**WatchService API**

Java 7 added a new feature to its NIO package called Watch Service API which allows applications to monitor a directory for changes

The [java.nio.file](https://docs.oracle.com/javase/7/docs/api/java/nio/file/package-summary.html) package provides a file change notification API, called the Watch Service API. This API enables you to register a directory (or directories) with the watch service. When registering, you tell the service which types of events you are interested in: file creation, file deletion, or file modification

* Implemented WatchService API to watch for file changes and handling in running mode without restarting server.
* Implemented Executor framework this mechanism separates the task creation and execution.

|  |
| --- |
| @Component  @Scope("prototype")  **publicclass**FileWatcherThread**implements** Runnable {  //private Long sleepTime = 1000L \* 60;  **private** Long sleepTime = 10000l;  **private** String directoryToWatch = "C:\\Users\\Mekala.Nagendra\\Desktop\\properties";  **private**WatchKeywatchKey;  @Autowired  **private**ProcessDataprocessData;  @PostConstruct  **publicvoid**setUpDirectory() {  processData.setFilterDataTemps("N");  }  **publicvoid**run() {  **try** {  // get the directory we want to watch  Path directory = Paths.*get*(directoryToWatch);  // creates a WatchService and registers the given directory  FileSystemfileSystem = directory.getFileSystem();  InputStreaminputStream =**null**;  // make a new watch service that we can register interest in  // directories and files with.  WatchServicewatchService = fileSystem.newWatchService();  watchKey = directory.register(watchService, ***ENTRY\_MODIFY***,***ENTRY\_CREATE***,***ENTRY\_DELETE***);  **while**(**true**) {  System.***out***.println("Waiting for key to be signalled...");  watchKey = watchService.take();  List<WatchEvent<?>>watchedEvents = watchKey.pollEvents();  // we only registered for one kind of event and this is it  **for** (WatchEvent<?>watchEvent :watchedEvents) {  String fileName = watchEvent.context().toString();  Kind<?>kind = watchEvent.kind();  **if** (kind.toString().equals("ENTRY\_DELETE") || kind.toString().equals("ENTRY\_CREATE")) {  System.***out***.printf("Received %s event for file: %s\n", watchEvent.kind(), watchEvent.context());  **break**;  } **else** {  // modification events for the file  System.***out***.printf("Received %s event for file: %s\n", watchEvent.kind(), watchEvent.context());  inputStream = **new**FileInputStream(directoryToWatch + "/" + fileName);  Properties prop = **new**Properties();  prop.load(inputStream);  processData.setFilterDataTemps(prop.getProperty("filterDataTemps"));  System.***out***.println("Value getting: "+prop.getProperty("filterDataTemps"));  **break**;  }  }  **boolean**valid = watchKey.reset();  **if**(!valid){  System.***out***.println(" watcher reset failed");  **break**;  }  Thread.*sleep*(sleepTime);  }  } **catch** (Exception e) {  System.***out***.println("Wather on file failed." + directoryToWatch + " " + e);  }  }    /\*\*  \* stopMonitoring  \*/  **publicvoid**stopMonitoring() {  **if** (watchKey != **null**) {  watchKey.cancel();  }  }  }  @Configuration  @EnableAsync  **publicclass**AppConfig {  @Bean(name="taskExecutor")  **public**ThreadPoolTaskExecutortaskExecutor() {  ThreadPoolTaskExecutorexecutor = **new**ThreadPoolTaskExecutor();  executor.setCorePoolSize(5);  executor.setMaxPoolSize(10);  executor.setQueueCapacity(25);  executor.setWaitForTasksToCompleteOnShutdown(**true**);  **return**executor;  }  } |

**JMS (Java Messaging System)**

### **What is Message?**

Message is a piece of information. It can be a text, XML document, JSON data or an Entity (Java Object) etc. Message is very useful data to communicate different systems.

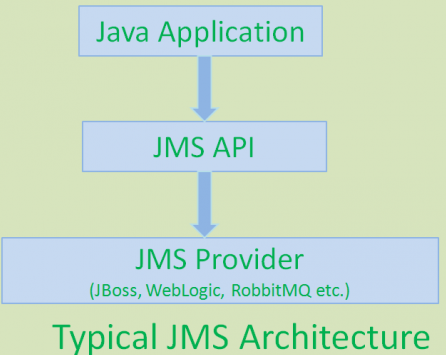
### **What is Messaging?**

Messaging means exchanging information between different components in the same system or different systems. It can happen in either synchronous manner or asynchronous manner.

