.state = state;
    }

    public String getName() {
        return this.name;
    }
}
```

```
    public String getState() {
        return this.state;
    }

    // ... etc
}
```



You can customize entity scanning locations by using the `@EntityScan` annotation. See the “[Section 84.4, “Separate @Entity Definitions from Spring Configuration”](#)” how-to.

31.3.2 Spring Data JPA Repositories

[Spring Data JPA](#) repositories are interfaces that you can define to access data. JPA queries are created automatically from your method names. For example, a `CityRepository` interface might declare a `findAllByState(String state)` method to find all the cities in a given state.

For more complex queries, you can annotate your method with Spring Data’s `Query` annotation.

Spring Data repositories usually extend from the `Repository` or `CrudRepository` interfaces. If you use auto-configuration, repositories are searched from the package containing your main configuration class (the one annotated with `@EnableAutoConfiguration` or `@SpringBootApplication`) down.

The following example shows a typical Spring Data repository interface definition:

```
package com.example.myapp.domain;

import org.springframework.data.domain.*;
import org.springframework.data.repository.*;

public interface CityRepository extends Repository<City, Long> {

    Page<City> findAll(Pageable pageable);
```

```
        City findByNameAndStateAllIgnoringCase(String name, String
state);  
    }
```

Spring Data JPA repositories support three different modes of bootstrapping: default, deferred, and lazy. To enable deferred or lazy bootstrapping, set the `spring.data.jpa.repositories.bootstrap-mode` to `deferred` or `lazy` respectively. When using deferred or lazy bootstrapping, the auto-configured `EntityManagerFactoryBuilder` will use the context's `AsyncTaskExecutor`, if any, as the bootstrap executor. If more than one exists, the one named `applicationTaskExecutor` will be used.



We have barely scratched the surface of Spring Data JPA. For complete details, see the [Spring Data JPA reference documentation](#).

31.3.3 Creating and Dropping JPA Databases

By default, JPA databases are automatically created **only** if you use an embedded database (H2, HSQL, or Derby). You can explicitly configure JPA settings by using `spring.jpa.*` properties. For example, to create and drop tables you can add the following line to your `application.properties`:

```
spring.jpa.hibernate.ddl-auto=create-drop
```



Hibernate's own internal property name for this (if you happen to remember it better) is `hibernate.hbm2ddl.auto`. You can set it, along with other Hibernate native properties, by using `spring.jpa.properties.*` (the prefix is stripped before adding them to the entity manager). The following line shows an example of setting JPA properties for Hibernate:

```
spring.jpa.properties.hibernate.globally_quoted_identifiers=true
```

The line in the preceding example passes a value of `true` for

the `hibernate.globally_quoted_identifiers` property to the Hibernate entity manager.

By default, the DDL execution (or validation) is deferred until the `ApplicationContext` has started. There is also a `spring.jpa.generate-ddl` flag, but it is not used if Hibernate auto-configuration is active, because the `ddl-auto` settings are more fine-grained.

31.3.4 Open EntityManager in View

If you are running a web application, Spring Boot by default registers `OpenEntityManagerInViewInterceptor` to apply the “Open EntityManager in View” pattern, to allow for lazy loading in web views. If you do not want this behavior, you should set `spring.jpa.open-in-view` to `false` in your `application.properties`.

31.4 Spring Data JDBC

Spring Data includes repository support for JDBC and will automatically generate SQL for the methods on `CrudRepository`. For more advanced queries, a `@Query` annotation is provided.

Spring Boot will auto-configure Spring Data’s JDBC repositories when the necessary dependencies are on the classpath. They can be added to your project with a single dependency on `spring-boot-starter-data-jdbc`. If necessary, you can take control of Spring Data JDBC’s configuration by adding the `@EnableJdbcRepositories` annotation or a `JdbcConfiguration` subclass to your application.



For complete details of Spring Data JDBC, please refer to the [reference documentation](#).

31.5 Using H2’s Web Console

The [H2 database](#) provides a [browser-based console](#) that Spring Boot can auto-configure for you. The console is auto-configured when the following conditions are met:

- You are developing a servlet-based web application.
- `com.h2database:h2` is on the classpath.
- You are using [Spring Boot's developer tools](#).



If you are not using Spring Boot's developer tools but would still like to make use of H2's console, you can configure the `spring.h2.console.enabled` property with a value of `true`.



The H2 console is only intended for use during development, so you should take care to ensure that `spring.h2.console.enabled` is not set to `true` in production.

31.5.1 Changing the H2 Console's Path

By default, the console is available at `/h2-console`. You can customize the console's path by using the `spring.h2.console.path` property.

31.6 Using jOOQ

Java Object Oriented Querying ([jOOQ](#)) is a popular product from [Data Geekery](#) which generates Java code from your database and lets you build type-safe SQL queries through its fluent API. Both the commercial and open source editions can be used with Spring Boot.

31.6.1 Code Generation

In order to use jOOQ type-safe queries, you need to generate Java classes from your database schema. You can follow the instructions in the [jOOQ user manual](#). If you use the `jooq-codegen-maven` plugin and you also use the `spring-boot-starter-parent` “parent POM”, you can safely omit the plugin's `<version>` tag. You can also use Spring Boot-defined version variables (such as `h2.version`) to declare the plugin's database dependency. The following listing shows an example:

```
<plugin>
  <groupId>org.jooq</groupId>
```

```
<artifactId>jooq-codegen-maven</artifactId>
<executions>
    ...
</executions>
<dependencies>
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <version>${h2.version}</version>
    </dependency>
</dependencies>
<configuration>
    <jdbc>
        <driver>org.h2.Driver</driver>
        <url>jdbc:h2:~/yourdatabase</url>
    </jdbc>
    <generator>
        ...
    </generator>
</configuration>
</plugin>
```

31.6.2 Using DSLContext

The fluent API offered by jOOQ is initiated through the `org.jooq.DSLContext` interface. Spring Boot auto-configures a `DSLContext` as a Spring Bean and connects it to your application `DataSource`. To use the `DSLContext`, you can `@Autowire` it, as shown in the following example:

```
@Component
public class JooqExample implements CommandLineRunner {

    private final DSLContext create;

    @Autowired
    public JooqExample(DSLContext dslContext) {
        this.create = dslContext;
    }

}
```



The jOOQ manual tends to use a variable named `create` to hold the `DSLContext`.

You can then use the `DSLContext` to construct your queries, as shown in the following example:

```
public List<GregorianCalendar> authorsBornAfter1980() {  
    return this.create.selectFrom(AUTHOR)  
        .where(AUTHOR.DATE_OF_BIRTH.greaterThan(new  
GregorianCalendar(1980, 0, 1)))  
        .fetch(AUTHOR.DATE_OF_BIRTH);  
}
```

31.6.3 jOOQ SQL Dialect

Unless the `spring.jooq.sql-dialect` property has been configured, Spring Boot determines the SQL dialect to use for your datasource. If Spring Boot could not detect the dialect, it uses `DEFAULT`.



Spring Boot can only auto-configure dialects supported by the open source version of jOOQ.

31.6.4 Customizing jOOQ

More advanced customizations can be achieved by defining your own `@Bean` definitions, which is used when the jOOQ `Configuration` is created. You can define beans for the following jOOQ Types:

- `ConnectionProvider`
- `ExecutorProvider`
- `TransactionProvider`
- `RecordMapperProvider`
- `RecordUnmapperProvider`
- `RecordListenerProvider`
- `ExecuteListenerProvider`
- `VisitListenerProvider`
- `TransactionListenerProvider`

You can also create your own `org.jooq.Configuration` `@Bean` if you want to take complete control of the jOOQ configuration.

32. Working with NoSQL Technologies

Spring Data provides additional projects that help you access a variety of NoSQL technologies, including: [MongoDB](#), [Neo4J](#), [Elasticsearch](#), [Solr](#), [Redis](#), [Gemfire](#), [Cassandra](#), [Couchbase](#) and [LDAP](#). Spring Boot provides auto-configuration for Redis, MongoDB, Neo4j, Elasticsearch, Solr Cassandra, Couchbase, and LDAP. You can make use of the other projects, but you must configure them yourself. Refer to the appropriate reference documentation at [projects.spring.io/spring-data](#).

32.1 Redis

[Redis](#) is a cache, message broker, and richly-featured key-value store. Spring Boot offers basic auto-configuration for the [Lettuce](#) and [Jedis](#) client libraries and the abstractions on top of them provided by [Spring Data Redis](#).

There is a `spring-boot-starter-data-redis` “Starter” for collecting the dependencies in a convenient way. By default, it uses [Lettuce](#). That starter handles both traditional and reactive applications.



we also provide a `spring-boot-starter-data-redis-reactive` “Starter” for consistency with the other stores with reactive support.

32.1.1 Connecting to Redis

You can inject an auto-configured `RedisConnectionFactory`, `StringRedisTemplate`, or vanilla `RedisTemplate` instance as you would any other Spring Bean. By default, the instance tries to connect to a Redis server at `localhost:6379`. The following listing shows an example of such a bean:

```
@Component
public class MyBean {

    private StringRedisTemplate template;

    @Autowired
    public MyBean(StringRedisTemplate template) {
        this.template = template;
    }

    // ...
}
```



You can also register an arbitrary number of beans that implement `LettuceClientConfigurationBuilderCustomizer` for more advanced customizations. If you use `Jedis`, `JedisClientConfigurationBuilderCustomizer` is also available.

If you add your own `@Bean` of any of the auto-configured types, it replaces the default (except in the case of `RedisTemplate`, when the exclusion is based on the bean name, `redisTemplate`, not its type). By default, if `commons-pool2` is on the classpath, you get a pooled connection factory.

32.2 MongoDB

[MongoDB](#) is an open-source NoSQL document database that uses a JSON-like schema instead of traditional table-based relational data. Spring Boot offers several conveniences for working with MongoDB, including the `spring-boot-starter-data-mongodb` and `spring-boot-starter-data-mongodb-reactive` “Starters”.

32.2.1 Connecting to a MongoDB Database

To access Mongo databases, you can inject an auto-configured `org.springframework.data.mongodb.MongoDbFactory`. By default, the instance tries to connect to a MongoDB server at `mongodb://localhost/test`. The following example shows how to connect to a MongoDB database:

```
import org.springframework.data.mongodb.MongoDbFactory;
import com.mongodb.DB;

@Component
public class MyBean {

    private final MongoDbFactory mongo;

    @Autowired
    public MyBean(MongoDbFactory mongo) {
        this.mongo = mongo;
    }

    // ...

    public void example() {
        DB db = mongo.getDb();
        // ...
    }
}
```

You can set the `spring.data.mongodb.uri` property to change the URL and configure additional settings such as the *replica set*, as shown in the following example:

```
spring.data.mongodb.uri=mongodb://user:secret@mongo1.example.com:12345
,mongo2.example.com:23456/test
```

Alternatively, as long as you use Mongo 2.x, you can specify a `host/port`. For example, you might declare the following settings in your `application.properties`:

```
spring.data.mongodb.host=mongoserver
spring.data.mongodb.port=27017
```

If you have defined your own `MongoClient`, it will be used to auto-configure a suitable `MongoDbFactory`. Both `com.mongodb.MongoClient` and `com.mongodb.client.MongoClient` are supported.



If you use the Mongo 3.0 Java driver, `spring.data.mongodb.host` and `spring.data.mongodb.port` are not supported. In such cases, `spring.data.mongodb.uri` should be used to provide all of the configuration.



If `spring.data.mongodb.port` is not specified, the default of `27017` is used. You could delete this line from the example shown earlier.



If you do not use Spring Data Mongo, you can inject `com.mongodb.MongoClient` beans instead of using `MongoDbFactory`. If you want to take complete control of establishing the MongoDB connection, you can also declare your own `MongoDbFactory` or `MongoClient` bean.



If you are using the reactive driver, Netty is required for SSL. The auto-configuration configures this factory automatically if Netty is available and the factory to use hasn't been customized already.

32.2.2 MongoTemplate

[Spring Data MongoDB](#) provides a `MongoTemplate` class that is very similar in its design to Spring's `JdbcTemplate`. As with `JdbcTemplate`, Spring Boot auto-configures a bean for you to inject the template, as follows:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {
```

```
private final MongoTemplate mongoTemplate;

@Autowired
public MyBean(MongoTemplate mongoTemplate) {
    this.mongoTemplate = mongoTemplate;
}

// ...

}
```

See the [MongoOperations Javadoc](#) for complete details.

32.2.3 Spring Data MongoDB Repositories

Spring Data includes repository support for MongoDB. As with the JPA repositories discussed earlier, the basic principle is that queries are constructed automatically, based on method names.

In fact, both Spring Data JPA and Spring Data MongoDB share the same common infrastructure. You could take the JPA example from earlier and, assuming that `City` is now a Mongo data class rather than a JPA `@Entity`, it works in the same way, as shown in the following example:

```
package com.example.myapp.domain;

import org.springframework.data.domain.*;
import org.springframework.data.repository.*;

public interface CityRepository extends Repository<City, Long> {

    Page<City> findAll(Pageable pageable);

    City findByNameAndStateAllIgnoringCase(String name, String
state);

}
```



You can customize document scanning locations by using the `@EntityScan` annotation.



For complete details of Spring Data MongoDB, including its rich object mapping technologies, refer to its [reference documentation](#).

32.2.4 Embedded Mongo

Spring Boot offers auto-configuration for [Embedded Mongo](#). To use it in your Spring Boot application, add a dependency on `de.flapdoodle.embed:de.flapdoodle.embed.mongo`.

The port that Mongo listens on can be configured by setting the `spring.data.mongodb.port` property. To use a randomly allocated free port, use a value of 0. The `MongoClient` created by `MongoAutoConfiguration` is automatically configured to use the randomly allocated port.



If you do not configure a custom port, the embedded support uses a random port (rather than 27017) by default.

If you have SLF4J on the classpath, the output produced by Mongo is automatically routed to a logger named `org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongo`.

You can declare your own `IMongodConfig` and `IRuntimeConfig` beans to take control of the Mongo instance's configuration and logging routing.

32.3 Neo4j

[Neo4j](#) is an open-source NoSQL graph database that uses a rich data model of nodes connected by first class relationships, which is better suited for connected big data than traditional RDBMS approaches. Spring Boot offers several conveniences for working with Neo4j, including the `spring-boot-starter-data-neo4j` “Starter”.

32.3.1 Connecting to a Neo4j Database

To access a Neo4j server, you can inject an auto-configured `org.neo4j.ogm.session.Session`. By default, the instance tries to connect to a Neo4j server at `localhost:7687` using the Bolt protocol. The following example shows how to inject a Neo4j `Session`:

```
@Component
public class MyBean {

    private final Session session;

    @Autowired
    public MyBean(Session session) {
        this.session = session;
    }

    // ...
}
```

You can configure the uri and credentials to use by setting the `spring.data.neo4j.*` properties, as shown in the following example:

```
spring.data.neo4j.uri=bolt://my-server:7687
spring.data.neo4j.username=neo4j
spring.data.neo4j.password=secret
```

You can take full control over the session creation by adding a `org.neo4j.ogm.config.Configuration` `@Bean`. Also, adding a `@Bean` of type `SessionFactory` disables the auto-configuration and gives you full control.

32.3.2 Using the Embedded Mode

If you add `org.neo4j:neo4j-ogm-embedded-driver` to the dependencies of your application, Spring Boot automatically configures an in-process embedded instance of Neo4j that does not persist any data when your application shuts down.



As the embedded Neo4j OGM driver does not provide the Neo4j kernel itself, you have to declare `org.neo4j:neo4j` as dependency yourself. Refer to [the Neo4j OGM documentation](#) for a list of compatible versions.

The embedded driver takes precedence over the other drivers when there are multiple drivers on the classpath. You can explicitly disable the embedded mode by setting `spring.data.neo4j.embedded.enabled=false`.

[Data Neo4j Tests](#) automatically make use of an embedded Neo4j instance if the embedded driver and Neo4j kernel are on the classpath as described above.



You can enable persistence for the embedded mode by providing a path to a database file in your configuration,
e.g. `spring.data.neo4j.uri=file://var/tmp/graph.db`.

32.3.3 Neo4jSession

By default, if you are running a web application, the session is bound to the thread for the entire processing of the request (that is, it uses the "Open Session in View" pattern). If you do not want this behavior, add the following line to your `application.properties` file:

```
spring.data.neo4j.open-in-view=false
```

32.3.4 Spring Data Neo4j Repositories

Spring Data includes repository support for Neo4j.

Spring Data Neo4j shares the common infrastructure with Spring Data JPA as many other Spring Data modules do. You could take the JPA example from earlier and define `City` as Neo4j OGM `@NodeEntity` rather than JPA `@Entity` and the repository abstraction works in the same way, as shown in the following example:

```
package com.example.myapp.domain;  
  
import java.util.Optional;
```

```
import org.springframework.data.neo4j.repository.*;  
  
public interface CityRepository extends Neo4jRepository<City, Long> {  
  
    Optional<City> findOneByNameAndState(String name, String  
state);  
}
```

The `spring-boot-starter-data-neo4j` “Starter” enables the repository support as well as transaction management. You can customize the locations to look for repositories and entities by using `@EnableNeo4jRepositories` and `@EntityScan` respectively on a `@Configuration`-bean.



For complete details of Spring Data Neo4j, including its object mapping technologies, refer to the [reference documentation](#).

32.4 Gemfire

[Spring Data Gemfire](#) provides convenient Spring-friendly tools for accessing the [Pivotal Gemfire](#) data management platform. There is a `spring-boot-starter-data-gemfire` “Starter” for collecting the dependencies in a convenient way. There is currently no auto-configuration support for Gemfire, but you can enable Spring Data Repositories with a single annotation: `@EnableGemfireRepositories`.

32.5 Solr

[Apache Solr](#) is a search engine. Spring Boot offers basic auto-configuration for the Solr 5 client library and the abstractions on top of it provided by [Spring Data Solr](#). There is a `spring-boot-starter-data-solr` “Starter” for collecting the dependencies in a convenient way.

32.5.1 Connecting to Solr

You can inject an auto-configured `SolrClient` instance as you would any other Spring bean. By default, the instance tries to connect to a server at localhost:8983/solr. The following example shows how to inject a Solr

bean:

```
@Component
public class MyBean {

    private SolrClient solr;

    @Autowired
    public MyBean(SolrClient solr) {
        this.solr = solr;
    }

    // ...
}
```

If you add your own `@Bean` of type `SolrClient`, it replaces the default.

32.5.2 Spring Data Solr Repositories

Spring Data includes repository support for Apache Solr. As with the JPA repositories discussed earlier, the basic principle is that queries are automatically constructed for \ you based on method names.