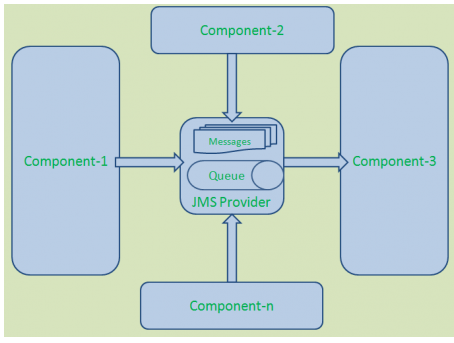
### **What is JMS?**

JMS stands for **Java Message Service**. JMS API is a Java API which contains common set of interfaces to implement enterprisebased messaging systems. JMS API is used to implement Messaging systems in Java-based applications only, does not support other languages.

JMS API is used to create, send, receive and read messages or exchange messages between different systems. Once we develop a Java Messaging System with JMS API, then we can deploy same application in any JMS Provider software.



Some Enterprise applications may have n number systems and they exists in different locations. They use different platforms. If they want to exchange information in loosely coupled manner, then we should use JMS Messaging system.



### **Advantages of JMS**:

* **Loosely Coupled:**

We can develop loosely coupled applications very easily. That means JMS API is standard or specification that should be implemented by all JMS Providers so that we can change existing JMS Provider to new JMS Provider with little changes (that means only configurations) or without changing our JMS Application code.

* **Asynchronous:**

We can develop asynchronous messaging applications very easily. That means JMS Sender can send messages and continue on its own work. It does not wait till the completion of message consumption by JMS Receiver.

* **Robust and Reliable:**

JMS ensures that a message is delivered one and only once to the destination system. So we can develop reliable applications very easily.

* **Interoperability:**

JMS API allows Interoperability between other Java Platform languages like Scala and Groovy.

### **JMS Components**

A typical JMS system contains the following components:

* JMS Client
  + JMS Sender
  + JMS Receiver
* JMS Provider
* JMS Administered Objects
  + ConnectionFactory
  + Destination
* JMS Message

**JMS Client** is a Java program used to send (or produce or publish) or receive (or consume or subscribe) messages.

**JMS Sender** is a JMS Client, which is used to send messages to destination system, JMS sender, is also known as JMS Producer or JMS Publisher.

**JMS Receiver** is a JMS Client, which is used to receive messages from Source system. JMS Receiver is also known as JMS Consumer or JMS Subscriber.

**JMS API** is a set of common interfaces, which does not contain any implementation.

**JMS Provider** is a third-party system who is responsible to implement the JMS API to provide messaging features to the clients.

**JMS Provider** is also known as MOM(Message Oriented Middleware) software or Message Broker or JMS Server or Messaging Server. JMS Provider also provides some UI components to administrate and control this MOM software.

The JMSProvider is a MOM (Message Oriented Middleware); sitting between sender and receiver components for asynchronous communication i.e. if a sender sends a message, the same will be stored inside MOM. The MOM then forwards the message to receiver.

To achieve the communication, following components will be configured inside the MOM.

* **Connectionfactoires**: Provides the communication to client programs
* **Destination**: Stores the messages.There are two types.

**Queue**: Message will be delivered to Only one receiver

**Topic**: Message will be delivered to more than one receiver

**JMS Message** is an object that contains the data being transferred between JMS clients. We will discuss about JMS Message in detail in coming chapter

**JMS Administered Objects** are JMS Objects which are preconfigured by an administrator for the use of JMS clients. They are ConnectionFactory and Destination Objects.

**ConnectionFactory object** is used to create a connection between Java Application and JMS Provider. It is used by Application to communicate with JMS Provider.

**Destinations** are also JMS Objects used by a JMS Client to specify the destination of messages it is sending and the source of messages it receives.

There are two types of Destinations:

* Queue
* Topic

## Messaging Domains

There are two types of messaging domains in JMS.

1. Point-to-Point Messaging Domain
2. Publisher/Subscriber Messaging Domain

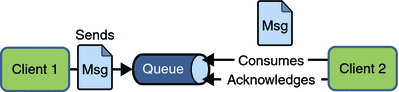
## 1) Point-to-Point (PTP) Messaging Domain

A point to point model is based on the concept of message queue, senders send message into the queue and the receiver reads messages from this queue,

In PTP model, one message is delivered to one receiver only. Here, Queue is used as a message oriented middleware (MOM).

The Queue is responsible to hold the message until receiver is ready.

In PTP model, there is no timing dependency between sender and receiver.

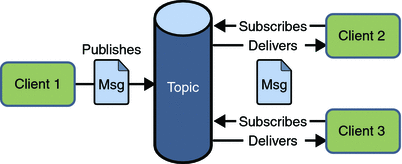


## 2) Publisher/Subscriber (Pub/Sub) Messaging Domain

## Publish subscriber model is based on the message topic concept, publisher send messages in a topic and all subscriber of the given topic receive messages

In Pub/Sub model, one message is delivered to all the subscribers. It is like broadcasting. Here, Topic is used as a message oriented middleware that is responsible to hold and deliver messages.

In PTP model, there is timing dependency between publisher and subscriber.



|  |  |  |
| --- | --- | --- |
| NO | POINT-TO-POINT MESSAGING MODEL | PUBLISH/SUBSCRIBE MESSAGING MODEL |
| 1. | Each message is delivered to one and only one JMS Receiver | Each message is delivered to multiple Consumers. |
| 2. | P2P Model has no timing dependency. | Pub/Sub model has some timing dependency. |
| 3. | JMS Receiver sends acknowledgements to JMS Sender once it receives messages. | Acknowledgement is not required. |
| 4 | A point to point model is based on the concept of message queue | Pub/Sub model uses topic |

**What is different between Queue and Topic**

In queues message can be consumed by only one client where as in the topic it can consumed by n number of clients. Both are separate domain in MOM.    
   
Queue represent PTP domain and Topic represent Pub/Sub domain

## WebLogic JMS Provider:

One of the widely used providers is Web logic. It is widely used because of following features.

* Provides messaging API.
* Supports clustering.
* Platform independent.
* Resources can be configured using Web Logic Administration Console.
* Supports XML message.
* Supports multicasting delivery.

## Spring JMS API:

Spring supports JMS framework that simplifies the complexity of the JMS implementation. The package org.springframework.jms.core provides the API for JMS core functionality. A JMS Programmer can write JMS application using Spring IOC and Spring JMS API easily. Using Spring JMS and Dependency injection, the JMS configuration details are migrated from Java code to xml file

The Spring JMS API supports a template mechanism to hide the details of Java APIs. Programmers can use[JDBC Template](https://javabeat.net/introduction-to-spring-jdbc-framework/) and [JNDI](https://javabeat.net/dependency-management-jndi/)Template classes to use Web logic JMS resource. Spring supports the JMS Template, so developers don’t have to write the complex code for JMS implementation. Templates are helper classes used to reduce the complexity of JMS and handle the creation and release of JMS resources like connection factories, destinations, and sender/receiver objects.

## Steps to integrate Spring JMS and Weblogic:

**Step 1:**

* Configure the following JMS resource inside the web logic server.
* ConnectionFactory- jms/connectionFactory and  
  **Queue** Destination- jms/testQueue

**Step 2:**

Create dynamic web application in the Web logic workshop.