In fact, both Spring Data JPA and Spring Data Solr share the same common infrastructure. You could take the JPA example from earlier and, assuming that `City` is now a `@SolrDocument` class rather than a JPA `@Entity`, it works in the same way.



For complete details of Spring Data Solr, refer to the [reference documentation](#).

32.6 Elasticsearch

[Elasticsearch](#) is an open source, distributed, RESTful search and analytics engine. Spring Boot offers basic auto-configuration for Elasticsearch.

Spring Boot supports several HTTP clients:

- The official Java "Low Level" and "High Level" REST clients
- [Jest](#)

The transport client is still being used by [Spring Data Elasticsearch](#), which you can start using with the `spring-boot-starter-data-elasticsearch` "Starter".

32.6.1 Connecting to Elasticsearch by REST clients

Elasticsearch ships [two different REST clients](#) that you can use to query a cluster: the "Low Level" client and the "High Level" client.

If you have the `org.elasticsearch.client:elasticsearch-rest-client` dependency on the classpath, Spring Boot will auto-configure and register a `RestClient` bean that by default targets `localhost:9200`. You can further tune how `RestClient` is configured, as shown in the following example:

```
spring.elasticsearch.rest.uris=https://search.example.com:9200
spring.elasticsearch.rest.username=user
spring.elasticsearch.rest.password=secret
```

You can also register an arbitrary number of beans that implement `RestClientBuilderCustomizer` for more advanced customizations. To take full control over the registration, define a `RestClient` bean.

If you have the `org.elasticsearch.client:elasticsearch-rest-high-level-client` dependency on the classpath, Spring Boot will auto-configure a `RestHighLevelClient`, which wraps any existing `RestClient` bean, reusing its HTTP configuration.

32.6.2 Connecting to Elasticsearch by Using Jest

If you have `Jest` on the classpath, you can inject an auto-configured `JestClient` that by default targets `localhost:9200`. You can further tune how the client is configured, as shown in the following example:

```
spring.elasticsearch.jest.uris=https://search.example.com:9200
spring.elasticsearch.jest.read-timeout=10000
spring.elasticsearch.jest.username=user
spring.elasticsearch.jest.password=secret
```

You can also register an arbitrary number of beans that implement `HttpClientConfigBuilderCustomizer` for more advanced customizations. The following example tunes additional HTTP settings:

```
static class HttpSettingsCustomizer implements  
HttpClientConfigBuilderCustomizer {  
  
    @Override  
    public void customize(HttpClientConfig.Builder builder) {  
  
        builder.maxTotalConnection(100).defaultMaxTotalConnectionPerRou-  
te(5);  
    }  
  
}
```

To take full control over the registration, define a `JestClient` bean.

32.6.3 Connecting to Elasticsearch by Using Spring Data

To connect to Elasticsearch, you must provide the address of one or more cluster nodes. The address can be specified by setting the `spring.data.elasticsearch.cluster-nodes` property to a comma-separated `host:port` list. With this configuration in place, an `ElasticsearchTemplate` or `TransportClient` can be injected like any other Spring bean, as shown in the following example:

```
spring.data.elasticsearch.cluster-nodes=localhost:9300  
@Component  
public class MyBean {  
  
    private final ElasticsearchTemplate template;  
  
    public MyBean(ElasticsearchTemplate template) {  
        this.template = template;  
    }  
  
    // ...  
}
```

If you add your own `ElasticsearchTemplate` or `TransportClient` `@Bean`, it

replaces the default.

32.6.4 Spring Data Elasticsearch Repositories

Spring Data includes repository support for Elasticsearch. As with the JPA repositories discussed earlier, the basic principle is that queries are constructed for you automatically based on method names.

In fact, both Spring Data JPA and Spring Data Elasticsearch share the same common infrastructure. You could take the JPA example from earlier and, assuming that `City` is now an Elasticsearch `@Document` class rather than a JPA `@Entity`, it works in the same way.



For complete details of Spring Data Elasticsearch, refer to the [reference documentation](#).

32.7 Cassandra

[Cassandra](#) is an open source, distributed database management system designed to handle large amounts of data across many commodity servers. Spring Boot offers auto-configuration for Cassandra and the abstractions on top of it provided by [Spring Data Cassandra](#). There is a `spring-boot-starter-data-cassandra` “Starter” for collecting the dependencies in a convenient way.

32.7.1 Connecting to Cassandra

You can inject an auto-configured `CassandraTemplate` or a Cassandra `Session` instance as you would with any other Spring Bean. The `spring.data.cassandra.*` properties can be used to customize the connection. Generally, you provide `keyspace-name` and `contact-points` properties, as shown in the following example:

```
spring.data.cassandra.keyspace-name=mykeyspace
spring.data.cassandra.contact-points=cassandrahost1,cassandrahost2
```

You can also register an arbitrary number of beans that implement `ClusterBuilderCustomizer` for more advanced customizations.

The following code listing shows how to inject a Cassandra bean:

```
@Component
public class MyBean {

    private CassandraTemplate template;

    @Autowired
    public MyBean(CassandraTemplate template) {
        this.template = template;
    }

    // ...
}
```

If you add your own `@Bean` of type `CassandraTemplate`, it replaces the default.

32.7.2 Spring Data Cassandra Repositories

Spring Data includes basic repository support for Cassandra. Currently, this is more limited than the JPA repositories discussed earlier and needs to annotate finder methods with `@Query`.



For complete details of Spring Data Cassandra, refer to the [reference documentation](#).

32.8 Couchbase

[Couchbase](#) is an open-source, distributed, multi-model NoSQL document-oriented database that is optimized for interactive applications. Spring Boot offers auto-configuration for Couchbase and the abstractions on top of it provided by [Spring Data Couchbase](#). There are `spring-boot-starter-data-couchbase` and `spring-boot-starter-data-couchbase-reactive` “Starters” for collecting the dependencies in a convenient way.

32.8.1 Connecting to Couchbase

You can get a `Bucket` and `Cluster` by adding the Couchbase SDK and

some configuration. The `spring.couchbase.*` properties can be used to customize the connection. Generally, you provide the bootstrap hosts, bucket name, and password, as shown in the following example:

```
spring.couchbase.bootstrap-hosts=my-host-1,192.168.1.123
spring.couchbase.bucket.name=my-bucket
spring.couchbase.bucket.password=secret
```



You need to provide *at least* the bootstrap host(s), in which case the bucket name is `default` and the password is an empty String. Alternatively, you can define your own `org.springframework.data.couchbase.config.CouchbaseConfigurer @Bean` to take control over the whole configuration.

It is also possible to customize some of the `CouchbaseEnvironment` settings. For instance, the following configuration changes the timeout to use to open a new `Bucket` and enables SSL support:

```
spring.couchbase.env.timeouts.connect=3000
spring.couchbase.env.ssl.key-store=/location/of/keystore.jks
spring.couchbase.env.ssl.key-store-password=secret
```

Check the `spring.couchbase.env.*` properties for more details.

32.8.2 Spring Data Couchbase Repositories

Spring Data includes repository support for Couchbase. For complete details of Spring Data Couchbase, refer to the [reference documentation](#).

You can inject an auto-configured `CouchbaseTemplate` instance as you would with any other Spring Bean, provided a *default* `CouchbaseConfigurer` is available (which happens when you enable Couchbase support, as explained earlier).

The following examples shows how to inject a Couchbase bean:

```
@Component
public class MyBean {
```

```

private final CouchbaseTemplate template;

@Autowired
public MyBean(CouchbaseTemplate template) {
    this.template = template;
}

// ...

}

```

There are a few beans that you can define in your own configuration to override those provided by the auto-configuration:

- A `CouchbaseTemplate` `@Bean` with a name of `couchbaseTemplate`.
- An `IndexManager` `@Bean` with a name of `couchbaseIndexManager`.
- A `CustomConversions` `@Bean` with a name of `couchbaseCustomConversions`.

To avoid hard-coding those names in your own config, you can reuse `BeanNames` provided by Spring Data Couchbase. For instance, you can customize the converters to use, as follows:

```

@Configuration
public class SomeConfiguration {

    @Bean(BeanNames.COUCHBASE_CUSTOM_CONVERSIONS)
    public CustomConversions myCustomConversions() {
        return new CustomConversions(...);
    }

    // ...
}

```



If you want to fully bypass the auto-configuration for Spring Data Couchbase, provide your own implementation

`of org.springframework.data.couchbase.config.AbstractCouchba`

seDataConfiguration.

32.9 LDAP

[LDAP](#) (Lightweight Directory Access Protocol) is an open, vendor-neutral, industry standard application protocol for accessing and maintaining distributed directory information services over an IP network. Spring Boot offers auto-configuration for any compliant LDAP server as well as support for the embedded in-memory LDAP server from [UnboundID](#).

LDAP abstractions are provided by [Spring Data LDAP](#). There is a [spring-boot-starter-data-ldap](#) “Starter” for collecting the dependencies in a convenient way.

32.9.1 Connecting to an LDAP Server

To connect to an LDAP server, make sure you declare a dependency on the [spring-boot-starter-data-ldap](#) “Starter” or [spring-ldap-core](#) and then declare the URLs of your server in your application.properties, as shown in the following example:

```
spring.ldap.urls=ldap://myserver:1235
spring.ldap.username=admin
spring.ldap.password=secret
```

If you need to customize connection settings, you can use the [spring.ldap.base](#) and [spring.ldap.base-environment](#) properties.

An [LdapContextSource](#) is auto-configured based on these settings. If you need to customize it, for instance to use a [PooledContextSource](#), you can still inject the auto-configured [LdapContextSource](#). Make sure to flag your customized [ContextSource](#) as [@Primary](#) so that the auto-configured [LdapTemplate](#) uses it.

32.9.2 Spring Data LDAP Repositories

Spring Data includes repository support for LDAP. For complete details of Spring Data LDAP, refer to the [reference documentation](#).

You can also inject an auto-configured [LdapTemplate](#) instance as you would with any other Spring Bean, as shown in the following example:

```
@Component
public class MyBean {

    private final LdapTemplate template;

    @Autowired
    public MyBean(LdapTemplate template) {
        this.template = template;
    }

    // ...
}
```

32.9.3 Embedded In-memory LDAP Server

For testing purposes, Spring Boot supports auto-configuration of an in-memory LDAP server from [UnboundID](#). To configure the server, add a dependency to `com.unboundid:unboundid-ldapsdk` and declare a `base-dn` property, as follows:

```
spring.ldap.embedded.base-dn=dc=spring,dc=io
```



It is possible to define multiple base-dn values, however, since distinguished names usually contain commas, they must be defined using the correct notation. In yaml files, you can use the yaml list notation:

```
spring.ldap.embedded.base-dn:
  - dc=spring,dc=io
  - dc=pivotal,dc=io
```

In properties files, you must include the index as part of the property name:

```
spring.ldap.embedded.base-dn[0]=dc=spring,dc=io
spring.ldap.embedded.base-dn[1]=dc=pivotal,dc=io
```

By default, the server starts on a random port and triggers the regular LDAP support. There is no need to specify a `spring.ldap.urls` property.

If there is a `schema.ldif` file on your classpath, it is used to initialize the server. If you want to load the initialization script from a different resource, you can also use the `spring.ldap.embedded.ldif` property.

By default, a standard schema is used to validate **LDIF** files. You can turn off validation altogether by setting the `spring.ldap.embedded.validation.enabled` property. If you have custom attributes, you can use `spring.ldap.embedded.validation.schema` to define your custom attribute types or object classes.

32.10 InfluxDB

[InfluxDB](#) is an open-source time series database optimized for fast, high-availability storage and retrieval of time series data in fields such as operations monitoring, application metrics, Internet-of-Things sensor data, and real-time analytics.

32.10.1 Connecting to InfluxDB

Spring Boot auto-configures an **InfluxDB** instance, provided the `influxdb-java` client is on the classpath and the URL of the database is set, as shown in the following example:

```
spring.influx.url=https://172.0.0.1:8086
```

If the connection to InfluxDB requires a user and password, you can set the `spring.influx.user` and `spring.influx.password` properties accordingly.

InfluxDB relies on OkHttp. If you need to tune the http client `InfluxDB` uses behind the scenes, you can register an `InfluxDbOkHttpClientBuilderProvider` bean.

33. Caching

The Spring Framework provides support for transparently adding caching to an application. At its core, the abstraction applies caching to methods, thus reducing the number of executions based on the information available in the cache. The caching logic is applied transparently, without any interference to the invoker. Spring Boot auto-configures the cache

infrastructure as long as caching support is enabled via the `@EnableCaching` annotation.



Check the [relevant section](#) of the Spring Framework reference for more details.

In a nutshell, adding caching to an operation of your service is as easy as adding the relevant annotation to its method, as shown in the following example:

```
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Component;

@Component
public class MathService {

    @Cacheable("piDecimals")
    public int computePiDecimal(int i) {
        // ...
    }
}
```

This example demonstrates the use of caching on a potentially costly operation. Before invoking `computePiDecimal`, the abstraction looks for an entry in the `piDecimals` cache that matches the `i` argument. If an entry is found, the content in the cache is immediately returned to the caller, and the method is not invoked. Otherwise, the method is invoked, and the cache is updated before returning the value.



Caution

You can also use the standard JSR-107 (JCache) annotations (such as `@CacheResult`) transparently. However, we strongly advise you to not mix and match the Spring Cache and JCache annotations.

If you do not add any specific cache library, Spring Boot auto-configures a [simple provider](#) that uses concurrent maps in memory. When a cache is

required (such as `piDecimals` in the preceding example), this provider creates it for you. The simple provider is not really recommended for production usage, but it is great for getting started and making sure that you understand the features. When you have made up your mind about the cache provider to use, please make sure to read its documentation to figure out how to configure the caches that your application uses. Nearly all providers require you to explicitly configure every cache that you use in the application. Some offer a way to customize the default caches defined by the `spring.cache.cache-names` property.



It is also possible to transparently [update](#) or [evict](#) data from the cache.

33.1 Supported Cache Providers

The cache abstraction does not provide an actual store and relies on abstraction materialized by the `org.springframework.cache.Cache` and `org.springframework.cache.CacheManager` interfaces.

If you have not defined a bean of type `CacheManager` or a `CacheResolver` named `cacheResolver` (see [CachingConfigurer](#)), Spring Boot tries to detect the following providers (in the indicated order):

1. [Generic](#)
2. [JCache \(JSR-107\)](#) (EhCache 3, Hazelcast, Infinispan, and others)
3. [EhCache 2.x](#)
4. [Hazelcast](#)
5. [Infinispan](#)
6. [Couchbase](#)
7. [Redis](#)
8. [Caffeine](#)
9. [Simple](#)



It is also possible to *force* a particular cache provider by setting the `spring.cache.type` property. Use this property if you need to [disable caching altogether](#) in certain environment (such as tests).



Use the `spring-boot-starter-cache` “Starter” to quickly add basic caching dependencies. The starter brings in `spring-context-support`. If you add dependencies manually, you must include `spring-context-support` in order to use the JCache, EhCache 2.x, or Guava support.

If the `CacheManager` is auto-configured by Spring Boot, you can further tune its configuration before it is fully initialized by exposing a bean that implements the `CacheManagerCustomizer` interface. The following example sets a flag to say that null values should be passed down to the underlying map:

```
@Bean
public CacheManagerCustomizer<ConcurrentMapCacheManager>
cacheManagerCustomizer() {
    return new CacheManagerCustomizer<ConcurrentMapCacheManager>()
{
    @Override
    public void customize(ConcurrentMapCacheManager
cacheManager) {
        cacheManager.setAllowNullValues(false);
    }
}
```



In the preceding example, an auto-configured `ConcurrentMapCacheManager` is expected. If that is not the case (either you provided your own config or a different cache provider was auto-configured), the customizer is not invoked at all. You can have as many customizers as you want, and you can also order them by using `@Order` or `Ordered`.

33.1.1 Generic

Generic caching is used if the context defines *at*

least one `org.springframework.cache.Cache` bean.

A `CacheManager` wrapping all beans of that type is created.

33.1.2 JCache (JSR-107)

`JCache` is bootstrapped through the presence of a `javax.cache.spi.CachingProvider` on the classpath (that is, a JSR-107 compliant caching library exists on the classpath), and the `JCacheCacheManager` is provided by the `spring-boot-starter-cache` “Starter”. Various compliant libraries are available, and Spring Boot provides dependency management for Ehcache 3, Hazelcast, and Infinispan. Any other compliant library can be added as well.

It might happen that more than one provider is present, in which case the provider must be explicitly specified. Even if the JSR-107 standard does not enforce a standardized way to define the location of the configuration file, Spring Boot does its best to accommodate setting a cache with implementation details, as shown in the following example:

```
# Only necessary if more than one provider is present
spring.cache.jcache.provider=com.acme.MyCachingProvider
spring.cache.jcache.config=classpath:acme.xml
```



When a cache library offers both a native implementation and JSR-107 support, Spring Boot prefers the JSR-107 support, so that the same features are available if you switch to a different JSR-107 implementation.



Spring Boot has [general support for Hazelcast](#). If a single `HazelcastInstance` is available, it is automatically reused for the `CacheManager` as well, unless the `spring.cache.jcache.config` property is specified.

There are two ways to customize the underlying `javax.cache.cacheManager`:

- Caches can be created on startup by setting

- the `spring.cache.cache-names` property. If a custom `javax.cache.configuration.Configuration` bean is defined, it is used to customize them.
- `org.springframework.boot.autoconfigure.cache.JCacheManagerCustomizer` beans are invoked with the reference of the `CacheManager` for full customization.



If a standard `javax.cache.CacheManager` bean is defined, it is wrapped automatically in an `org.springframework.cache.CacheManager` implementation that the abstraction expects. No further customization is applied to it.

33.1.3 EhCache 2.x

[EhCache](#) 2.x is used if a file named `ehcache.xml` can be found at the root of the classpath. If EhCache 2.x is found, the `EhCacheCacheManager` provided by the `spring-boot-starter-cache` “Starter” is used to bootstrap the cache manager. An alternate configuration file can be provided as well, as shown in the following example:

```
spring.cache.ehcache.config=classpath:config/another-config.xml
```

33.1.4 Hazelcast

Spring Boot has [general support for Hazelcast](#). If a `HazelcastInstance` has been auto-configured, it is automatically wrapped in a `CacheManager`.

33.1.5 Infinispan

[Infinispan](#) has no default configuration file location, so it must be specified explicitly. Otherwise, the default bootstrap is used.

```
spring.cache.infinispan.config=infinispan.xml
```

Caches can be created on startup by setting the `spring.cache.cache-names` property. If a custom `ConfigurationBuilder` bean is defined, it is

used to customize the caches.