|  |
| --- |
| **Message Consumer:**  package jms;  import javax.jms.JMSException;  import javax.jms.Message;  import javax.jms.Session;  import org.springframework.jms.core.JmsTemplate;  import org.springframework.jms.core.MessageCreator;  public class InvoiceQueueSender {  private JmsTemplatejmsTemplate;  public void setJmsTemplate(JmsTemplatejmsTemplate) {  this.jmsTemplate = jmsTemplate;  }  public void sendMesage() {  MessageCreatormessageCreator=new MessageCreator() {  public Message createMessage(Session session) throws  JMSException {  return session.createTextMessage("I am  sending Invoice message");}  };  jmsTemplate.send("jms/testQueue", messageCreator);  }  }  **Message Recevier:**  import javax.jms.JMSException;  import javax.jms.Message;  import javax.jms.MessageListener;  import javax.jms.TextMessage;  public class InvoiceMDB implements MessageListener {  public void onMessage(Message message) {  try {  System.out.println(((TextMessage) message).getText());  System.out.println("Hello");  } catch (JMSException ex) {  throw new RuntimeException(ex);  }  }  }  **JNDITemplate:**  It helps us to perform JNDI operations. It has methods to do lookup and binding operation. The web logic context details are configured using this interface.  <bean id="jndiTemplate" class="org.springframework  .jndi.JndiTemplate">  <property name="environment">  <props>  <prop key="java.naming.factory.initial">  weblogic.jndi.WLInitialContextFactory</prop>  <prop key="java.naming.provider.url">t3://localhost:7001</prop>  </props>  </property>  </bean>  **JndiObjectFactoryBean:**  It is used to look up the JNDI object on startup and cache it. This interface is used to configure connection factory.  <bean id="queueConnectionFactory"  class="org.springframework.jndi.JndiObjectFactoryBean">  <property name="jndiTemplate">  <ref bean="jndiTemplate" />  </property>  <property name="jndiName">  <value>jms/connectionFactory</value>  </property>  </bean>  DestinationResolver  It is used by JmsTemplate to resolve destination names  <bean id="jmsDestinationResolver"  class="org.springframework.jms.support.destination  .JndiDestinationResolver">  <property name="jndiTemplate">  <ref bean="jndiTemplate" />  </property>  <property name="cache">  <value>true</value>  </property>  </bean>  **JMSTemplate**  It is used to send messages. This requires information about connection factory and destination resolver while configuring.  <bean id="invoiceQueueTemplate" class="org.springframework.jms  .core.JmsTemplate">  <property name="connectionFactory">  <ref bean="queueConnectionFactory" />  </property>  <property name="destinationResolver">  <ref bean="jmsDestinationResolver" />  </property>  </bean>  Queue  The destination details will be configured using JndiObjectFactoryBean  <bean id="invoiceQueue" class="org.springframework  .jndi.JndiObjectFactoryBean">  <property name="jndiTemplate">  <ref bean="jndiTemplate" />  </property>  <property name="jndiName">  <value>jms/testQueue</value>  </property>  </bean>  MDB(MessageDrivenBean)  The details about Message driven will configured as a normal bean  <bean id="invoiceListener" class="jms.InvoiceMDB" />  Listener Configuarion  This class is used to configure the normal bean as message driven bean. While configuring, the destination details will be supplied so that the framework can have the relationship between MDB and destination  <bean id="Invoicelistener" class="org.springframework.jms  .listener.DefaultMessageListenerContainer">  <property name="concurrentConsumers" value="5" />  <property name="connectionFactory" ref="queueConnectionFactory" />  <property name="destination" ref="invoiceQueue" />  <property name="messageListener" ref="invoiceListener" />  </bean>  SenderBean  <bean id="jmsInvoiceSender" class="jms.InvoiceQueueSender">  <property name="jmsTemplate">  <ref bean="invoiceQueueTemplate" />  The final contents will be  <?xml version="1.0" encoding="UTF-8"?>  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"  "http://www.springframework.org/dtd/spring-beans.dtd">  <beans>  <bean id="invoiceListener" class="jms.InvoiceMDB" />  <bean id="jndiTemplate" class="org.springframework.jndi.JndiTemplate">  <property name="environment">  <props>  <prop key="java.naming.factory.initial">weblogic.jndi  .WLInitialContextFactory</prop>  <prop key="java.naming.provider.url">t3://localhost:7001</prop>  </props>  </property>  </bean>  <bean id="queueConnectionFactory" class="org.springframework.  jndi.JndiObjectFactoryBean">  <property name="jndiTemplate">  <ref bean="jndiTemplate" />  </property>  <property name="jndiName">  <value>jms/connectionFactory</value>  </property>  </bean>  <bean id="jmsDestinationResolver" class="org.springframework  .jms.support.destination.JndiDestinationResolver">  <property name="jndiTemplate">  <ref bean="jndiTemplate" />  </property>  <property name="cache">  <value>true</value>  </property>  </bean>  <bean id="invoiceQueueTemplate" class="org.springframework  .jms.core.JmsTemplate">  <property name="connectionFactory">  <ref bean="queueConnectionFactory" />  </property>  <property name="destinationResolver">  <ref bean="jmsDestinationResolver" />  </property>  </bean>  <bean id="jmsInvoiceSender" class="jms.InvoiceQueueSender">  <property name="jmsTemplate">  <ref bean="invoiceQueueTemplate" />  </property>  </bean>  <bean id="invoiceQueue" class="org.springframework.jndi  .JndiObjectFactoryBean">  <property name="jndiTemplate">  <ref bean="jndiTemplate" />  </property>  <property name="jndiName">  <value>jms/testQueue</value>  </property>  </bean>  <bean id="Invoicelistener" class="org.springframework.jms  .listener.DefaultMessageListenerContainer">  <property name="concurrentConsumers" value="5" />  <property name="connectionFactory" ref="queueConnectionFactory" />  <property name="destination" ref="invoiceQueue" />  <property name="messageListener" ref="invoiceListener" />  </bean>  </beans>  **Step 7:**  Create Servlet to send message  package jms;  import java.io.IOException;  import javax.servlet.ServletException;  import javax.servlet.http.HttpServletRequest;  import javax.servlet.http.HttpServletResponse;  import org.springframework.web.context.WebApplicationContext;  import org.springframework.web.context.support.WebApplicationContextUtils;  public class InvoiceSenderServlet extends javax.servlet.http.HttpServlet  implements javax.servlet.Servlet {  protected void service(HttpServletRequest request,  HttpServletResponse response) throws ServletException, IOException {  WebApplicationContextctx = WebApplicationContextUtils  .getRequiredWebApplicationContext(this.getServletContext());  InvoiceQueueSender sender = (InvoiceQueueSender) ctx  .getBean("jmsInvoiceSender");  sender.sendMesage();  }  } |

#### **Spring Cache:**

**Logging mechanism in spring boot:**

We implemented SLF4J in GPR3 application as it acts as a façade for different logging frameworks (e.g. java.util.logging, logback, Log4j). It offers a generic API making the logging independent of the actual implementation. This allows for different logging frameworks to coexist. It also helps migrate from one framework to another. It is simple, yet flexible, and allows for readability and performance improvements.

**Management and Monitoring using JMX:**

We created LoggerHandler java class to handle log levels through jconsole at run time so that we do not have to re-run the application to switch between the log levels.

**Cache Implementation (Ehcache) in spring boot:**

We implemented Ehcache for caching in GPR3 application. Ehcache is Fast &LightWeight, provides scalability into terabytes, extensible, flexible and alsoEhcache has full support for high performance distributed caching.

**Management and Monitoring using JMX:**

We created EhcacheHandler java class to handle caching operations (enableCache, disableCache, clearCache, clearAllCachemdeleteCache) at run time via jconsole without restarting the application.

Caching is a mechanism to enhance the performance of a system. It is a temporary memory that lies between the application and the persistent database. Cache memory stores recently used data items in order to reduce the number of database hits as much as possible.

In any application or system there may be some data such type which is used frequently in the processing the request for any client. If such data we will be fetch from the database then it should be impact the performance of the system. Spring Framework provides support for caching means we can cache the frequently used data in the application at startup time

## Supported cache providers in Spring:

As we know that the actual implementations of the cache is by third party library and Spring provides only the abstract layer for enabling that specific cache implementation to store the cache data. So there are following cache supported by the Spring.

* Generic
* JCache (JSR-107)
* EhCache 2.x
* Hazelcast
* Infinispan
* Couchbase
* Redis
* Caffeine
* Guava
* Simple

[**Ehcache**](http://ehcache.org/)

 is a pure Java cache with the following features: fast, simple, small foot print, minimal dependencies, provides memory and disk stores for scalability into gigabytes, scalable to hundreds of caches.  
is a pluggable cache for Hibernate

The caching feature can be declaratively enabled by simply adding the @EnableCaching annotation to any of the configuration classes:

In xml: <cache:annotation-driven />

Use Caching With Annotations:

After enabling the cache we can use following list of declarative annotations.

@Cacheable

@CacheEvict

@CachePut

@Caching

@CacheConfig

**@Cacheable:** It is one of the most important and common annotation for caching the requests. If you annotate a method with @Cacheable, if multiple requests are received by the application, then this annotation will not execute the method multiple times, instead it will send the result from the cached storage.

The simplest way to enable caching behavior for a method is to demarcate it with @Cacheable and parameterize it with the name of the cache where the results would be stored:

@Cacheable(value="cities")

public List<City>findAllCity(){

return (List<City>) cityRepository.findAll();

}

The findAllCity() call will first checks the cache cities before actually invoking the method and then caching the result.

Spring framework also supports multiple caches to be passed as parameters:

@Cacheable(value={"cities","city-list"})

public List<City>findAllCity(){

return (List<City>) cityRepository.findAll();

}

**@CacheEvict:**

If we annotate all methods with @Cacheable then the size of cache may be some problem. We don’t want to populate the cache with values that we don’t need often. Caches can grow quite large, quite fast, and we could be holding on to a lot of stale or unused data. @CacheEvict annotation is used for removing a single cache or clearing the entire cache from the cache storage so that fresh values can be loaded into the cache again:

@CacheEvict(value="cities", allEntries=true)

public List<City>findAllCity(){

return (List<City>) cityRepository.findAll();

}

Here allEntries indicated whether all the data in the cache has to be removed.