The support of Infinispan in Spring Boot is restricted to the embedded mode and is quite basic. If you want more options, you should use the official Infinispan Spring Boot starter instead. See [Infinispan's documentation](#) for more details.

33.1.6 Couchbase

If the [Couchbase](#) Java client and the `couchbase-spring-cache` implementation are available and Couchbase is [configured](#), a `CouchbaseCacheManager` is auto-configured. It is also possible to create additional caches on startup by setting the `spring.cache.cache-names` property. These caches operate on the `Bucket` that was auto-configured. You can *also* create additional caches on another `Bucket` by using the customizer. Assume you need two caches (`cache1` and `cache2`) on the "main" `Bucket` and one (`cache3`) cache with a custom time to live of 2 seconds on the "another" `Bucket`. You can create the first two caches through configuration, as follows:

```
spring.cache.cache-names=cache1,cache2
```

Then you can define a `@Configuration` class to configure the extra `Bucket` and the `cache3` cache, as follows:

```
@Configuration
public class CouchbaseCacheConfiguration {

    private final Cluster cluster;

    public CouchbaseCacheConfiguration(Cluster cluster) {
        this.cluster = cluster;
    }

    @Bean
    public Bucket anotherBucket() {
        return this.cluster.openBucket("another", "secret");
    }
}
```

```
@Bean
public CacheManagerCustomizer<CouchbaseCacheManager>
cacheManagerCustomizer() {
    return c -> {
        c.prepareCache("cache3",
CacheBuilder.newInstance(anotherBucket())
        .withExpiration(2));
    };
}

}
```

This sample configuration reuses the `Cluster` that was created through auto-configuration.

33.1.7 Redis

If [Redis](#) is available and configured, a `RedisCacheManager` is auto-configured. It is possible to create additional caches on startup by setting the `spring.cache.cache-names` property and cache defaults can be configured by using `spring.cache.redis.*` properties. For instance, the following configuration creates `cache1` and `cache2` caches with a *time to live* of 10 minutes:

```
spring.cache.cache-names=cache1,cache2
spring.cache.redis.time-to-live=600000
```



By default, a key prefix is added so that, if two separate caches use the same key, Redis does not have overlapping keys and cannot return invalid values. We strongly recommend keeping this setting enabled if you create your own `RedisCacheManager`.



You can take full control of the configuration by adding a `RedisCacheConfiguration @Bean` of your own. This can be useful if you're looking for customizing the serialization strategy.

33.1.8 Caffeine

[Caffeine](#) is a Java 8 rewrite of Guava's cache that supersedes support for Guava. If Caffeine is present, a `CaffeineCacheManager` (provided by the `spring-boot-starter-cache` "Starter") is auto-configured. Caches can be created on startup by setting the `spring.cache.cache-names` property and can be customized by one of the following (in the indicated order):

1. A cache spec defined by `spring.cache.caffeine.spec`
2. A `com.github.benmanes.caffeine.cache.CaffeineSpec` bean is defined
3. A `com.github.benmanes.caffeine.cache.Caffeine` bean is defined

For instance, the following configuration creates `cache1` and `cache2` caches with a maximum size of 500 and a *time to live* of 10 minutes

```
spring.cache.cache-names=cache1,cache2
spring.cache.caffeine.spec=maximumSize=500,expireAfterAccess=600s
```

If a `com.github.benmanes.caffeine.cache.CacheLoader` bean is defined, it is automatically associated to the `CaffeineCacheManager`. Since the `CacheLoader` is going to be associated with *all* caches managed by the cache manager, it must be defined as `CacheLoader<Object, Object>`. The auto-configuration ignores any other generic type.

33.1.9 Simple

If none of the other providers can be found, a simple implementation using a `ConcurrentHashMap` as the cache store is configured. This is the default if no caching library is present in your application. By default, caches are created as needed, but you can restrict the list of available caches by setting the `cache-names` property. For instance, if you want only `cache1` and `cache2` caches, set the `cache-names` property as follows:

```
spring.cache.cache-names=cache1,cache2
```

If you do so and your application uses a cache not listed, then it fails at runtime when the cache is needed, but not on startup. This is similar to the way the "real" cache providers behave if you use an undeclared cache.

33.1.10 None

When `@EnableCaching` is present in your configuration, a suitable cache configuration is expected as well. If you need to disable caching altogether in certain environments, force the cache type to `none` to use a no-op implementation, as shown in the following example:

```
spring.cache.type=none
```

34. Messaging

The Spring Framework provides extensive support for integrating with messaging systems, from simplified use of the JMS API using `JmsTemplate` to a complete infrastructure to receive messages asynchronously. Spring AMQP provides a similar feature set for the Advanced Message Queuing Protocol. Spring Boot also provides auto-configuration options for `RabbitTemplate` and RabbitMQ. Spring WebSocket natively includes support for STOMP messaging, and Spring Boot has support for that through starters and a small amount of auto-configuration. Spring Boot also has support for Apache Kafka.

34.1 JMS

The `javax.jms.ConnectionFactory` interface provides a standard method of creating a `javax.jms.Connection` for interacting with a JMS broker. Although Spring needs a `ConnectionFactory` to work with JMS, you generally need not use it directly yourself and can instead rely on higher level messaging abstractions. (See the [relevant section](#) of the Spring Framework reference documentation for details.) Spring Boot also auto-configures the necessary infrastructure to send and receive messages.

34.1.1 ActiveMQ Support

When [ActiveMQ](#) is available on the classpath, Spring Boot can also configure a `ConnectionFactory`. If the broker is present, an embedded broker is automatically started and configured (provided no broker URL is specified through configuration).



If you use `spring-boot-starter-activemq`, the necessary dependencies to connect or embed an ActiveMQ instance are provided, as is the Spring infrastructure to integrate with JMS.

ActiveMQ configuration is controlled by external configuration properties in `spring.activemq.*`. For example, you might declare the following section in `application.properties`:

```
spring.activemq.broker-url=tcp://192.168.1.210:9876  
spring.activemq.user=admin  
spring.activemq.password=secret
```

By default, a `CachingConnectionFactory` wraps the native `ConnectionFactory` with sensible settings that you can control by external configuration properties in `spring.jms.*`:

```
spring.jms.cache.session-cache-size=5
```

If you'd rather use native pooling, you can do so by adding a dependency to `org.messaginghub:pooled-jms` and configuring the `JmsPoolConnectionFactory` accordingly, as shown in the following example:

```
spring.activemq.pool.enabled=true  
spring.activemq.pool.max-connections=50
```



See [ActiveMQProperties](#) for more of the supported options. You can also register an arbitrary number of beans that implement `ActiveMQConnectionFactoryCustomizer` for more advanced customizations.

By default, ActiveMQ creates a destination if it does not yet exist so that destinations are resolved against their provided names.

34.1.2 Artemis Support

Spring Boot can auto-configure a `ConnectionFactory` when it detects that [Artemis](#) is available on the classpath. If the broker is present, an embedded broker is automatically started and configured (unless the mode property has been explicitly set). The supported modes are `embedded` (to make explicit that an embedded broker is required and that an error should occur if the broker is not available on the classpath) and `native` (to connect to a broker using the `netty` transport protocol). When the latter is configured, Spring Boot configures a `ConnectionFactory` that connects to a broker running on the local machine with the default settings.



If you use `spring-boot-starter-artemis`, the necessary dependencies to connect to an existing Artemis instance are provided, as well as the Spring infrastructure to integrate with JMS.

Adding `org.apache.activemq:artemis-jms-server` to your application lets you use embedded mode.

Artemis configuration is controlled by external configuration properties in `spring.artemis.*`. For example, you might declare the following section in `application.properties`:

```
spring.artemis.mode=native
spring.artemis.host=192.168.1.210
spring.artemis.port=9876
spring.artemis.user=admin
spring.artemis.password=secret
```

When embedding the broker, you can choose if you want to enable persistence and list the destinations that should be made available. These can be specified as a comma-separated list to create them with the default options, or you can define bean(s) of type `org.apache.activemq.artemis.jms.server.config.JMSQueueConfiguration` or `org.apache.activemq.artemis.jms.server.config.TopicConfiguration`, for advanced queue and topic configurations, respectively.

By default, a `CachingConnectionFactory` wraps the native `ConnectionFactory` with sensible settings that you can control by external configuration properties in `spring.jms.*`:

```
spring.jms.cache.session-cache-size=5
```

If you'd rather use native pooling, you can do so by adding a dependency to `org.messaginghub:pooled-jms` and configuring the `JmsPoolConnectionFactory` accordingly, as shown in the following example:

```
spring.artemis.pool.enabled=true
spring.artemis.pool.max-connections=50
```

See [ArtemisProperties](#) for more supported options.

No JNDI lookup is involved, and destinations are resolved against their names, using either the `name` attribute in the Artemis configuration or the names provided through configuration.

34.1.3 Using a JNDI ConnectionFactory

If you are running your application in an application server, Spring Boot tries to locate a JMS `ConnectionFactory` by using JNDI. By default, the `java:/JmsXA` and `java:/XAConnectionFactory` location are checked. You can use the `spring.jms.jndi-name` property if you need to specify an alternative location, as shown in the following example:

```
spring.jms.jndi-name=java:/MyConnectionFactory
```

34.1.4 Sending a Message

Spring's `JmsTemplate` is auto-configured, and you can autowire it directly into your own beans, as shown in the following example:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {
```

```

private final JmsTemplate jmsTemplate;

@Autowired
public MyBean(JmsTemplate jmsTemplate) {
    this.jmsTemplate = jmsTemplate;
}

// ...

}

```



`JmsMessagingTemplate` can be injected in a similar manner. If a `DestinationResolver` or a `MessageConverter` bean is defined, it is associated automatically to the auto-configured `JmsTemplate`.

34.1.5 Receiving a Message

When the JMS infrastructure is present, any bean can be annotated with `@JmsListener` to create a listener endpoint. If no `JmsListenerContainerFactory` has been defined, a default one is configured automatically. If a `DestinationResolver` or a `MessageConverter` beans is defined, it is associated automatically to the default factory.

By default, the default factory is transactional. If you run in an infrastructure where a `JtaTransactionManager` is present, it is associated to the listener container by default. If not, the `sessionTransacted` flag is enabled. In that latter scenario, you can associate your local data store transaction to the processing of an incoming message by adding `@Transactional` on your listener method (or a delegate thereof). This ensures that the incoming message is acknowledged, once the local transaction has completed. This also includes sending response messages that have been performed on the same JMS session.

The following component creates a listener endpoint on the `someQueue` destination:

```

@Component
public class MyBean {

```

```
@JmsListener(destination = "someQueue")
public void processMessage(String content) {
    // ...
}

}
```



See [the Javadoc of @EnableJms](#) for more details.

If you need to create more `JmsListenerContainerFactory` instances or if you want to override the default, Spring Boot provides a `DefaultJmsListenerContainerConfigurer` that you can use to initialize a `DefaultJmsListenerContainerFactory` with the same settings as the one that is auto-configured.

For instance, the following example exposes another factory that uses a specific `MessageConverter`:

```
@Configuration
static class JmsConfiguration {

    @Bean
    public DefaultJmsListenerContainerFactory myFactory(
        DefaultJmsListenerContainerFactoryConfigurer
configurer) {
        DefaultJmsListenerContainerFactory factory =
            new DefaultJmsListenerContainerFactory();
        configurer.configure(factory, connectionFactory());
        factory.setMessageConverter(myMessageConverter());
        return factory;
    }

}
```

Then you can use the factory in any `@JmsListener`-annotated method as follows:

```
@Component
public class MyBean {
```

```
@JmsListener(destination = "someQueue",
containerFactory="myFactory")
public void processMessage(String content) {
    // ...
}
```

34.2 AMQP

The Advanced Message Queuing Protocol (AMQP) is a platform-neutral, wire-level protocol for message-oriented middleware. The Spring AMQP project applies core Spring concepts to the development of AMQP-based messaging solutions. Spring Boot offers several conveniences for working with AMQP through RabbitMQ, including the `spring-boot-starter-amqp` “Starter”.

34.2.1 RabbitMQ support

RabbitMQ is a lightweight, reliable, scalable, and portable message broker based on the AMQP protocol. Spring uses RabbitMQ to communicate through the AMQP protocol.

RabbitMQ configuration is controlled by external configuration properties in `spring.rabbitmq.*`. For example, you might declare the following section in `application.properties`:

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=admin
spring.rabbitmq.password=secret
```

If a `ConnectionNameStrategy` bean exists in the context, it will be automatically used to name connections created by the auto-configured `ConnectionFactory`. See `RabbitProperties` for more of the supported options.



See [Understanding AMQP, the protocol used by RabbitMQ](#) for more details.

34.2.2 Sending a Message

Spring's `AmqpTemplate` and `AmqpAdmin` are auto-configured, and you can autowire them directly into your own beans, as shown in the following example:

```
import org.springframework.amqp.core.AmqpAdmin;
import org.springframework.amqp.core.AmqpTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final AmqpAdmin amqpAdmin;
    private final AmqpTemplate amqpTemplate;

    @Autowired
    public MyBean(AmqpAdmin amqpAdmin, AmqpTemplate amqpTemplate) {
        this.amqpAdmin = amqpAdmin;
        this.amqpTemplate = amqpTemplate;
    }

    // ...
}
```



`RabbitMessagingTemplate` can be injected in a similar manner. If a `MessageConverter` bean is defined, it is associated automatically to the auto-configured `AmqpTemplate`.

If necessary, any `org.springframework.amqp.core.Queue` that is defined as a bean is automatically used to declare a corresponding queue on the RabbitMQ instance.

To retry operations, you can enable retries on the `AmqpTemplate` (for example, in the event that the broker connection is lost):

```
spring.rabbitmq.template.retry.enabled=true
spring.rabbitmq.template.retry.initial-interval=2s
```

Retries are disabled by default. You can also customize the `RetryTemplate` programmatically by declaring a `RabbitRetryTemplateCustomizer` bean.

34.2.3 Receiving a Message

When the Rabbit infrastructure is present, any bean can be annotated with `@RabbitListener` to create a listener endpoint. If no `RabbitListenerContainerFactory` has been defined, a default `SimpleRabbitListenerContainerFactory` is automatically configured and you can switch to a direct container using the `spring.rabbitmq.listener.type` property. If a `MessageConverter` or a `MessageRecoverer` bean is defined, it is automatically associated with the default factory.

The following sample component creates a listener endpoint on the `someQueue` queue:

```
@Component
public class MyBean {

    @RabbitListener(queues = "someQueue")
    public void processMessage(String content) {
        // ...
    }
}
```



See [the Javadoc of `@EnableRabbit`](#) for more details.

If you need to create more `RabbitListenerContainerFactory` instances or if you want to override the default, Spring Boot provides a `SimpleRabbitListenerContainerFactoryConfigurer` and a `DirectRabbitListenerContainerFactoryConfigurer` that you can use to initialize a `SimpleRabbitListenerContainerFactory` and a `DirectRabbitListenerContainerFactory` with the same settings as the factories used by the auto-configuration.



It does not matter which container type you chose. Those two beans are exposed by the auto-configuration.

For instance, the following configuration class exposes another factory that uses a specific `MessageConverter`:

```
@Configuration
static class RabbitConfiguration {

    @Bean
    public SimpleRabbitListenerContainerFactory myFactory(
            SimpleRabbitListenerContainerFactoryConfigurer
configurer) {
        SimpleRabbitListenerContainerFactory factory =
                new
SimpleRabbitListenerContainerFactory();
        configurer.configure(factory, connectionFactory);
        factory.setMessageConverter(myMessageConverter());
        return factory;
    }

}
```

Then you can use the factory in any `@RabbitListener`-annotated method, as follows:

```
@Component
public class MyBean {

    @RabbitListener(queues = "someQueue",
containerFactory="myFactory")
    public void processMessage(String content) {
        // ...
    }
}
```

You can enable retries to handle situations where your listener throws an exception. By default, `RejectAndDontRequeueRecoverer` is used, but you

can define a `MessageRecoverer` of your own. When retries are exhausted, the message is rejected and either dropped or routed to a dead-letter exchange if the broker is configured to do so. By default, retries are disabled. You can also customize the `RetryTemplate` programmatically by declaring a `RabbitRetryTemplateCustomizer` bean.



Important

By default, if retries are disabled and the listener throws an exception, the delivery is retried indefinitely. You can modify this behavior in two ways: Set the `defaultQueueRejected` property to `false` so that zero re-deliveries are attempted or throw an `AmqpRejectAndDontRequeueException` to signal the message should be rejected. The latter is the mechanism used when retries are enabled and the maximum number of delivery attempts is reached.

34.3 Apache Kafka Support

[Apache Kafka](#) is supported by providing auto-configuration of the `spring-kafka` project.

Kafka configuration is controlled by external configuration properties in `spring.kafka.*`. For example, you might declare the following section in `application.properties`:

```
spring.kafka.bootstrap-servers=localhost:9092  
spring.kafka.consumer.group-id=myGroup
```



To create a topic on startup, add a bean of type `NewTopic`. If the topic already exists, the bean is ignored.

See [KafkaProperties](#) for more supported options.

34.3.1 Sending a Message

Spring's `KafkaTemplate` is auto-configured, and you can autowire it directly in your own beans, as shown in the following example:

```
@Component
```

```

public class MyBean {

    private final KafkaTemplate kafkaTemplate;

    @Autowired
    public MyBean(KafkaTemplate kafkaTemplate) {
        this.kafkaTemplate = kafkaTemplate;
    }

    // ...
}

```



If the property `spring.kafka.producer.transaction-id-prefix` is defined, a `KafkaTransactionManager` is automatically configured. Also, if a `RecordMessageConverter` bean is defined, it is automatically associated to the auto-configured `KafkaTemplate`.

34.3.2 Receiving a Message

When the Apache Kafka infrastructure is present, any bean can be annotated with `@KafkaListener` to create a listener endpoint. If no `KafkaListenerContainerFactory` has been defined, a default one is automatically configured with keys defined in `spring.kafka.listener.*`.

The following component creates a listener endpoint on the `someTopic` topic:

```

@Component
public class MyBean {

    @KafkaListener(topics = "someTopic")
    public void processMessage(String content) {
        // ...
    }
}

```

If a `KafkaTransactionManager` bean is defined, it is automatically associated to the container factory. Similarly, if

a `RecordMessageConverter`, `ErrorHandler` or `AfterRollbackProcessor` bean is defined, it is automatically associated to the default factory.