**@CachePut:**

@CachePut annotation helps for updating the cache with the latest execution without stopping the method execution. The difference between @Cacheable and @CachePut is that @Cacheable will skip running the method, whereas @CachePut will actually run the method and then put its results in the cache.

@CachePut(value="cities")

public List<City>findAllCity(){

return (List<City>) cityRepository.findAll();

}

@Caching:

What if you want to use multiple annotations of the same type for caching a method? @Caching annotation used for grouping multiple annotations of the same type together when one annotation is not sufficient for the specifying the suitable condition. For example, you can put multiple @CacheEvict or @CachePut annotation inside @Caching to narrow down your conditions as you need.

@Caching(evict = {

@CacheEvict("cities"),

@CacheEvict(value="city-list", key="#city.name") })

public List<City>findAllCity(){

return (List<City>) cityRepository.findAll();

}

@CacheConfig:

You can annotate @CacheConfig at the class level to avoid repeated mentioning in each method. For example, in the class level you can provide the cache name and in the method you just annotate with @Cacheable annotation.

@CacheConfig(cacheNames={"cities"})

public class CityMasterService {

@Cacheable

public List<City>) findAllCity() {

return (List<City>)) cityRepository.findAll();

}

}

Conditional Caching:

Sometimes we don’t want to cache some result there are no need to cache. For example – reusing our example from the @CachePut annotation – this will both execute the method as well as cache the results each and every time:

@CachePut(value="cities")

public List<City>) findAllCity(State state){

return (List<City>)) cityRepository.findAll(state.getStateCode());

}

Condition Parameter:

@CachePut(value="cities", condition="#state.stateName=='UP'")

public ListList<City>) findAllCity(State state){

return (List<City>)) cityRepository.findAll(state.getStateCode());

}

@CachePut can be parameterized with a condition parameter that takes a SpEL expression to ensure that the results are cached based on evaluating that expression.

Unless Parameter:

We can also control the caching based on the output of the method rather than the input – via the unless parameter

@CachePut(value="cities", unless="#result.length()>25")

public List<City>) findAllCity(String state){

return (List<City>)) cityRepository.findAll(state.getStateCode());

}

### 1. @Cacheable

Used for Cache-population. @Cacheable annotation indicates that the result of invoking a method (or all methods in a class) can be cached. Almost anything [object,array,list,..] can be cached. A cache itself can be imagined as a key-value based store. First time a method annotated with @Cacheable gets called, it gets executed and it’s return value is stored in Cache using a key[method parameter for instance, ]. Next time, if the method gets called using same key[same parameter for instance], the result is returned directly from Cache, **without executing the method**.

@Cacheable annotation supports many optional attributes to control cache population. These attributes can use SpEL to specify the caching criteria.

value : Specifies the name of the cache being used.  
key : Used to specify the key for cache storage. You can use SpEL to specify the key.

|  |
| --- |
| @Cacheable(value="products", key="#product.name")  public Product findProduct(Product product){//product name will be used as a key  ..  return aproduct;  } |

**Important:** If you missed to provide the ‘key’ attribute, Spring may generate the key based on method argument itself [product as a key]. Hence you must make sure to implement **hashCode() and equals()** for that modal object. In contrast, you can use [KeyGenerator](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/cache/interceptor/KeyGenerator.html) to generate a key for you.

condition : Conditional Caching. Item will be cached only if the condition mentioned in ‘condition’ met. Note that condition applies to method argument and evaluated before method execution.

|  |
| --- |
| @Cacheable(value="products", key="#product.name", condition="#product.price<500")  public Product findProduct(Product product){  ..  return aproduct;  } |

unless :Conditional Caching, applies to return value of method. Item will be cached, unless the condition mentioned in ‘unless’ met. Note that condition applies to return value of method.#result refers to method return value.

|  |
| --- |
| @Cacheable(value="products", key="#product.name", condition="#product.price<500", unless="#result.outofstock")  public Product findProduct(Product product){  ..  return aproduct;  } |

**Multiple Caches:**@Cacheable can use multiple caches at the same time. In this situation, a requested item will be checked in all the mentioned cached and if it found in any of them, method will not be executed.If it does not exist in any of the cache, method will gets executed and it’s result will be stored in all of those caches.

|  |
| --- |
| @Cacheable({"products", "items"})  public Product findProduct(Product product) {...  ..  return aproduct;  } |

### **@CachePut**

Used for Cache-update operation. Method annotated with @CachePut are **always gets executed** and there result gets stored in the cache, eventually overriding any entry with same key in cache. @CachePut, like @Cacheable, supports several attributes, having similar functionality as described above.

Think about a product-refresh operation, where we want a specific product to be re-calculated [may be due to a new price] and then store that product in cache for any future reference. Note that while @CacheEvict is used to remove an item[or all of them] from cache, @CachePut is to update an item.

|  |
| --- |
| @CachePut(value = "products", key = "#product.name" , unless="#result==null")  public Product updateProduct(Product product) {      logger.info("<!----------Entering updateProduct ------------------->");      for(Product p : products){          if(p.getName().equalsIgnoreCase(product.getName()))              p.setPrice(product.getPrice());              return p;      }      return null;  } |

#### **@CacheEvict**

Used for Cache-removal /cache-cleanup operation. @CacheEvict annotation indicates that a method (or all methods on a class) triggers a cache evict operation, removing specific [or all] items from cache. Various attributes provides complete control to enforce the required behavior for cache-eviction.

|  |
| --- |
| @CacheEvict(value = "products", key = "#product.name")  public void refreshProduct(Product product) {      //This method will remove only this specific product from 'products' cache.  }    @CacheEvict(value = "products", allEntries = true)  public void refreshAllProducts() {      //This method will remove all 'products' from cache, say as a result of flush-all API.  } |

### **@Caching**

@Caching annotation comes handy when you want to specify multiple annotations of the same type, such as @CacheEvict or @CachePut on same method.

Let’s say you have two caches containing same product using same keys. Now, if you want to evict the specific product from both caches, it’s straight forward.

Let’s say you have two caches containing same product using same keys. Now, if you want to evict the specific product from both caches, it’s straight forward.

|  |
| --- |
| @CacheEvict(value = {"products", "items"}, key = "#product.name")  public void refreshProduct(Product product) {      //This method will remove only this specific product from 'products' & 'items' cache.  } |

But what if they are using different keys? You may think something like below would be good enough.

|  |
| --- |
| @CacheEvict(value = "products", key = "#product.name")  @CacheEvict(value = "items"  ,  key = "#product.id")  public void refreshProduct(Product product) {      //This method will remove only this specific product from 'products' & 'items' cache.  } |

Instead you will get a compiler error, as it is not allowed by the language itself to have two annotations of the same time on same element.

@Caching to the rescue.

|  |
| --- |
| @Caching(evict = {          @CacheEvict(value = "products", key="#product.name"),          @CacheEvict(value = "items"   , key = "#product.id")  })  public void refreshProduct(Product product) {      //This method will remove only this specific product from 'products' & 'items' cache.  } |

### **@CacheConfig**

@CacheConfig is a class-level annotation which can be used to specify the common caching related settings directly on class level, thus freeing user from duplicating them on each method level. You can of course override the setting specified on class level, on individual method. Common configuration setting that can be specified at class level are cache names, custom KeyGenerator, the custom CacheManager& custom CacheResolve.

|  |
| --- |
| @CacheConfig(value="products", keyGenerator="myKeyGenerator")  class MyClass{        @Cacheable      public Product findProduct(Product product) {...          ..          return aproduct;      }        @Cacheable(value="items")      public Product findSoldProduct(Product product) {...          ..          return aproduct;      }  } |

In above example, findProduct will be using “products” cache, while findSoldProduct has overwritten the cache to be used. Additionally, both of them will use the keyGenerator specified on class level.