A custom `ChainedKafkaTransactionManager` must be marked `@Primary` as it usually references the auto-configured `KafkaTransactionManager` bean.

34.3.3 Kafka Streams

Spring for Apache Kafka provides a factory bean to create a `StreamsBuilder` object and manage the lifecycle of its streams. Spring Boot auto-configures the required `KafkaStreamsConfiguration` bean as long as `kafka-streams` is on the classpath and Kafka Streams is enabled via the `@EnableKafkaStreams` annotation.

Enabling Kafka Streams means that the application id and bootstrap servers must be set. The former can be configured using `spring.kafka.streams.application-id`, defaulting to `spring.application.name` if not set. The latter can be set globally or specifically overridden just for streams.

Several additional properties are available using dedicated properties; other arbitrary Kafka properties can be set using the `spring.kafka.streams.properties` namespace. See also [Section 34.3.4, “Additional Kafka Properties”](#) for more information.

To use the factory bean, simply wire `StreamsBuilder` into your `@Bean` as shown in the following example:

```
@Configuration  
@EnableKafkaStreams  
static class KafkaStreamsExampleConfiguration {  
  
    @Bean  
    public KStream<Integer, String> kStream(StreamsBuilder  
streamsBuilder) {  
        KStream<Integer, String> stream =  
streamsBuilder.stream("ks1In");  
        stream.map((k, v) -> new KeyValue<>(k,
```

```

    v.toUpperCase()).to("ks1out",
                           Produced.with(Serdes.Integer(), new
JsonSerde<>()));
        return stream;
    }

}

```

By default, the streams managed by the `StreamBuilder` object it creates are started automatically. You can customize this behaviour using the `spring.kafka.streams.auto-startup` property.

34.3.4 Additional Kafka Properties

The properties supported by auto configuration are shown in [Appendix A, Common application properties](#). Note that, for the most part, these properties (hyphenated or camelCase) map directly to the Apache Kafka dotted properties. Refer to the Apache Kafka documentation for details.

The first few of these properties apply to all components (producers, consumers, admins, and streams) but can be specified at the component level if you wish to use different values. Apache Kafka designates properties with an importance of HIGH, MEDIUM, or LOW. Spring Boot auto-configuration supports all HIGH importance properties, some selected MEDIUM and LOW properties, and any properties that do not have a default value.

Only a subset of the properties supported by Kafka are available directly through the `KafkaProperties` class. If you wish to configure the producer or consumer with additional properties that are not directly supported, use the following properties:

```

spring.kafka.properties.prop.one=first
spring.kafka.admin.properties.prop.two=second
spring.kafka.consumer.properties.prop.three=third
spring.kafka.producer.properties.prop.four=fourth
spring.kafka.streams.properties.prop.five=fifth

```

This sets the common `prop.one` Kafka property to `first` (applies to producers, consumers and admins), the `prop.two` admin property to `second`, the `prop.three` consumer property to `third`,

the `prop.four` producer property to `fourth` and the `prop.five` streams property to `fifth`.

You can also configure the Spring Kafka `JsonDeserializer` as follows:

```
spring.kafka.consumer.value-
deserializer=org.springframework.kafka.support.serializer.JsonDeserial
izer
spring.kafka.consumer.properties.spring.json.value.default.type=com.ex
ample.Invoice
spring.kafka.consumer.properties.spring.json.trusted.packages=com.exam
ple,org.acme
```

Similarly, you can disable the `JsonSerializer` default behavior of sending type information in headers:

```
spring.kafka.producer.value-
serializer=org.springframework.kafka.support.serializer.JsonSerializer
spring.kafka.producer.properties.spring.json.add.type.headers=false
```

Calling REST Services with `RestTemplate`

If you need to call remote REST services from your application, you can use the Spring Framework's `RestTemplate` class.

Since `RestTemplate` instances often need to be customized before being used, Spring Boot does not provide any single auto-configured `RestTemplate` bean. It does, however, auto-configure a `RestTemplateBuilder`, which can be used to create `RestTemplate` instances when needed. The auto-configured `RestTemplateBuilder` ensures that sensible `HttpMessageConverters` are applied to `RestTemplate` instances.

The following code shows a typical example:

```
@Service
public class MyService {

    private final RestTemplate restTemplate;

    public MyService(RestTemplateBuilder restTemplateBuilder) {
```

```

        this.restTemplate = restTemplateBuilder.build();
    }

    public Details someRestCall(String name) {
        return this.restTemplate.getForObject("/{name}/details",
Details.class, name);
    }

}

```



`RestTemplateBuilder` includes a number of useful methods that can be used to quickly configure a `RestTemplate`. For example, to add BASIC auth support, you can use `builder.basicAuthentication("user", "password").build()`.

35.1 RestTemplate Customization

There are three main approaches to `RestTemplate` customization, depending on how broadly you want the customizations to apply.

To make the scope of any customizations as narrow as possible, inject the auto-configured `RestTemplateBuilder` and then call its methods as required. Each method call returns a new `RestTemplateBuilder` instance, so the customizations only affect this use of the builder.

To make an application-wide, additive customization, use a `RestTemplateCustomizer` bean. All such beans are automatically registered with the auto-configured `RestTemplateBuilder` and are applied to any templates that are built with it.

The following example shows a customizer that configures the use of a proxy for all hosts except `192.168.0.5`:

```

static class ProxyCustomizer implements RestTemplateCustomizer {

    @Override
    public void customize(RestTemplate restTemplate) {
        HttpHost proxy = new HttpHost("proxy.example.com");
        HttpClient httpClient = HttpClientBuilder.create()
            .setRoutePlanner(new
DefaultProxyRoutePlanner(proxy) {

```

```

        @Override
        public HttpHost
determineProxy(HttpHost target,
                           HttpRequest
request, HttpContext context)
                           throws
HttpException {
        if
(target.getHostName().equals("192.168.0.5")) {
            return null;
        }
        return
super.determineProxy(target, request, context);
    }

}).build();
restTemplate.setRequestFactory(
new
HttpComponentsClientHttpRequestFactory(httpClient));
}

}

```

Finally, the most extreme (and rarely used) option is to create your own `RestTemplateBuilder` bean. Doing so switches off the auto-configuration of a `RestTemplateBuilder` and prevents any `RestTemplateCustomizer` beans from being used.

36. Calling REST Services with `WebClient`

If you have Spring WebFlux on your classpath, you can also choose to use `WebClient` to call remote REST services. Compared to `RestTemplate`, this client has a more functional feel and is fully reactive. You can learn more about the `WebClient` in the dedicated [section in the Spring Framework docs](#).

Spring Boot creates and pre-configures a `WebClient.Builder` for you; it is strongly advised to inject it in your components and use it to create `WebClient` instances. Spring Boot is configuring that builder to share HTTP resources, reflect codecs setup in the same fashion as the server ones (see [WebFlux HTTP codecs auto-configuration](#)), and more.

The following code shows a typical example:

```
@Service
public class MyService {

    private final WebClient webClient;

    public MyService(WebClient.Builder webClientBuilder) {
        this.webClient =
    webClientBuilder.baseUrl("https://example.org").build();
    }

    public Mono<Details> someRestCall(String name) {
        return this.webClient.get().uri("/{name}/details", name)
            .retrieve().bodyToMono(Details.class);
    }

}
```

36.1 WebClient Runtime

Spring Boot will auto-detect which `ClientHttpConnector` to use to drive `WebClient`, depending on the libraries available on the application classpath. For now, Reactor Netty and Jetty RS client are supported.

The `spring-boot-starter-webflux` starter depends on `io.projectreactor.netty:reactor-netty` by default, which brings both server and client implementations. If you choose to use Jetty as a reactive server instead, you should add a dependency on the Jetty Reactive HTTP client library, `org.eclipse.jetty:jetty-reactive-httpclient`. Using the same technology for server and client has its advantages, as it will automatically share HTTP resources between client and server.

Developers can override the resource configuration for Jetty and Reactor Netty by providing a custom `ReactorResourceFactory` or `JettyResourceFactory` bean - this will be applied to both clients and servers.

If you wish to override that choice for the client, you can define your own `ClientHttpConnector` bean and have full control over the client configuration.

You can learn more about the [WebClient configuration options in the Spring Framework reference documentation](#).

36.2 WebClient Customization

There are three main approaches to `WebClient` customization, depending on how broadly you want the customizations to apply.

To make the scope of any customizations as narrow as possible, inject the auto-configured `WebClient.Builder` and then call its methods as required. `WebClient.Builder` instances are stateful: Any change on the builder is reflected in all clients subsequently created with it. If you want to create several clients with the same builder, you can also consider cloning the builder with `WebClient.Builder other = builder.clone();`.

To make an application-wide, additive customization to all `WebClient.Builder` instances, you can declare `WebClientCustomizer` beans and change the `WebClient.Builder` locally at the point of injection.

Finally, you can fall back to the original API and use `WebClient.create()`. In that case, no auto-configuration or `WebClientCustomizer` is applied.

37. Validation

The method validation feature supported by Bean Validation 1.1 is automatically enabled as long as a JSR-303 implementation (such as Hibernate validator) is on the classpath. This lets bean methods be annotated with `javax.validation` constraints on their parameters and/or on their return value. Target classes with such annotated methods need to be annotated with the `@Validated` annotation at the type level for their methods to be searched for inline constraint annotations.

For instance, the following service triggers the validation of the first argument, making sure its size is between 8 and 10:

```
@Service  
@Validated
```

```
public class MyBean {

    public Archive findByCodeAndAuthor(@Size(min = 8, max = 10) String
code,
                                         Author author) {
        ...
    }
}
```

38. Sending Email

The Spring Framework provides an easy abstraction for sending email by using the `JavaMailSender` interface, and Spring Boot provides auto-configuration for it as well as a starter module.



See the [reference documentation](#) for a detailed explanation of how you can use `JavaMailSender`.

If `spring.mail.host` and the relevant libraries (as defined by `spring-boot-starter-mail`) are available, a default `JavaMailSender` is created if none exists. The sender can be further customized by configuration items from the `spring.mail` namespace. See [MailProperties](#) for more details.

In particular, certain default timeout values are infinite, and you may want to change that to avoid having a thread blocked by an unresponsive mail server, as shown in the following example:

```
spring.mail.properties.mail.smtp.connectiontimeout=5000
spring.mail.properties.mail.smtp.timeout=3000
spring.mail.properties.mail.smtp.writetimeout=5000
```

It is also possible to configure a `JavaMailSender` with an existing `Session` from JNDI:

```
spring.mail.jndi-name=mail/Session
```

When a `jndi-name` is set, it takes precedence over all other Session-related settings.

39. Distributed Transactions with JTA

Spring Boot supports distributed JTA transactions across multiple XA resources by using either an [Atomikos](#) or [Bitronix](#) embedded transaction manager. JTA transactions are also supported when deploying to a suitable Java EE Application Server.

When a JTA environment is detected, Spring's `JtaTransactionManager` is used to manage transactions. Auto-configured JMS, DataSource, and JPA beans are upgraded to support XA transactions. You can use standard Spring idioms, such as `@Transactional`, to participate in a distributed transaction. If you are within a JTA environment and still want to use local transactions, you can set the `spring.jta.enabled` property to `false` to disable the JTA auto-configuration.

39.1 Using an Atomikos Transaction Manager

[Atomikos](#) is a popular open source transaction manager which can be embedded into your Spring Boot application. You can use the `spring-boot-starter-jta-atomikos` Starter to pull in the appropriate Atomikos libraries. Spring Boot auto-configures Atomikos and ensures that appropriate `depends-on` settings are applied to your Spring beans for correct startup and shutdown ordering.

By default, Atomikos transaction logs are written to a `transaction-logs` directory in your application's home directory (the directory in which your application jar file resides). You can customize the location of this directory by setting a `spring.jta.log-dir` property in your `application.properties` file. Properties starting with `spring.jta.atomikos.properties` can also be used to customize the Atomikos `UserTransactionServiceImp`. See the [AtomikosProperties Javadoc](#) for complete details.



To ensure that multiple transaction managers can safely coordinate the same resource managers, each Atomikos instance must be configured with a unique ID. By default, this ID is the IP address of the machine on which Atomikos is running. To ensure uniqueness in production, you should configure the `spring.jta.transaction-manager-id` property with a different value for each instance of your application.

39.2 Using a Bitronix Transaction Manager

[Bitronix](#) is a popular open-source JTA transaction manager implementation. You can use the `spring-boot-starter-jta-bitronix` starter to add the appropriate Bitronix dependencies to your project. As with Atomikos, Spring Boot automatically configures Bitronix and post-processes your beans to ensure that startup and shutdown ordering is correct.

By default, Bitronix transaction log files (`part1.btm` and `part2.btm`) are written to a `transaction-logs` directory in your application home directory. You can customize the location of this directory by setting the `spring.jta.log-dir` property. Properties starting with `spring.jta.bitronix.properties` are also bound to the `bitronix.tm.Configuration` bean, allowing for complete customization. See the [Bitronix documentation](#) for details.



To ensure that multiple transaction managers can safely coordinate the same resource managers, each Bitronix instance must be configured with a unique ID. By default, this ID is the IP address of the machine on which Bitronix is running. To ensure uniqueness in production, you should configure the `spring.jta.transaction-manager-id` property with a different value for each instance of your application.

39.3 Using a Java EE Managed Transaction Manager

If you package your Spring Boot application as a `war` or `ear` file and deploy it to a Java EE application server, you can use your application server's built-in transaction manager. Spring Boot tries to auto-configure a transaction manager by looking at common JNDI locations

(`java:comp/UserTransaction`, `java:comp/TransactionManager`, and so on). If you use a transaction service provided by your application server, you generally also want to ensure that all resources are managed by the server and exposed over JNDI. Spring Boot tries to auto-configure JMS by looking for a `ConnectionFactory` at the JNDI path

(`java:/JmsXA` or `java:/XAConnectionFactory`), and you can use the `spring.datasource.jndi-name` [property](#) to configure your `DataSource`.

39.4 Mixing XA and Non-XA JMS Connections

When using JTA, the primary JMS `ConnectionFactory` bean is XA-aware and participates in distributed transactions. In some situations, you might want to process certain JMS messages by using a non-XA `ConnectionFactory`. For example, your JMS processing logic might take longer than the XA timeout.

If you want to use a non-XA `ConnectionFactory`, you can inject the `nonXaJmsConnectionFactory` bean rather than the `@Primary jmsConnectionFactory` bean. For consistency, the `jmsConnectionFactory` bean is also provided by using the bean alias `xaJmsConnectionFactory`.

The following example shows how to inject `ConnectionFactory` instances:

```
// Inject the primary (XA aware) ConnectionFactory
@Autowired
private ConnectionFactory defaultConnectionFactory;

// Inject the XA aware ConnectionFactory (uses the alias and injects the
// same as above)
@Autowired
@Qualifier("xaJmsConnectionFactory")
private ConnectionFactory xaConnectionFactory;

// Inject the non-XA aware ConnectionFactory
@Autowired
@Qualifier("nonXaJmsConnectionFactory")
private ConnectionFactory nonXaConnectionFactory;
```

39.5 Supporting an Alternative Embedded Transaction Manager

The `XAConnectionFactoryWrapper` and `XADatasourceWrapper` interfaces can be used to support alternative embedded transaction managers. The interfaces are responsible for wrapping `XAConnectionFactory` and `XADatasource` beans and exposing them as regular `ConnectionFactory` and `DataSource` beans, which transparently enroll in the distributed transaction. DataSource and JMS auto-configuration use JTA variants, provided you have a `JtaTransactionManager` bean and appropriate XA wrapper beans registered within your `ApplicationContext`.

The [BitronixXAConnectionFactoryWrapper](#) and [BitronixXADatasourceWrapper](#) provide good examples of how to write XA wrappers.

40. Hazelcast

If [Hazelcast](#) is on the classpath and a suitable configuration is found, Spring Boot auto-configures a `HazelcastInstance` that you can inject in your application.

If you define a `com.hazelcast.config.Config` bean, Spring Boot uses that. If your configuration defines an instance name, Spring Boot tries to locate an existing instance rather than creating a new one.

You could also specify the `hazelcast.xml` configuration file to use through configuration, as shown in the following example:

```
spring.hazelcast.config=classpath:config/my-hazelcast.xml
```

Otherwise, Spring Boot tries to find the Hazelcast configuration from the default locations: `hazelcast.xml` in the working directory or at the root of the classpath. We also check if the `hazelcast.config` system property is set. See the [Hazelcast documentation](#) for more details.

If `hazelcast-client` is present on the classpath, Spring Boot first attempts to create a client by checking the following configuration options:

- The presence of a `com.hazelcast.client.config.ClientConfig` bean.
- A configuration file defined by the `spring.hazelcast.config` property.
- The presence of the `hazelcast.client.config` system property.
- A `hazelcast-client.xml` in the working directory or at the root of the classpath.



41. Quartz Scheduler

Spring Boot offers several conveniences for working with the [Quartz scheduler](#), including the `spring-boot-starter-quartz` “Starter”. If Quartz is available, a `Scheduler` is auto-configured (through the `SchedulerFactoryBean` abstraction).

Beans of the following types are automatically picked up and associated with the `Scheduler`:

- `JobDetail`: defines a particular Job. `JobDetail` instances can be built with the `JobBuilder` API.
- `Calendar`.
- `Trigger`: defines when a particular job is triggered.

By default, an in-memory `JobStore` is used. However, it is possible to configure a JDBC-based store if a `DataSource` bean is available in your application and if the `spring.quartz.job-store-type` property is configured accordingly, as shown in the following example:

```
spring.quartz.job-store-type=jdbc
```

When the JDBC store is used, the schema can be initialized on startup, as shown in the following example:

```
spring.quartz.jdbc.initialize-schema=always
```



By default, the database is detected and initialized by using the standard scripts provided with the Quartz library. These scripts drop existing tables, deleting all triggers on every restart. It is also possible to provide a custom script by setting the `spring.quartz.jdbc.schema` property.

To have Quartz use a `DataSource` other than the application’s main `DataSource`, declare a `DataSource` bean, annotating its `@Bean` method with `@QuartzDataSource`. Doing so ensures that the Quartz-specific `DataSource` is used by both the `SchedulerFactoryBean` and for schema initialization.

By default, jobs created by configuration will not overwrite already registered jobs that have been read from a persistent job store. To enable overwriting existing job definitions set the `spring.quartz.overwrite-existing-jobs` property.

Quartz Scheduler configuration can be customized using `spring.quartz` properties and `SchedulerFactoryBeanCustomizer` beans, which allow programmatic `SchedulerFactoryBean` customization. Advanced Quartz configuration properties can be customized using `spring.quartz.properties.*`.



In particular, an `Executor` bean is not associated with the scheduler as Quartz offers a way to configure the scheduler via `spring.quartz.properties`. If you need to customize the task executor, consider implementing `SchedulerFactoryBeanCustomizer`.

Jobs can define setters to inject data map properties. Regular beans can also be injected in a similar manner, as shown in the following example:

```
public class SampleJob extends QuartzJobBean {

    private MyService myService;

    private String name;

    // Inject "MyService" bean
    public void setMyService(MyService myService) { ... }

    // Inject the "name" job data property
    public void setName(String name) { ... }

    @Override
    protected void executeInternal(JobExecutionContext context)
        throws JobExecutionException {
        ...
    }
}
```

42. Task Execution and Scheduling

In the absence of an `Executor` bean in the context, Spring Boot auto-configures a `ThreadPoolTaskExecutor` with sensible defaults that can be

automatically associated to asynchronous task execution (`@EnableAsync`) and Spring MVC asynchronous request processing.



If you have defined a custom **Executor** in the context, regular task execution (i.e. `@EnableAsync`) will use it transparently but the Spring MVC support will not be configured as it requires an **AsyncTaskExecutor** implementation (named `applicationTaskExecutor`). Depending on your target arrangement, you could change your **Executor** into a **ThreadPoolTaskExecutor** or define both a **ThreadPoolTaskExecutor** and an **AsyncConfigurer** wrapping your custom **Executor**.

The auto-configured **TaskExecutorBuilder** allows you to easily create instances that reproduce what the auto-configuration does by default.

The thread pool uses 8 core threads that can grow and shrink according to the load. Those default settings can be fine-tuned using the `spring.task.execution` namespace as shown in the following example:

```
spring.task.execution.pool.max-threads=16  
spring.task.execution.pool.queue-capacity=100  
spring.task.execution.pool.keep-alive=10s
```

This changes the thread pool to use a bounded queue so that when the queue is full (100 tasks), the thread pool increases to maximum 16 threads. Shrinking of the pool is more aggressive as threads are reclaimed when they are idle for 10 seconds (rather than 60 seconds by default).

A **ThreadPoolTaskScheduler** can also be auto-configured if need to be associated to scheduled task execution (`@EnableScheduling`). The thread pool uses one thread by default and those settings can be fine-tuned using the `spring.task.scheduling` namespace.

Both a **TaskExecutorBuilder** bean and a **TaskSchedulerBuilder** bean are made available in the context if a custom executor or scheduler needs to be created.

43. Spring boot

Integration

Spring Boot offers several conveniences for working with [Spring Integration](#), including the `spring-boot-starter-integration` "Starter". Spring Integration provides abstractions over messaging and also other transports such as HTTP, TCP, and others. If Spring Integration is available on your classpath, it is initialized through the `@EnableIntegration` annotation.

Spring Boot also configures some features that are triggered by the presence of additional Spring Integration modules. If `spring-integration-jmx` is also on the classpath, message processing statistics are published over JMX . If `spring-integration-jdbc` is available, the default database schema can be created on startup, as shown in the following line:

```
spring.integration.jdbc.initialize-schema=always
```

See the `IntegrationAutoConfiguration` and `IntegrationProperties` classes for more details.

By default, if a Micrometer `meterRegistry` bean is present, Spring Integration metrics will be managed by Micrometer. If you wish to use legacy Spring Integration metrics, add a `DefaultMetricsFactory` bean to the application context.

44. Spring boot Session

Spring Boot provides [Spring Session](#) auto-configuration for a wide range of data stores. When building a Servlet web application, the following stores can be auto-configured:

- JDBC
- Redis
- Hazelcast
- MongoDB

When building a reactive web application, the following stores can be auto-configured:

- Redis
- MongoDB

If a single Spring Session module is present on the classpath, Spring Boot uses that store implementation automatically. If you have more than one implementation, you must choose the `StoreType` that you wish to use to store the sessions. For instance, to use JDBC as the back-end store, you can configure your application as follows:

```
spring.session.store-type=jdbc
```



You can disable Spring Session by setting the `store-type` to `none`.

Each store has specific additional settings. For instance, it is possible to customize the name of the table for the JDBC store, as shown in the following example:

```
spring.session.jdbc.table-name=SESSIONS
```

For setting the timeout of the session you can use the `spring.session.timeout` property. If that property is not set, the auto-configuration falls back to the value of `server.servlet.session.timeout`.

45. Monitoring and Management over JMX

Java Management Extensions (JMX) provide a standard mechanism to monitor and manage applications. Spring Boot exposes the most suitable `MBeanServer` as a bean with an ID of `mbeanServer`. Any of your beans that are annotated with Spring JMX annotations (`@ManagedResource`, `@ManagedAttribute`, or `@ManagedOperation`) are exposed to it.

If your platform provides a standard `MBeanServer`, Spring Boot will use that and default to the VM `MBeanServer` if necessary. If all that fails, a new `MBeanServer` will be created.

See the `JmxAutoConfiguration` class for more details.

46. Testing

Spring Boot provides a number of utilities and annotations to help when testing your application. Test support is provided by two modules: `spring-boot-test` contains core items, and `spring-boot-test-autoconfigure` supports auto-configuration for tests.

Most developers use the `spring-boot-starter-test` “Starter”, which imports both Spring Boot test modules as well as JUnit, AssertJ, Hamcrest, and a number of other useful libraries.

46.1 Test Scope Dependencies

The `spring-boot-starter-test` “Starter” (in the `test` `scope`) contains the following provided libraries:

- [JUnit](#): The de-facto standard for unit testing Java applications.
- [Spring Test](#) & Spring Boot Test: Utilities and integration test support for Spring Boot applications.
- [AssertJ](#): A fluent assertion library.
- [Hamcrest](#): A library of matcher objects (also known as constraints or predicates).
- [Mockito](#): A Java mocking framework.
- [JSONassert](#): An assertion library for JSON.
- [JsonPath](#): XPath for JSON.

We generally find these common libraries to be useful when writing tests. If these libraries do not suit your needs, you can add additional test dependencies of your own.

46.2 Testing Spring Applications

One of the major advantages of dependency injection is that it should make your code easier to unit test. You can instantiate objects by using the `new` operator without even involving Spring. You can also use *mock objects* instead of real dependencies.

Often, you need to move beyond unit testing and start integration testing (with a Spring `ApplicationContext`). It is useful to be able to perform integration testing without requiring deployment of your application or needing to connect to other infrastructure.

The Spring Framework includes a dedicated test module for such integration testing. You can declare a dependency directly to `org.springframework:spring-test` or use the `spring-boot-starter-test` “Starter” to pull it in transitively.

If you have not used the `spring-test` module before, you should start by reading the [relevant section](#) of the Spring Framework reference documentation.

46.3 Testing Spring Boot Applications

A Spring Boot application is a Spring `ApplicationContext`, so nothing very special has to be done to test it beyond what you would normally do with a vanilla Spring context.



External properties, logging, and other features of Spring Boot are installed in the context by default only if you use `SpringApplication` to create it.

Spring Boot provides a `@SpringBootTest` annotation, which can be used as an alternative to the standard `spring-test @ContextConfiguration` annotation when you need Spring Boot features. The annotation works by [creating the `ApplicationContext` used in your tests through `SpringApplication`](#). In addition to `@SpringBootTest` a number of other annotations are also provided for [testing more specific slices](#) of an application.



If you are using JUnit 4, don't forget to also add `@RunWith(SpringRunner.class)` to your test, otherwise the annotations will be ignored. If you are using JUnit 5, there's no need to add the equivalent `@ExtendWith(SpringExtension.class)` as `@SpringBootTest` and the other `@...Test` annotations are already annotated with it.

By default, `@SpringBootTest` will not start a server. You can use the `webEnvironment` attribute of `@SpringBootTest` to further refine how your tests run:

- `MOCK`(Default) : Loads a web `ApplicationContext` and provides a mock web environment. Embedded servers are not started when using this annotation. If a web environment is not available on your classpath, this mode transparently falls back to creating a regular non-web `ApplicationContext`. It can be used in conjunction with `@AutoConfigureMockMvc` or `@AutoConfigureWebTestClient` for mock-based testing of your web application.
- `RANDOM_PORT`: Loads a `WebServerApplicationContext` and provides a real web environment. Embedded servers are started and listen on a random port.
- `DEFINED_PORT`: Loads a `WebServerApplicationContext` and provides a real web environment. Embedded servers are started and listen on a defined port (from your `application.properties`) or on the default port of `8080`.
- `NONE`: Loads an `ApplicationContext` by using `SpringApplication` but does not provide *any* web environment (mock or otherwise).



If your test is `@Transactional`, it rolls back the transaction at the end of each test method by default. However, as using this arrangement with either `RANDOM_PORT` or `DEFINED_PORT` implicitly provides a real servlet environment, the HTTP client and server run in separate threads and, thus, in separate transactions. Any transaction initiated on the server does not roll back in this case.



`@SpringBootTest` with `webEnvironment =`

`WebEnvironment.RANDOM_PORT` will also start the management server on a separate random port if your application uses a different port for the management server.

46.3.1 Detecting Web Application Type

If Spring MVC is available, a regular MVC-based application context is configured. If you have only Spring WebFlux, we'll detect that and configure a WebFlux-based application context instead.

If both are present, Spring MVC takes precedence. If you want to test a reactive web application in this scenario, you must set the `spring.main.web-application-type` property:

```
@RunWith(SpringRunner.class)
@SpringBootTest(properties = "spring.main.web-application-type=reactive")
public class MyWebFluxTests { ... }
```

46.3.2 Detecting Test Configuration

If you are familiar with the Spring Test Framework, you may be used to using `@ContextConfiguration(classes=...)` in order to specify which Spring `@Configuration` to load. Alternatively, you might have often used nested `@Configuration` classes within your test.

When testing Spring Boot applications, this is often not required. Spring Boot's `@*Test` annotations search for your primary configuration automatically whenever you do not explicitly define one.

The search algorithm works up from the package that contains the test until it finds a class annotated with `@SpringBootApplication` or `@SpringBootConfiguration`. As long as you [structured your code](#) in a sensible way, your main configuration is usually found.



If you use a [test annotation to test a more specific slice of your application](#), you should avoid adding configuration settings that are specific to a particular area on the [main method's application class](#).

The underlying component scan configuration of `@SpringBootApplication` defines exclude filters that are used to make sure slicing works as expected. If you are using an explicit `@ComponentScan` directive on your `@SpringBootApplication`-annotated class, be aware that those filters will be disabled. If you are using slicing, you should

define them again.

If you want to customize the primary configuration, you can use a nested `@TestConfiguration` class. Unlike a nested `@Configuration` class, which would be used instead of your application's primary configuration, a nested `@TestConfiguration` class is used in addition to your application's primary configuration.



Spring's test framework caches application contexts between tests. Therefore, as long as your tests share the same configuration (no matter how it is discovered), the potentially time-consuming process of loading the context happens only once.

46.3.3 Excluding Test Configuration

If your application uses component scanning (for example, if you use `@SpringBootApplication` or `@ComponentScan`), you may find top-level configuration classes that you created only for specific tests accidentally get picked up everywhere.

As we [have seen earlier](#), `@TestConfiguration` can be used on an inner class of a test to customize the primary configuration. When placed on a top-level class, `@TestConfiguration` indicates that classes in `src/test/java` should not be picked up by scanning. You can then import that class explicitly where it is required, as shown in the following example:

```
@RunWith(SpringRunner.class)
@SpringBootTest
@Import(MyTestsConfiguration.class)
public class MyTests {

    @Test
    public void exampleTest() {
        ...
    }
}
```



If you directly use `@ComponentScan` (that is, not

through `@SpringBootApplication`) you need to register the `TypeExcludeFilter` with it. See [the Javadoc](#) for details.

46.3.4 Testing with a mock environment

By default, `@SpringBootTest` does not start the server. If you have web endpoints that you want to test against this mock environment, you can additionally configure `MockMvc` as shown in the following example:

```
import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockM
vc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;

import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc
public class MockMvcExampleTests {

    @Autowired
    private MockMvc mvc;

    @Test
    public void exampleTest() throws Exception {
        this.mvc.perform(get("/")).andExpect(status().isOk())
            .andExpect(content().string("Hello World"));
    }
}
```



If you want to focus only on the web layer and not start a

complete `ApplicationContext`, consider [using `@WebMvcTest` instead](#).

Alternatively, you can configure a `WebTestClient` as shown in the following example:

```
import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.web.reactive.AutoConfigureWebT
estClient;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.reactive.server.WebTestClient;

@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureWebTestClient
public class MockWebTestClientExampleTests {

    @Autowired
    private WebTestClient webClient;

    @Test
    public void exampleTest() {

        this.webClient.get().uri("/")
            .exchange()
            .expectStatus().isOk()
            .expectBody(String.class).isEqualTo("Hello
World");
    }
}
```

46.3.5 Testing with a running server

If you need to start a full running server, we recommend that you use random ports. If you

use `@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)`, an available port is picked at random each time your test runs.

The `@LocalServerPort` annotation can be used to [inject the actual port used](#) into your test. For convenience, tests that need to make REST calls to the started server can additionally `@Autowire` a `WebTestClient`, which

resolves relative links to the running server and comes with a dedicated API for verifying responses, as shown in the following example:

```
import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import
org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.reactive.server.WebTestClient;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class RandomPortWebTestClientExampleTests {

    @Autowired
    private WebTestClient webClient;

    @Test
    public void exampleTest() {

        this.webClient.get().uri("/").exchange().expectStatus().isOk()
            .expectBody(String.class).isEqualTo("Hello
World");
    }

}
```

This setup requires `spring-webflux` on the classpath. If you can't or won't add webflux, Spring Boot also provides a `TestRestTemplate` facility:

```
import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import
org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.test.context.junit4.SpringRunner;

import static org.assertj.core.api.Assertions.assertThat;

@RunWith(SpringRunner.class)
```

```

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class RandomPortTestRestTemplateExampleTests {

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    public void exampleTest() {
        String body = this.restTemplate.getForObject("/", 
String.class);
        assertThat(body).isEqualTo("Hello World");
    }

}

```

46.3.6 Using JMX

As the test context framework caches context, JMX is disabled by default to prevent identical components to register on the same domain. If such test needs access to an `MBeanServer`, consider marking it dirty as well:

```

@RunWith(SpringRunner.class)
@SpringBootTest(properties = "spring.jmx.enabled=true")
@DirtiesContext
public class SampleJmxTests {

    @Autowired
    private MBeanServer mBeanServer;

    @Test
    public void exampleTest() {
        // ...
    }

}

```

46.3.7 Mocking and Spying Beans

When running tests, it is sometimes necessary to mock certain components within your application context. For example, you may have a facade over some remote service that is unavailable during development. Mocking can also be useful when you want to simulate failures that might be hard to trigger in a real environment.

Spring Boot includes a `@MockBean` annotation that can be used to define a Mockito mock for a bean inside your `ApplicationContext`. You can use the

annotation to add new beans or replace a single existing bean definition. The annotation can be used directly on test classes, on fields within your test, or on `@Configuration` classes and fields. When used on a field, the instance of the created mock is also injected. Mock beans are automatically reset after each test method.



If your test uses one of Spring Boot's test annotations (such as `@SpringBootTest`), this feature is automatically enabled. To use this feature with a different arrangement, a listener must be explicitly added, as shown in the following example:

```
@TestExecutionListeners(MockitoTestExecutionListener.class)
```

The following example replaces an existing `RemoteService` bean with a mock implementation:

```
import org.junit.*;
import org.junit.runner.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.context.*;
import org.springframework.boot.test.mock.mockito.*;
import org.springframework.test.context.junit4.*;

import static org.assertj.core.api.Assertions.*;
import static org.mockito.BDDMockito.*;

@RunWith(SpringRunner.class)
@SpringBootTest
public class MyTests {

    @MockBean
    private RemoteService remoteService;

    @Autowired
    private Reverser reverser;

    @Test
    public void exampleTest() {
        // RemoteService has been injected into the reverser bean
        given(this.remoteService.someCall()).willReturn("mock");
        String reverse = reverser.reverseSomeCall();
        assertThat(reverse).isEqualTo("kcom");
    }
}
```

}



`@MockBean` cannot be used to mock the behavior of a bean that's exercised during application context refresh. By the time the test is executed, the application context refresh has completed and it is too late to configure the mocked behavior. We recommend using a `@Bean` method to create and configure the mock in this situation.

Additionally, you can use `@SpyBean` to wrap any existing bean with a Mockito `spy`. See the [Javadoc](#) for full details.



While Spring's test framework caches application contexts between tests and reuses a context for tests sharing the same configuration, the use of `@MockBean` or `@SpyBean` influences the cache key, which will most likely increase the number of contexts.



If you are using `@SpyBean` to spy on a bean with `@Cacheable` methods that refer to parameters by name, your application must be compiled with `-parameters`. This ensures that the parameter names are available to the caching infrastructure once the bean has been spied upon.

46.3.8 Auto-configured Tests

Spring Boot's auto-configuration system works well for applications but can sometimes be a little too much for tests. It often helps to load only the parts of the configuration that are required to test a "slice" of your application. For example, you might want to test that Spring MVC controllers are mapping URLs correctly, and you do not want to involve database calls in those tests, or you might want to test JPA entities, and you are not interested in the web layer when those tests run.

The `spring-boot-test-autoconfigure` module includes a number of annotations that can be used to automatically configure such "slices". Each of them works in a similar way, providing a `@...Test` annotation that loads the `ApplicationContext` and one or more `@AutoConfigure...` annotations that can be used to customize auto-configuration settings.



Each slice restricts component scan to appropriate components and loads a very restricted set of auto-configuration classes. If you need to exclude one of them, most `@...Test` annotations provide an `excludeAutoConfiguration` attribute. Alternatively, you can use `@ImportAutoConfiguration#exclude`.



Including multiple “slices” by using several `@...Test` annotations in one test is not supported. If you need multiple “slices”, pick one of the `@...Test` annotations and include the `@AutoConfigure...` annotations of the other “slices” by hand.



It is also possible to use the `@AutoConfigure...` annotations with the standard `@SpringBootTest` annotation. You can use this combination if you are not interested in “slicing” your application but you want some of the auto-configured test beans.

46.3.9 Auto-configured JSON Tests

To test that object JSON serialization and deserialization is working as expected, you can use the `@JsonTest` annotation. `@JsonTest` auto-configures the available supported JSON mapper, which can be one of the following libraries:

- Jackson `ObjectMapper`, any `@JsonComponent` beans and any Jackson `Modules`
- `Gson`
- `Jsonb`



A list of the auto-configurations that are enabled by `@JsonTest` can be [found in the appendix](#).

If you need to configure elements of the auto-configuration, you can use the `@AutoConfigureJsonTesters` annotation.

Spring Boot includes AssertJ-based helpers that work with the JSONAssert and JsonPath libraries to check that JSON appears as expected.

The `JacksonTester`, `GsonTester`, `JsonbTester`, and `BasicJsonTester` classes can be used for Jackson, Gson, Jsonb, and Strings respectively. Any helper fields on the test class can be `@Autowired` when using `@JsonTest`. The following example shows a test class for Jackson:

```
import org.junit.*;
import org.junit.runner.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.autoconfigure.json.*;
import org.springframework.boot.test.context.*;
import org.springframework.boot.test.json.*;
import org.springframework.test.context.junit4.*;

import static org.assertj.core.api.Assertions.*;

@RunWith(SpringRunner.class)
@JsonTest
public class MyJsonTests {

    @Autowired
    private JacksonTester<VehicleDetails> json;

    @Test
    public void testSerialize() throws Exception {
        VehicleDetails details = new VehicleDetails("Honda",
"Civic");
        // Assert against a `*.json` file in the same package as the
test
        assertThat(this.json.write(details)).isEqualToJson("expected.json");
        // Or use JSON path based assertions
        assertThat(this.json.write(details)).hasJsonPathStringValue("@.make");
        assertThat(this.json.write(details)).extractingJsonPathStringValue
("@.make")
            .isEqualTo("Honda");
    }

    @Test
    public void testDeserialize() throws Exception {
        String content = "{\"make\":\"Ford\", \"model\":\"Focus\"}";
    }
}
```

```

        assertThat(this.json.parse(content))
            .isEqualTo(new VehicleDetails("Ford",
"Focus"));

        assertThat(this.json.parseObject(content).getMake()).isEqualTo("Fo
rd");
    }

}

```



JSON helper classes can also be used directly in standard unit tests. To do so, call the `initFields` method of the helper in your `@Before` method if you do not use `@JsonTest`.

If you're using Spring Boot's AssertJ-based helpers to assert on a number value at a given JSON path, you might not be able to use `isEqualTo` depending on the type. Instead, you can use AssertJ's `satisfies` to assert that the value matches the given condition. For instance, the following example asserts that the actual number is a float value close to `0.15` within an offset of `0.01`.

```

assertThat(json.write(message))
    .extractingJsonPathNumberValue("@.test.numberValue")
    .satisfies(number) ->
assertThat(number.floatValue()).isCloseTo(0.15f, within(0.01f)));

```

46.3.10 Auto-configured Spring MVC Tests

To test whether Spring MVC controllers are working as expected, use the `@WebMvcTest` annotation. `@WebMvcTest` auto-configures the Spring MVC infrastructure and limits scanned beans to `@Controller`, `@ControllerAdvice`, `@JsonComponent`, `Converter`, `GenericConverter`, `Filter`, `WebMvcConfigurer`, and `HandlerMethodArgumentResolver`. Regular `@Component` beans are not scanned when using this annotation.



A list of the auto-configuration settings that are enabled by `@WebMvcTest` can be [found in the appendix](#).



If you need to register extra components, such as the Jackson **Module**, you can import additional configuration classes by using **@Import** on your test.

Often, **@WebMvcTest** is limited to a single controller and is used in combination with **@MockBean** to provide mock implementations for required collaborators.

@WebMvcTest also auto-configures **MockMvc**. Mock MVC offers a powerful way to quickly test MVC controllers without needing to start a full HTTP server.



You can also auto-configure **MockMvc** in a non-**@WebMvcTest** (such as **@SpringBootTest**) by annotating it with **@AutoConfigureMockMvc**. The following example uses **MockMvc**:

```
import org.junit.*;
import org.junit.runner.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.autoconfigure.web.servlet.*;
import org.springframework.boot.test.mock.mockito.*;

import static org.assertj.core.api.Assertions.*;
import static org.mockito.BDDMockito.*;
import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@RunWith(SpringRunner.class)
@WebMvcTest(UserVehicleController.class)
public class MyControllerTests {

    @Autowired
    private MockMvc mvc;

    @MockBean
    private UserVehicleService userVehicleService;

    @Test
    public void testExample() throws Exception {
        given(this.userVehicleService.getVehicleDetails("sboot"))
    }
}
```

```

        .willReturn(new VehicleDetails("Honda",
        "Civic"));

    this.mvc.perform(get("/sboot/vehicle").accept(MediaType.TEXT_PLAIN))
        .andExpect(status().isOk()).andExpect(content().string("Honda
        Civic"));
    }

}

```



If you need to configure elements of the auto-configuration (for example, when servlet filters should be applied) you can use attributes in the `@AutoConfigureMockMvc` annotation.

If you use HtmlUnit or Selenium, auto-configuration also provides an `HTMLUnit` `WebClient` bean and/or a `WebDriver` bean. The following example uses HtmlUnit:

```

import com.gargoylesoftware.htmlunit.*;
import org.junit.*;
import org.junit.runner.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.autoconfigure.web.servlet.*;
import org.springframework.boot.test.mock.mockito.*;

import static org.assertj.core.api.Assertions.*;
import static org.mockito.BDDMockito.*;

@RunWith(SpringRunner.class)
@WebMvcTest(UserVehicleController.class)
public class MyHtmlUnitTests {

    @Autowired
    private WebClient webClient;

    @MockBean
    private UserVehicleService userVehicleService;

    @Test
    public void testExample() throws Exception {
        given(this.userVehicleService.getVehicleDetails("sboot"))

```

```

        .willReturn(new VehicleDetails("Honda",
"Civic"));
    HtmlPage page =
this.webClient.getPage("/sboot/vehicle.html");

    assertThat(page.getBody().getTextContent()).isEqualTo("Honda
Civic");
}
}

```



By default, Spring Boot puts `WebDriver` beans in a special “scope” to ensure that the driver exits after each test and that a new instance is injected. If you do not want this behavior, you can add `@Scope("singleton")` to your `WebDriver @Bean` definition.



The `webDriver` scope created by Spring Boot will replace any user defined scope of the same name. If you define your own `webDriver` scope you may find it stops working when you use `@WebMvcTest`.

If you have Spring Security on the classpath, `@WebMvcTest` will also scan `WebSecurityConfigurer` beans. Instead of disabling security completely for such tests, you can use Spring Security’s test support. More details on how to use Spring Security’s `MockMvc` support can be found in this [Chapter 80, Testing With Spring Security](#) how-to section.



Sometimes writing Spring MVC tests is not enough; Spring Boot can help you run [full end-to-end tests with an actual server](#).

46.3.11 Auto-configured Spring WebFlux Tests

To test that `Spring WebFlux` controllers are working as expected, you can use the `@WebFluxTest` annotation. `@WebFluxTest` auto-configures the Spring WebFlux infrastructure and limits scanned beans to `@Controller`, `@ControllerAdvice`, `@JsonComponent`, `Converter`, `GenericCon`

`verter`, and `WebFluxConfigurer`. Regular `@Component` beans are not scanned when the `@WebFluxTest` annotation is used.



A list of the auto-configurations that are enabled by `@WebFluxTest` can be [found in the appendix](#).



If you need to register extra components, such as Jackson `Module`, you can import additional configuration classes using `@Import` on your test.

Often, `@WebFluxTest` is limited to a single controller and used in combination with the `@MockBean` annotation to provide mock implementations for required collaborators.

`@WebFluxTest` also auto-configures `WebTestClient`, which offers a powerful way to quickly test WebFlux controllers without needing to start a full HTTP server.



You can also auto-configure `WebTestClient` in a non-`@WebFluxTest` (such as `@SpringBootTest`) by annotating it with `@AutoConfigureWebTestClient`. The following example shows a class that uses both `@WebFluxTest` and a `WebTestClient`:

```
import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.reactive.WebFluxTest;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.reactive.server.WebTestClient;

@RunWith(SpringRunner.class)
@WebFluxTest(UserVehicleController.class)
public class MyControllerTests {
```

```

@Autowired
private WebTestClient webClient;

@MockBean
private UserVehicleService userVehicleService;

@Test
public void testExample() throws Exception {
    given(this.userVehicleService.getVehicleDetails("sboot"))
        .willReturn(new VehicleDetails("Honda",
"Civic"));

    this.webClient.get().uri("/sboot/vehicle").accept(MediaType.TEXT_PLAIN)
        .exchange()
        .expectStatus().isOk()
        .expectBody(String.class).isEqualTo("Honda
Civic");
}

}

```



This setup is only supported by WebFlux applications as using `WebTestClient` in a mocked web application only works with WebFlux at the moment.



`@WebFluxTest` cannot detect routes registered via the functional web framework. For testing `RouterFunction` beans in the context, consider importing your `RouterFunction` yourself via `@Import` or using `@SpringBootTest`.



`@WebFluxTest` cannot detect custom security configuration registered via a `@Bean` of type `SecurityWebFilterChain`. To include that in your test, you will need to import the configuration that registers the bean via `@Import` or use `@SpringBootTest`.



Sometimes writing Spring WebFlux tests is not enough; Spring Boot can help you run [full end-to-end tests with an actual server](#).

46.3.12 Auto-configured Data JPA Tests

You can use the `@DataJpaTest` annotation to test JPA applications. By default, it configures an in-memory embedded database, scans for `@Entity` classes, and configures Spring Data JPA repositories. Regular `@Component` beans are not loaded into the `ApplicationContext`.



A list of the auto-configuration settings that are enabled by `@DataJpaTest` can be [found in the appendix](#).

By default, data JPA tests are transactional and roll back at the end of each test. See the [relevant section](#) in the Spring Framework Reference Documentation for more details. If that is not what you want, you can disable transaction management for a test or for the whole class as follows:

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@RunWith(SpringRunner.class)
@DataJpaTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
public class ExampleNonTransactionalTests {

}
```

Data JPA tests may also inject a `TestEntityManager` bean, which provides an alternative to the standard JPA `EntityManager` that is specifically designed for tests. If you want to use `TestEntityManager` outside of `@DataJpaTest` instances, you can also use the `@AutoConfigureTestEntityManager` annotation. A `JdbcTemplate` is also available if you need that. The following example shows the `@DataJpaTest` annotation in use:

```

import org.junit.*;
import org.junit.runner.*;
import org.springframework.boot.test.autoconfigure.orm.jpa.*;

import static org.assertj.core.api.Assertions.*;

@RunWith(SpringRunner.class)
@DataJpaTest
public class ExampleRepositoryTests {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private UserRepository repository;

    @Test
    public void testExample() throws Exception {
        this.entityManager.persist(new User("sboot", "1234"));
        User user = this.repository.findByUsername("sboot");
        assertThat(user.getUsername()).isEqualTo("sboot");
        assertThat(user.getPassword()).isEqualTo("1234");
    }
}

```

In-memory embedded databases generally work well for tests, since they are fast and do not require any installation. If, however, you prefer to run tests against a real database you can use the `@AutoConfigureTestDatabase` annotation, as shown in the following example:

```

@RunWith(SpringRunner.class)
@DataJpaTest
@AutoConfigureTestDatabase(replace=Replace.NONE)
public class ExampleRepositoryTests {

    // ...

}

```

46.3.13 Auto-configured JDBC Tests

`@JdbcTest` is similar to `@DataJpaTest` but is for tests that only require a `DataSource` and do not use Spring Data JDBC. By default, it configures an

in-memory embedded database and a `JdbcTemplate`.

Regular `@Component` beans are not loaded into the `ApplicationContext`.



A list of the auto-configurations that are enabled by `@JdbcTest` can be [found in the appendix](#).

By default, JDBC tests are transactional and roll back at the end of each test. See the [relevant section](#) in the Spring Framework Reference Documentation for more details. If that is not what you want, you can disable transaction management for a test or for the whole class, as follows:

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.autoconfigure.jdbc.JdbcTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@RunWith(SpringRunner.class)
@JdbcTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
public class ExampleNonTransactionalTests {

}
```

If you prefer your test to run against a real database, you can use the `@AutoConfigureTestDatabase` annotation in the same way as for `DataJpaTest`. (See "Section 46.3.12, “Auto-configured Data JPA Tests”".)

46.3.14 Auto-configured Data JDBC Tests

`@DataJdbcTest` is similar to `@JdbcTest` but is for tests that use Spring Data JDBC repositories. By default, it configures an in-memory embedded database, a `JdbcTemplate`, and Spring Data JDBC repositories.

Regular `@Component` beans are not loaded into the `ApplicationContext`.



A list of the auto-configurations that are enabled by `@DataJdbcTest` can be [found in](#)

[the appendix](#).

By default, Data JDBC tests are transactional and roll back at the end of each test. See the [relevant section](#) in the Spring Framework Reference Documentation for more details. If that is not what you want, you can disable transaction management for a test or for the whole test class as [shown in the JDBC example](#).

If you prefer your test to run against a real database, you can use the `@AutoConfigureTestDatabase` annotation in the same way as for `DataJpaTest`. (See "[Section 46.3.12, “Auto-configured Data JPA Tests”](#)".)

46.3.15 Auto-configured jOOQ Tests

You can use `@JooqTest` in a similar fashion as `@JdbcTest` but for jOOQ-related tests. As jOOQ relies heavily on a Java-based schema that corresponds with the database schema, the existing `DataSource` is used. If you want to replace it with an in-memory database, you can use `@AutoConfigureTestDatabase` to override those settings. (For more about using jOOQ with Spring Boot, see "[Section 31.6, “Using jOOQ”](#)", earlier in this chapter.)

Regular `@Component` beans are not loaded into the `ApplicationContext`.



A list of the auto-configuration that are enabled by `@JooqTest` can be [found in the appendix](#).

`@JooqTest` configures a `DSLContext`. Regular `@Component` beans are not loaded into the `ApplicationContext`. The following example shows the `@JooqTest` annotation in use:

```
import org.jooq.DSLContext;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.autoconfigure.jooq.JooqTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@JooqTest
public class ExampleJooqTests {

    @Autowired
```

```
    private DSLContext dslContext;  
}
```

JOOQ tests are transactional and roll back at the end of each test by default. If that is not what you want, you can disable transaction management for a test or for the whole test class as [shown in the JDBC example](#).

46.3.16 Auto-configured Data MongoDB Tests

You can use `@DataMongoTest` to test MongoDB applications. By default, it configures an in-memory embedded MongoDB (if available), configures a `MongoTemplate`, scans for `@Document` classes, and configures Spring Data MongoDB repositories. Regular `@Component` beans are not loaded into the `ApplicationContext`. (For more about using MongoDB with Spring Boot, see "[Section 32.2, “MongoDB”](#)", earlier in this chapter.)



A list of the auto-configuration settings that are enabled by `@DataMongoTest` can be [found in the appendix](#).

The following class shows the `@DataMongoTest` annotation in use:

```
import org.junit.runner.RunWith;  
import org.springframework.beans.factory.annotation.Autowired;  
import  
org.springframework.boot.test.autoconfigure.mongo.DataMongoTest;  
import org.springframework.data.mongodb.core.MongoTemplate;  
import org.springframework.test.context.junit4.SpringRunner;  
  
@RunWith(SpringRunner.class)  
@DataMongoTest  
public class ExampleDataMongoTests {  
  
    @Autowired  
    private MongoTemplate mongoTemplate;  
  
    //  
}
```

In-memory embedded MongoDB generally works well for tests, since it is fast and does not require any developer installation. If, however, you prefer to run

tests against a real MongoDB server, you should exclude the embedded MongoDB auto-configuration, as shown in the following example:

```
import org.junit.runner.RunWith;
import
org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongoAutoCon
figuration;
import
org.springframework.boot.test.autoconfigure.data.mongo.DataMongoTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@DataMongoTest(excludeAutoConfiguration =
EmbeddedMongoAutoConfiguration.class)
public class ExampleDataMongoNonEmbeddedTests {

}
```

46.3.17 Auto-configured Data Neo4j Tests

You can use `@DataNeo4jTest` to test Neo4j applications. By default, it uses an in-memory embedded Neo4j (if the embedded driver is available), scans for `@NodeEntity` classes, and configures Spring Data Neo4j repositories. Regular `@Component` beans are not loaded into the `ApplicationContext`. (For more about using Neo4J with Spring Boot, see "[Section 32.3, “Neo4j”](#)", earlier in this chapter.)



A list of the auto-configuration settings that are enabled by `@DataNeo4jTest` can be [found in the appendix](#).

The following example shows a typical setup for using Neo4J tests in Spring Boot:

```
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.data.neo4j.DataNeo4jTest;
import org.springframework.test.context.junit4.SpringRunner;

RunWith(SpringRunner.class)
@DataNeo4jTest
public class ExampleDataNeo4jTests {
```

```
    @Autowired  
    private YourRepository repository;  
  
    //  
}
```

By default, Data Neo4j tests are transactional and roll back at the end of each test. See the [relevant section](#) in the Spring Framework Reference Documentation for more details. If that is not what you want, you can disable transaction management for a test or for the whole class, as follows:

```
import org.junit.Test;  
import org.junit.runner.RunWith;  
import  
org.springframework.boot.test.autoconfigure.data.neo4j.DataNeo4jTest;  
import org.springframework.test.context.junit4.SpringRunner;  
import org.springframework.transaction.annotation.Propagation;  
import org.springframework.transaction.annotation.Transactional;  
  
@RunWith(SpringRunner.class)  
@DataNeo4jTest  
@Transactional(propagation = Propagation.NOT_SUPPORTED)  
public class ExampleNonTransactionalTests {  
  
}
```

46.3.18 Auto-configured Data Redis Tests

You can use `@DataRedisTest` to test Redis applications. By default, it scans for `@RedisHash` classes and configures Spring Data Redis repositories. Regular `@Component` beans are not loaded into the `ApplicationContext`. (For more about using Redis with Spring Boot, see "[Section 32.1, “Redis”](#)", earlier in this chapter.)



A list of the auto-configuration settings that are enabled by `@DataRedisTest` can be [found in the appendix](#).

The following example shows the `@DataRedisTest` annotation in use:

```
import org.junit.runner.RunWith;  
import org.springframework.beans.factory.annotation.Autowired;
```

```

import
org.springframework.boot.test.autoconfigure.data.redis.DataRedisTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@DataRedisTest
public class ExampleDataRedisTests {

    @Autowired
    private YourRepository repository;

    //
}

```

46.3.19 Auto-configured Data LDAP Tests

You can use `@DataLdapTest` to test LDAP applications. By default, it configures an in-memory embedded LDAP (if available), configures an `LdapTemplate`, scans for `@Entry` classes, and configures Spring Data LDAP repositories. Regular `@Component` beans are not loaded into the `ApplicationContext`. (For more about using LDAP with Spring Boot, see ["Section 32.9, “LDAP”"](#), earlier in this chapter.)



A list of the auto-configuration settings that are enabled by `@DataLdapTest` can be [found in the appendix](#).

The following example shows the `@DataLdapTest` annotation in use:

```

import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.ldap.DataLdapTest;
import org.springframework.ldap.core.LdapTemplate;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@DataLdapTest
public class ExampleDataLdapTests {

    @Autowired
    private LdapTemplate ldapTemplate;

    //
}

```

```
}
```

In-memory embedded LDAP generally works well for tests, since it is fast and does not require any developer installation. If, however, you prefer to run tests against a real LDAP server, you should exclude the embedded LDAP auto-configuration, as shown in the following example:

```
import org.junit.runner.RunWith;
import
org.springframework.boot.autoconfigure.ldap.embedded.EmbeddedLdapAutoConfiguration;
import org.springframework.boot.test.autoconfigure.data.ldap.DataLdapTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@DataLdapTest(excludeAutoConfiguration =
EmbeddedLdapAutoConfiguration.class)
public class ExampleDataLdapNonEmbeddedTests {

}
```

46.3.20 Auto-configured REST Clients

You can use the `@RestClientTest` annotation to test REST clients. By default, it auto-configures Jackson, GSON, and Jsonb support, configures a `RestTemplateBuilder`, and adds support for `MockRestServiceServer`. Regular `@Component` beans are not loaded into the `ApplicationContext`.



A list of the auto-configuration settings that are enabled by `@RestClientTest` can be [found in the appendix](#).

The specific beans that you want to test should be specified by using the `value` or `components` attribute of `@RestClientTest`, as shown in the following example:

```
@RunWith(SpringRunner.class)
@RestClientTest(RemoteVehicleDetailsService.class)
public class ExampleRestClientTest {

    @Autowired
    private RemoteVehicleDetailsService service;
```

```

    @Autowired
    private MockRestServiceServer server;

    @Test
    public void
getVehicleDetailsWhenResultIsSuccessShouldReturnDetails()
        throws Exception {
    this.server.expect(requestTo("/greet/details"))
        .andRespond(withSuccess("hello",
    MediaType.TEXT_PLAIN));
    String greeting = this.service.callRestService();
    assertThat(greeting).isEqualTo("hello");
}
}

```

46.3.21 Auto-configured Spring REST Docs Tests

You can use the `@AutoConfigureRestDocs` annotation to use [Spring REST Docs](#) in your tests with Mock MVC, REST Assured, or WebTestClient. It removes the need for the JUnit rule in Spring REST Docs.

`@AutoConfigureRestDocs` can be used to override the default output directory (`target/generated-snippets` if you are using Maven or `build/generated-snippets` if you are using Gradle). It can also be used to configure the host, scheme, and port that appears in any documented URIs.

Auto-configured Spring REST Docs Tests with Mock MVC

`@AutoConfigureRestDocs` customizes the `MockMvc` bean to use Spring REST Docs. You can inject it by using `@Autowired` and use it in your tests as you normally would when using Mock MVC and Spring REST Docs, as shown in the following example:

```

import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;

import static
org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.document;

```

```

import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@RunWith(SpringRunner.class)
@WebMvcTest(UserController.class)
@AutoConfigureRestDocs
public class UserDocumentationTests {

    @Autowired
    private MockMvc mvc;

    @Test
    public void listUsers() throws Exception {

        this.mvc.perform(get("/users").accept(MediaType.TEXT_PLAIN))
            .andExpect(status().isOk())
            .andDo(document("list-users"));
    }

}

```

If you require more control over Spring REST Docs configuration than offered by the attributes of `@AutoConfigureRestDocs`, you can use a `RestDocsMockMvcConfigurationCustomizer` bean, as shown in the following example:

```

@Configuration
static class CustomizationConfiguration
    implements RestDocsMockMvcConfigurationCustomizer {

    @Override
    public void customize(MockMvcRestDocumentationConfigurer
configurer) {

        configurer.snippets().withTemplateFormat(TemplateFormats.markdown());
    }
}

```

If you want to make use of Spring REST Docs support for a parameterized output directory, you can create a `RestDocumentationResultHandler` bean. The auto-configuration calls `alwaysDo` with this result handler, thereby causing

each `MockMvc` call to automatically generate the default snippets. The following example shows a `RestDocumentationResultHandler` being defined:

```
@TestConfiguration
static class ResultHandlerConfiguration {

    @Bean
    public RestDocumentationResultHandler restDocumentation() {
        return MockMvcRestDocumentation.document("{method-name}");
    }

}
```

Auto-configured Spring REST Docs Tests with WebTestClient

`@AutoConfigureRestDocs` can also be used with `WebTestClient`. You can inject it by using `@Autowired` and use it in your tests as you normally would when using `@WebFluxTest` and Spring REST Docs, as shown in the following example:

```
import org.junit.jupiter.api.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.restdocs.AutoConfigureRestDocs;
;
import org.springframework.boot.test.autoconfigure.web.reactive.WebFluxTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.reactive.server.WebTestClient;

import static
org.springframework.restdocs.webtestclient.WebTestClientRestDocumentation.
document;

@RunWith(SpringRunner.class)
@WebFluxTest
@AutoConfigureRestDocs
public class UsersDocumentationTests {

    @Autowired
    private WebTestClient webTestClient;

    @Test
    void listUsers() {
```

```

        this.webTestClient.get().uri("/").exchange().expectStatus().isOk()
.expectBody()
        .consumeWith(document("list-users")));
    }

}

```

If you require more control over Spring REST Docs configuration than offered by the attributes of `@AutoConfigureRestDocs`, you can use a `RestDocsWebTestClientConfigurationCustomizer` bean, as shown in the following example:

```

@TestConfiguration
public static class CustomizationConfiguration
    implements RestDocsWebTestClientConfigurationCustomizer {

    @Override
    public void customize(WebTestClientRestDocumentationConfigurer
configurer) {
        configurer.snippets().withEncoding("UTF-8");
    }
}

```

Auto-configured Spring REST Docs Tests with REST Assured

`@AutoConfigureRestDocs` makes a `RequestSpecification` bean, preconfigured to use Spring REST Docs, available to your tests. You can inject it by using `@Autowired` and use it in your tests as you normally would when using REST Assured and Spring REST Docs, as shown in the following example:

```

import io.restassured.specification.RequestSpecification;
import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.restdocs.AutoConfigureRestDocs
;
import org.springframework.boot.test.context.SpringBootTest;
import
org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.boot.web.server.LocalServerPort;
import org.springframework.test.context.junit4.SpringRunner;

```

```

import static io.restassured.RestAssured.given;
import static org.hamcrest.CoreMatchers.is;
import static
org.springframework.restdocs.restassured3.RestAssuredRestDocumentation.doc
ument;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@AutoConfigureRestDocs
public class UserDocumentationTests {

    @LocalServerPort
    private int port;

    @Autowired
    private RequestSpecification documentationSpec;

    @Test
    public void listUsers() {
        given(this.documentationSpec).filter(document("list-
users")).when()

            .port(this.port).get("/").then().assertThat().statusCode(is(200));
    }

}

```

If you require more control over Spring REST Docs configuration than offered by the attributes of `@AutoConfigureRestDocs`, a `RestDocsRestAssuredConfigurationCustomizer` bean can be used, as shown in the following example:

```

@Configuration
public static class CustomizationConfiguration
    implements RestDocsRestAssuredConfigurationCustomizer {

    @Override
    public void customize(RestAssuredRestDocumentationConfigurer
configurer) {

        configurer.snippets().withTemplateFormat(TemplateFormats.markdown(
));
    }
}

```

```
}
```

46.3.22 Additional Auto-configuration and Slicing

Each slice provides one or more `@AutoConfigure...` annotations that namely defines the auto-configurations that should be included as part of a slice. Additional auto-configurations can be added by creating a custom `@AutoConfigure...` annotation or simply by adding `@ImportAutoConfiguration` to the test as shown in the following example:

```
@RunWith(SpringRunner.class)
@JdbcTest
@ImportAutoConfiguration(IntegrationAutoConfiguration.class)
public class ExampleJdbcTests {

}
```



Make sure to not use the regular `@Import` annotation to import auto-configurations as they are handled in a specific way by Spring Boot.

46.3.23 User Configuration and Slicing

If you [structure your code](#) in a sensible way, your `@SpringBootApplication` class is [used by default](#) as the configuration of your tests.

It then becomes important not to litter the application's main class with configuration settings that are specific to a particular area of its functionality.

Assume that you are using Spring Batch and you rely on the auto-configuration for it. You could define your `@SpringBootApplication` as follows:

```
@SpringBootApplication
@EnableBatchProcessing
public class SampleApplication { ... }
```

Because this class is the source configuration for the test, any slice test actually tries to start Spring Batch, which is definitely not what you want to do. A recommended approach is to move that area-specific configuration to a separate `@Configuration` class at the same level as your application, as shown in the following example:

```
@Configuration  
@EnableBatchProcessing  
public class BatchConfiguration { ... }
```



Depending on the complexity of your application, you may either have a single `@Configuration` class for your customizations or one class per domain area. The latter approach lets you enable it in one of your tests, if necessary, with the `@Import` annotation.

Test slices exclude `@Configuration` classes from scanning. For example, for a `@WebMvcTest`, the following configuration will not include the given `WebMvcConfigurer` bean in the application context loaded by the test slice:

```
@Configuration  
public class WebConfiguration {  
    @Bean  
    public WebMvcConfigurer testConfigurer() {  
        return new WebMvcConfigurer() {  
            ...  
        };  
    }  
}
```

The configuration below will, however, cause the custom `WebMvcConfigurer` to be loaded by the test slice.

```
@Component  
public class TestWebMvcConfigurer extends WebMvcConfigurer {  
    ...  
}
```

Another source of confusion is classpath scanning. Assume that, while you structured your code in a sensible way, you need to scan an additional package. Your application may resemble the following code:

```
@SpringBootApplication  
@ComponentScan({ "com.example.app", "org.acme.another" })  
public class SampleApplication { ... }
```

Doing so effectively overrides the default component scan directive with the side effect of scanning those two packages regardless of the slice that you chose. For instance, a `@DataJpaTest` seems to suddenly scan components and user configurations of your application. Again, moving the custom directive to a separate class is a good way to fix this issue.



If this is not an option for you, you can create a `@SpringBootTestConfiguration` somewhere in the hierarchy of your test so that it is used instead. Alternatively, you can specify a source for your test, which disables the behavior of finding a default one.

46.3.24 Using Spock to Test Spring Boot Applications

If you wish to use Spock to test a Spring Boot application, you should add a dependency on Spock's `spock-spring` module to your application's build. `spock-spring` integrates Spring's test framework into Spock. It is recommended that you use Spock 1.2 or later to benefit from a number of improvements to Spock's Spring Framework and Spring Boot integration. See [the documentation for Spock's Spring module](#) for further details.

46.4 Test Utilities

A few test utility classes that are generally useful when testing your application are packaged as part of `spring-boot`.

46.4.1 ConfigFileApplicationContextInitializer

`ConfigFileApplicationContextInitializer` is an `ApplicationContextInitializer` that you can apply to your tests to load Spring Boot `application.properties` files. You can use it when you do not need the full set of features provided by `@SpringBootTest`, as shown in the following example:

```
@ContextConfiguration(classes = Config.class,  
    initializers = ConfigFileApplicationContextInitializer.class)
```



Using `ConfigFileApplicationContextInitializer` alone does not provide support for `@Value("${...}")` injection. Its only job is to ensure

that `application.properties` files are loaded into Spring's Environment. For `@Value` support, you need to either additionally configure a `PropertySourcesPlaceholderConfigurer` or use `@SpringBootTest`, which auto-configures one for you.

46.4.2 TestPropertyValues

`TestPropertyValues` lets you quickly add properties to a `ConfigurableEnvironment` or `ConfigurableApplicationContext`. You can call it with `key=value` strings, as follows:

```
TestPropertyValues.of("org=Spring", "name=Boot").applyTo(env);
```

46.4.3 OutputCapture

`OutputCapture` is a JUnit `Rule` that you can use to capture `System.out` and `System.err` output. You can declare the capture as a `@Rule` and then use `toString()` for assertions, as follows:

```
import org.junit.Rule;
import org.junit.Test;
import org.springframework.boot.test.rule.OutputCapture;

import static org.hamcrest.Matchers.*;
import static org.junit.Assert.*;

public class MyTest {

    @Rule
    public OutputCapture capture = new OutputCapture();

    @Test
    public void testName() throws Exception {
        System.out.println("Hello World!");
        assertThat(capture.toString(), containsString("World"));
    }
}
```

46.4.4 TestRestTemplate



Spring Framework 5.0 provides a new `WebTestClient` that works for [WebFlux integration tests](#) and both [WebFlux and MVC end-to-end testing](#). It provides a fluent API

for assertions, unlike `TestRestTemplate`.

`TestRestTemplate` is a convenience alternative to Spring's `RestTemplate` that is useful in integration tests. You can get a vanilla template or one that sends Basic HTTP authentication (with a username and password). In either case, the template behaves in a test-friendly way by not throwing exceptions on server-side errors. It is recommended, but not mandatory, to use the Apache HTTP Client (version 4.3.2 or better). If you have that on your classpath, the `TestRestTemplate` responds by configuring the client appropriately. If you do use Apache's HTTP client, some additional test-friendly features are enabled:

- Redirects are not followed (so you can assert the response location).
- Cookies are ignored (so the template is stateless).

`TestRestTemplate` can be instantiated directly in your integration tests, as shown in the following example:

```
public class MyTest {  
  
    private TestRestTemplate template = new TestRestTemplate();  
  
    @Test  
    public void testRequest() throws Exception {  
        HttpHeaders headers = this.template.getForEntity(  
            "https://myhost.example.com/example",  
            String.class).getHeaders();  
  
        assertThat(headers.getLocation()).hasHost("other.example.com");  
    }  
}
```

Alternatively, if you use the `@SpringBootTest` annotation with `WebEnvironment.RANDOM_PORT` or `WebEnvironment.DEFINED_PORT`, you can inject a fully configured `TestRestTemplate` and start using it. If necessary, additional customizations can be applied through the `RestTemplateBuilder` bean. Any URLs that do not specify a host and port automatically connect to the embedded server, as shown in the following example:

```
@RunWith(SpringRunner.class)
```

```

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class SampleWebClientTests {

    @Autowired
    private TestRestTemplate template;

    @Test
    public void testRequest() {
        HttpHeaders headers =
this.template.getForEntity("/example", String.class)
            .getHeaders();

        assertThat(headers.getLocation()).hasHost("other.example.com");
    }

    @TestConfiguration
    static class Config {

        @Bean
        public RestTemplateBuilder restTemplateBuilder() {
            return new
RestTemplateBuilder().setConnectTimeout(Duration.ofSeconds(1))

            .setReadTimeout(Duration.ofSeconds(1));
        }

    }
}

```

47. WebSockets

Spring Boot provides WebSockets auto-configuration for embedded Tomcat, Jetty, and Undertow. If you deploy a war file to a standalone container, Spring Boot assumes that the container is responsible for the configuration of its WebSocket support.

Spring Framework provides [rich WebSocket support](#) for MVC web applications that can be easily accessed through the `spring-boot-starter-websocket` module.

WebSocket support is also available for [reactive web applications](#) and requires to include the WebSocket API alongside `spring-boot-starter-webflux`:

```
<dependency>
    <groupId>javax.websocket</groupId>
    <artifactId>javax.websocket-api</artifactId>
</dependency>
```

48. Web Services

Spring Boot provides Web Services auto-configuration so that all you must do is define your `Endpoints`.

The [Spring Web Services features](#) can be easily accessed with the `spring-boot-starter-webservices` module.

`SimpleWsdl11Definition` and `SimpleXsdSchema` beans can be automatically created for your WSDLs and XSDs respectively. To do so, configure their location, as shown in the following example:

```
spring.webservices.wsdl-locations=classpath:/wsdl
```

48.1 Calling Web Services with `WebServiceTemplate`

If you need to call remote Web services from your application, you can use the `WebServiceTemplate` class. Since `WebServiceTemplate` instances often need to be customized before being used, Spring Boot does not provide any single auto-configured `WebServiceTemplate` bean. It does, however, auto-configure a `WebServiceTemplateBuilder`, which can be used to create `WebServiceTemplate` instances when needed.

The following code shows a typical example:

```
@Service
public class MyService {

    private final WebServiceTemplate webServiceTemplate;
```

```

public MyService(WebServiceTemplateBuilder
webServiceTemplateBuilder) {
    this.webServiceTemplate =
webServiceTemplateBuilder.build();
}

public DetailsResp someWsCall(DetailsReq detailsReq) {
    return (DetailsResp)
this.webServiceTemplate.marshalSendAndReceive(detailsReq, new
SoapActionCallback(ACTION));

}

}

```

By default, `WebServiceTemplateBuilder` detects a suitable HTTP-based `WebServiceMessageSender` using the available HTTP client libraries on the classpath. You can also customize read and connection timeouts as follows:

```

@Bean
public WebServiceTemplate webServiceTemplate(WebServiceTemplateBuilder
builder) {
    return builder.messageSenders(new
HttpWebServiceMessageSenderBuilder()

.setConnectTimeout(5000).setReadTimeout(2000).build()).build();
}

```

49. Creating Your Own Auto-configuration

If you work in a company that develops shared libraries, or if you work on an open-source or commercial library, you might want to develop your own auto-configuration. Auto-configuration classes can be bundled in external jars and still be picked-up by Spring Boot.

Auto-configuration can be associated to a “starter” that provides the auto-configuration code as well as the typical libraries that you would use with it.

We first cover what you need to know to build your own auto-configuration and then we move on to the [typical steps required to create a custom starter](#).



A [demo project](#) is available to showcase how you can create a starter step-by-step.

49.1 Understanding Auto-configured Beans

Under the hood, auto-configuration is implemented with standard `@Configuration` classes. Additional `@Conditional` annotations are used to constrain when the auto-configuration should apply. Usually, auto-configuration classes use `@ConditionalOnClass` and `@ConditionalOnMissingBean` annotations. This ensures that auto-configuration applies only when relevant classes are found and when you have not declared your own `@Configuration`.

You can browse the source code of [spring-boot-autoconfigure](#) to see the `@Configuration` classes that Spring provides (see the `META-INF/spring.factories` file).

49.2 Locating Auto-configuration Candidates

Spring Boot checks for the presence of a `META-INF/spring.factories` file within your published jar. The file should list your configuration classes under the `EnableAutoConfiguration` key, as shown in the following example:

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.mycorp.libx.autoconfigure.LibXAutoConfiguration,\
com.mycorp.libx.autoconfigure.LibXWebAutoConfiguration
```



Auto-configurations must be loaded that way *only*. Make sure that they are defined in a specific package space and that they are never the target of component scanning. Furthermore, auto-configuration classes should not enable component scanning to find additional components. Specific `@Imports` should be used instead.

You can use the `@AutoConfigureAfter` or `@AutoConfigureBefore` annotations if your configuration needs to be applied in a specific order. For example, if you provide web-specific configuration, your class may need to be applied after `WebMvcAutoConfiguration`.

If you want to order certain auto-configurations that should not have any direct knowledge of each other, you can also use `@AutoConfigureOrder`. That annotation has the same semantic as the regular `@Order` annotation but provides a dedicated order for auto-configuration classes.

49.3 Condition Annotations

You almost always want to include one or more `@Conditional` annotations on your auto-configuration class. The `@ConditionalOnMissingBean` annotation is one common example that is used to allow developers to override auto-configuration if they are not happy with your defaults.

Spring Boot includes a number of `@Conditional` annotations that you can reuse in your own code by annotating `@Configuration` classes or individual `@Bean` methods. These annotations include:

- [Section 49.3.1, “Class Conditions”](#)
- [Section 49.3.2, “Bean Conditions”](#)
- [Section 49.3.3, “Property Conditions”](#)
- [Section 49.3.4, “Resource Conditions”](#)
- [Section 49.3.5, “Web Application Conditions”](#)
- [Section 49.3.6, “SpEL Expression Conditions”](#)

49.3.1 Class Conditions

The `@ConditionalOnClass` and `@ConditionalOnMissingClass` annotations let `@Configuration` classes be included based on the presence or absence of specific classes. Due to the fact that annotation metadata is parsed by using `ASM`, you can use the `value` attribute to refer to the real class, even though that class might not actually appear on the running application classpath. You can also use the `name` attribute if you prefer to specify the class name by using a `String` value.

This mechanism does not apply the same way to `@Bean` methods where typically the return type is the target of the condition: before the condition on the method applies, the JVM will have loaded the class and potentially processed method references which will fail if the class is not present.

To handle this scenario, a separate `@Configuration` class can be used to isolate the condition, as shown in the following example:

```

@Configuration
// Some conditions
public class MyAutoConfiguration {

    // Auto-configured beans

    @Configuration
    @ConditionalOnClass(EmbeddedAcmeService.class)
    static class EmbeddedConfiguration {

        @Bean
        @ConditionalOnMissingBean
        public EmbeddedAcmeService embeddedAcmeService() { ... }

    }

}

```



If you use `@ConditionalOnClass` or `@ConditionalOnMissingClass` as a part of a meta-annotation to compose your own composed annotations, you must use `name` as referring to the class in such a case is not handled.

49.3.2 Bean Conditions

The `@ConditionalOnBean` and `@ConditionalOnMissingBean` annotations let a bean be included based on the presence or absence of specific beans. You can use the `value` attribute to specify beans by type or `name` to specify beans by name. The `search` attribute lets you limit the `ApplicationContext` hierarchy that should be considered when searching for beans.

When placed on a `@Bean` method, the target type defaults to the return type of the method, as shown in the following example:

```

@Configuration
public class MyAutoConfiguration {

    @Bean
    @ConditionalOnMissingBean
    public MyService myService() { ... }

}

```

In the preceding example, the `myService` bean is going to be created if no bean of type `MyService` is already contained in the `ApplicationContext`.



You need to be very careful about the order in which bean definitions are added, as these conditions are evaluated based on what has been processed so far. For this reason, we recommend using only `@ConditionalOnBean` and `@ConditionalOnMissingBean` annotations on auto-configuration classes (since these are guaranteed to load after any user-defined bean definitions have been added).



`@ConditionalOnBean` and `@ConditionalOnMissingBean` do not prevent `@Configuration` classes from being created. The only difference between using these conditions at the class level and marking each contained `@Bean` method with the annotation is that the former prevents registration of the `@Configuration` class as a bean if the condition does not match.

49.3.3 Property Conditions

The `@ConditionalOnProperty` annotation lets configuration be included based on a Spring Environment property. Use the `prefix` and `name` attributes to specify the property that should be checked. By default, any property that exists and is not equal to `false` is matched. You can also create more advanced checks by using the `havingValue` and `matchIfMissing` attributes.

49.3.4 Resource Conditions

The `@ConditionalOnResource` annotation lets configuration be included only when a specific resource is present. Resources can be specified by using the usual Spring conventions, as shown in the following example: `file:/home/user/test.dat`.

49.3.5 Web Application Conditions

The `@ConditionalOnWebApplication` and `@ConditionalOnNotWebApplication` annotations let configuration be included depending on whether the application

is a “web application”. A web application is any application that uses a Spring `WebApplicationContext`, defines a `session` scope, or has a `StandardServletEnvironment`.

49.3.6 SpEL Expression Conditions

The `@ConditionalOnExpression` annotation lets configuration be included based on the result of a [SpEL expression](#).

49.4 Testing your Auto-configuration

An auto-configuration can be affected by many factors: user configuration (`@Bean` definition and `Environment` customization), condition evaluation (presence of a particular library), and others. Concretely, each test should create a well defined `ApplicationContext` that represents a combination of those customizations. `ApplicationContextRunner` provides a great way to achieve that.

`ApplicationContextRunner` is usually defined as a field of the test class to gather the base, common configuration. The following example makes sure that `UserServiceAutoConfiguration` is always invoked:

```
private final ApplicationContextRunner contextRunner = new
ApplicationContextRunner()
    .withConfiguration(AutoConfigurations.of(UserServiceAutoConfiguration.class));
```



If multiple auto-configurations have to be defined, there is no need to order their declarations as they are invoked in the exact same order as when running the application.

Each test can use the runner to represent a particular use case. For instance, the sample below invokes a user configuration (`UserConfiguration`) and checks that the auto-configuration backs off properly. Invoking `run` provides a callback context that can be used with `Assert4J`.

```
@Test
public void defaultServiceBacksOff() {
    this.contextRunner.withUserConfiguration(UserConfiguration.class)
        .run((context) -> {
```

```

        assertThat(context).hasSingleBean(UserService.class);

        assertThat(context.getBean(UserService.class)).isSameAs(
            context.getBean(UserConfiguration.class).myUserService());
    });

@Configuration
static class UserConfiguration {

    @Bean
    public UserService myUserService() {
        return new UserService("mine");
    }

}

```

It is also possible to easily customize the [Environment](#), as shown in the following example:

```

@Test
public void serviceNameCanBeConfigured() {
    this.contextRunner.withPropertyValues("user.name=test123").run((context) -> {
        assertThat(context).hasSingleBean(UserService.class);

        assertThat(context.getBean(UserService.class).getName()).isEqualTo("test123");
    });
}

```

The runner can also be used to display the [ConditionEvaluationReport](#). The report can be printed at [INFO](#) or [DEBUG](#) level. The following example shows how to use the [ConditionEvaluationReportLoggingListener](#) to print the report in auto-configuration tests.

```

@Test
public void autoConfigTest {
    ConditionEvaluationReportLoggingListener initializer = new
    ConditionEvaluationReportLoggingListener(
        LogLevel.INFO);
    ApplicationContextRunner contextRunner = new
    ApplicationContextRunner()
}

```

```
        .withInitializer(initializer).run((context) -> {
            // Do something...
        });
    }
```

49.4.1 Simulating a Web Context

If you need to test an auto-configuration that only operates in a Servlet or Reactive web application context, use the `WebApplicationContextRunner` or `ReactiveWebApplicationContextRunner` respectively.

49.4.2 Overriding the Classpath

It is also possible to test what happens when a particular class and/or package is not present at runtime. Spring Boot ships with a `FilteredClassLoader` that can easily be used by the runner. In the following example, we assert that if `UserService` is not present, the auto-configuration is properly disabled:

```
@Test
public void serviceIsIgnoredIfLibraryIsNotPresent() {
    this.contextRunner.withClassLoader(new
    FilteredClassLoader(UserService.class))
        .run((context) ->
    assertThat(context).doesNotHaveBean("userService"));
}
```

49.5 Creating Your Own Starter

A full Spring Boot starter for a library may contain the following components:

- The `autoconfigure` module that contains the auto-configuration code.
- The `starter` module that provides a dependency to the `autoconfigure` module as well as the library and any additional dependencies that are typically useful. In a nutshell, adding the starter should provide everything needed to start using that library.



You may combine the auto-configuration code and the dependency management in a single module if you do not need to separate those two concerns.

49.5.1 Naming

You should make sure to provide a proper namespace for your starter. Do not start your module names with `spring-boot`, even if you use a different Maven `groupId`. We may offer official support for the thing you auto-configure in the future.

As a rule of thumb, you should name a combined module after the starter. For example, assume that you are creating a starter for "acme" and that you name the auto-configure module `acme-spring-boot-autoconfigure` and the starter `acme-spring-boot-starter`. If you only have one module that combines the two, name it `acme-spring-boot-starter`.

Also, if your starter provides configuration keys, use a unique namespace for them. In particular, do not include your keys in the namespaces that Spring Boot uses (such as `server`, `management`, `spring`, and so on). If you use the same namespace, we may modify these namespaces in the future in ways that break your modules.

Make sure to [trigger meta-data generation](#) so that IDE assistance is available for your keys as well. You may want to review the generated meta-data (`META-INF/spring-configuration-metadata.json`) to make sure your keys are properly documented.

49.5.2 `autoconfigure` Module

The `autoconfigure` module contains everything that is necessary to get started with the library. It may also contain configuration key definitions (such as `@ConfigurationProperties`) and any callback interface that can be used to further customize how the components are initialized.



You should mark the dependencies to the library as optional so that you can include the `autoconfigure` module in your projects more easily. If you do it that way, the library is not provided and, by default, Spring Boot backs off.

Spring Boot uses an annotation processor to collect the conditions on auto-configurations in a metadata file (`META-INF/spring-autoconfigure-metadata.properties`). If that file is present, it is used to eagerly filter auto-configurations that do not match, which will improve startup time. It is

recommended to add the following dependency in a module that contains auto-configurations:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-autoconfigure-processor</artifactId>
    <optional>true</optional>
</dependency>
```

With Gradle 4.5 and earlier, the dependency should be declared in the `compileOnly` configuration, as shown in the following example:

```
dependencies {
    compileOnly "org.springframework.boot:spring-boot-autoconfigure-processor"
}
```

With Gradle 4.6 and later, the dependency should be declared in the `annotationProcessor` configuration, as shown in the following example:

```
dependencies {
    annotationProcessor "org.springframework.boot:spring-boot-autoconfigure-processor"
}
```

49.5.3 Starter Module

The starter is really an empty jar. Its only purpose is to provide the necessary dependencies to work with the library. You can think of it as an opinionated view of what is required to get started.

Do not make assumptions about the project in which your starter is added. If the library you are auto-configuring typically requires other starters, mention them as well. Providing a proper set of *default* dependencies may be hard if the number of optional dependencies is high, as you should avoid including dependencies that are unnecessary for a typical usage of the library. In other words, you should not include optional dependencies.

Either way, your starter must reference the core Spring Boot starter (`spring-boot-starter`) directly or indirectly (i.e. no need to add it if your starter relies on another starter). If a project is created with only your custom starter, Spring Boot's core features will

be honoured by the presence of the core starter.

50. Kotlin support

[Kotlin](#) is a statically-typed language targeting the JVM (and other platforms) which allows writing concise and elegant code while providing [interoperability](#) with existing libraries written in Java.

Spring Boot provides Kotlin support by leveraging the support in other Spring projects such as Spring Framework, Spring Data, and Reactor. See the [Spring Framework Kotlin support documentation](#) for more information.

The easiest way to start with Spring Boot and Kotlin is to follow [this comprehensive tutorial](#). You can create new Kotlin projects via [start.spring.io](#). Feel free to join the #spring channel of [Kotlin Slack](#) or ask a question with the `spring` and `kotlin` tags on [Stack Overflow](#) if you need support.

50.1 Requirements

Spring Boot supports Kotlin 1.2.x. To use Kotlin, `org.jetbrains.kotlin:kotlin-stdlib` and `org.jetbrains.kotlin:kotlin-reflect` must be present on the classpath. The `kotlin-stdlib` variants `kotlin-stdlib-jdk7` and `kotlin-stdlib-jdk8` can also be used.

Since [Kotlin classes are final by default](#), you are likely to want to configure [kotlin-spring](#) plugin in order to automatically open Spring-annotated classes so that they can be proxied.

[Jackson's Kotlin module](#) is required for serializing / deserializing JSON data in Kotlin. It is automatically registered when found on the classpath. A warning message is logged if Jackson and Kotlin are present but the Jackson Kotlin module is not.



These dependencies and plugins are provided by default if one bootstraps a Kotlin project on start.spring.io.

50.2 Null-safety

One of Kotlin's key features is [null-safety](#). It deals with `null` values at compile time rather than deferring the problem to runtime and encountering a `NullPointerException`. This helps to eliminate a common source of bugs without paying the cost of wrappers like `Optional`. Kotlin also allows using functional constructs with nullable values as described in this [comprehensive guide to null-safety in Kotlin](#).

Although Java does not allow one to express null-safety in its type system, Spring Framework, Spring Data, and Reactor now provide null-safety of their API via tooling-friendly annotations. By default, types from Java APIs used in Kotlin are recognized as [platform types](#) for which null-checks are relaxed. [Kotlin's support for JSR 305 annotations](#) combined with nullability annotations provide null-safety for the related Spring API in Kotlin.

The JSR 305 checks can be configured by adding the `-Xjsr305` compiler flag with the following options: `-Xjsr305={strict|warn|ignore}`. The default behavior is the same as `-Xjsr305=warn`. The `strict` value is required to have null-safety taken in account in Kotlin types inferred from Spring API but should be used with the knowledge that Spring API nullability declaration could evolve even between minor releases and more checks may be added in the future).



Generic type arguments, varargs and array elements nullability are not yet supported. See [SPR-15942](#) for up-to-date information. Also be aware that Spring Boot's own API is [not yet annotated](#).

50.3 Kotlin API

50.3.1 runApplication

Spring Boot provides an idiomatic way to run an application with `runApplication<MyApplication>(*args)` as shown in the following example:

```
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

@SpringBootApplication
class MyApplication

fun main(args: Array<String>) {
    runApplication<MyApplication>(*args)
}
```

This is a drop-in replacement

for `SpringApplication.run(MyApplication::class.java, *args)`. It also allows customization of the application as shown in the following example:

```
runApplication<MyApplication>(*args) {
    setBannerMode(OFF)
}
```

50.3.2 Extensions

Kotlin [extensions](#) provide the ability to extend existing classes with additional functionality. The Spring Boot Kotlin API makes use of these extensions to add new Kotlin specific conveniences to existing APIs.

`TestRestTemplate` extensions, similar to those provided by Spring Framework for `RestOperations` in Spring Framework, are provided. Among other things, the extensions make it possible to take advantage of Kotlin reified type parameters.

50.4 Dependency management

In order to avoid mixing different version of Kotlin dependencies on the classpath, dependency management of the following Kotlin dependencies is provided:

- `kotlin-reflect`
- `kotlin-runtime`
- `kotlin-stdlib`
- `kotlin-stdlib-jdk7`
- `kotlin-stdlib-jdk8`
- `kotlin-stdlib-jre7`
- `kotlin-stdlib-jre8`

With Maven, the Kotlin version can be customized via the `kotlin.version` property and plugin management is provided for `kotlin-maven-plugin`. With Gradle, the Spring Boot plugin automatically aligns the `kotlin.version` with the version of the Kotlin plugin.

50.5 @ConfigurationProperties

`@ConfigurationProperties` currently only works with `lateinit` or nullable `var` properties (the former is recommended), since immutable classes initialized by constructors are [not yet supported](#).

```
@ConfigurationProperties("example.kotlin")
class KotlinExampleProperties {

    lateinit var name: String

    lateinit var description: String

    val myService = MyService()

    class MyService {

        lateinit var apiToken: String

        lateinit var uri: URI

    }
}
```



To generate [your own metadata](#) using the annotation processor, [kapt should be configured](#) with the `spring-boot-configuration-processor` dependency.

50.6 Testing

While it is possible to use JUnit 4 (the default provided by `spring-boot-starter-test`) to test Kotlin code, JUnit 5 is recommended. JUnit 5 enables a test class to be instantiated once and reused for all of the

class's tests. This makes it possible to use `@BeforeAll` and `@AfterAll` annotations on non-static methods, which is a good fit for Kotlin.

To use JUnit 5, exclude `junit:junit` dependency from `spring-boot-starter-test`, add JUnit 5 dependencies, and configure the Maven or Gradle plugin accordingly. See the [JUnit 5 documentation](#) for more details. You also need to [switch test instance lifecycle to "per-class"](#).

To mock Kotlin classes, [MockK](#) is recommended. If you need the [Mockkk](#) equivalent of the Mockito specific `@MockBean` and `@SpyBean` annotations, you can use [SpringMockK](#) which provides similar `@MockkBean` and `@SpykBean` annotations.

50.7 Resources

50.7.1 Further reading

- [Kotlin language reference](#)
- [Kotlin Slack](#) (with a dedicated #spring channel)
- [Stackoverflow with spring and kotlin tags](#)
- [Try Kotlin in your browser](#)
- [Kotlin blog](#)
- [Awesome Kotlin](#)
- [Tutorial: building web applications with Spring Boot and Kotlin](#)
- [Developing Spring Boot applications with Kotlin](#)
- [A Geospatial Messenger with Kotlin, Spring Boot and PostgreSQL](#)
- [Introducing Kotlin support in Spring Framework 5.0](#)
- [Spring Framework 5 Kotlin APIs, the functional way](#)

50.7.2 Examples

- [spring-boot-kotlin-demo](#): regular Spring Boot + Spring Data JPA project
- [mixit](#): Spring Boot 2 + WebFlux + Reactive Spring Data MongoDB
- [spring-kotlin-fullstack](#): WebFlux Kotlin fullstack example with Kotlin2js for frontend instead of JavaScript or TypeScript
- [spring-petclinic-kotlin](#): Kotlin version of the Spring PetClinic Sample Application
- [spring-kotlin-deepdive](#): a step by step migration for Boot 1.0 + Java

to Boot 2.0 + Kotlin