|  |
| --- |
| Cache.xml file:  <ehcachexmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:noNamespaceSchemaLocation=*"ehcache.xsd"*updateCheck=*"true"*  monitoring=*"autodetect"*dynamicConfig=*"true"*>  <!-- By default, Ehcache stored the cached files in temp folder. -->  <!--<diskStore path="java.io.tmpdir" /> -->    <!-- Ask Ehcache to store cache in this path -->  <diskStorepath=*"c:\\cache"*/>  <!-- Sample cache named cache1  This cache contains a maximum in memory of 10000 elements, and will expire  an element if it is idle for more than 5 minutes and lives for more than  10 minutes.  If there are more than 10000 elements it will overflow to the  disk cache, which in this configuration will go to wherever java.io.tmp is  defined on your system. On a standard Linux system this will be /tmp" -->  <cachename=*"cache1"*  maxEntriesLocalHeap=*"10000"*  maxEntriesLocalDisk=*"1000"*  eternal=*"false"*  diskSpoolBufferSizeMB=*"20"*  timeToIdleSeconds=*"300"*timeToLiveSeconds=*"600"*  memoryStoreEvictionPolicy=*"LFU"*  transactionalMode=*"off"*>  <persistencestrategy=*"localTempSwap"*/>  </cache>  </ehcache>  @EnableCaching  @Configuration  @ComponentScan(basePackages = "com.example.SpringBootCache")  **publicclass**AppConfig {  **private**EhCacheManagerFactoryBean factoryBean;    @Bean  **Public** CacheManager cacheManager() {  **Return new** EhCacheCacheManager(ehCacheManager().getObject());  }    @Bean  **Public** EhCacheManagerFactoryBean ehCacheManager() {  factoryBean = **new**EhCacheManagerFactoryBean();  factoryBean.setConfigLocation(**new**ClassPathResource("ehcache.xml"));  factoryBean.setShared(**true**);  **return**factoryBean;  }  }  Handler Cheche:  @SpringBootApplication  @EnableMBeanExport  **publicclass**SpringBootLoggerApplication {  **publicstaticvoid**main(String[] args) {  SpringApplication.*run*(SpringBootLoggerApplication.**class**, args);  }    }  @Component  @ManagedResource(objectName = "EhCacheServkice-EhCache:name=EhCacheHandler", description = "Handles the Ehcache Operations")  **publicclass**EhCacheHandler {  **privatestaticfinal** Logger ***logger*** = LoggerFactory.*getLogger*(EhCacheHandler.**class**);  @Autowired  **private**EhCacheManagerFactoryBeanehCacheManagerFactoryBean;    @Autowired  **private**EhCacheCacheManagerehCacheCacheManager;  @ManagedOperation(description = "Clear all caches")  **public** String ClearAllCaches() {  getcacheManager().clearAll();  ***logger***.info("Clear all caches");  **return**"Cleared all caches";  }    @ManagedOperation(description = "Clear all caches start with name")  @ManagedOperationParameters({  @ManagedOperationParameter(name = "cacheName", description = "Clear all caches start with name") })  **public** String clearAllStartingWith(String cacheName) {  getcacheManager().clearAllStartingWith(cacheName);  ***logger***.info("Clear all caches start with name");  **return**"Cleared caches";  }  @ManagedOperation(description = "Delete cache from cache")  @ManagedOperationParameters({ @ManagedOperationParameter(name = "cacheName", description = "Delete cache from cache") })  **public** String DeleteCache(String cacheName) {  **if** (getcacheManager().cacheExists(cacheName)) {  getcacheManager().removeCache(cacheName);  ***logger***.debug("deleted {} cache from cache : {} ", cacheName);  **return**"removed cache from cache";  }  **return**"cache not exists";  }  @ManagedOperation(description = "Remove all caches from cache ")  **public** String DeleteAllCaches() {  getcacheManager().removeAllCaches();  ***logger***.debug("removed all caches from cache");  **return**"removed all caches from cache";  }  @ManagedOperation(description = "Show all caches from cache ")  **public** String ShowAllCaches() {  List<String>list = Arrays.*asList*(getcacheManager().getCacheNames());  **if**(!list.isEmpty()) {  **return**list.toString();  }  **return**"cache is empty";  }    @ManagedOperation(description = "Enable Ehcache")  **public** String disableCache() {  net.sf.ehcache.CacheManagerobj = ehCacheManagerFactoryBean.getObject();  obj.shutdown();  **return**"cache is shutdown";  }    @ManagedOperation(description = "Enable cache from cache")  @ManagedOperationParameters({ @ManagedOperationParameter(name = "cacheName", description = "Enable cache from cache") })  **publicvoid**enableCaches(String cacheName){  String[] cacheNames = getcacheManager().getCacheNames();  **for** (String cache :cacheNames) {  getcacheManager().getCache(cache).setDisabled(**false**);  }  }  @ManagedOperation(description = "Disable cache from cache")  @ManagedOperationParameters({ @ManagedOperationParameter(name = "cacheName", description = "Disable cache from cache") })  **publicvoid**disableCaches(String cacheName){  String[] cacheNames = getcacheManager().getCacheNames();  **for** (String cache :cacheNames) {  getcacheManager().getCache(cache).setDisabled(**true**);  }  }    **public** String DisableCache() {  **returnnull**;  }    /\*\*  \* Get the CacheManager  \*  \* **@return**CacheManager  \*/  **public**net.sf.ehcache.CacheManagergetcacheManager() {  **return**ehCacheManagerFactoryBean.getObject();  //return ehCacheCacheManager.getCacheManager();  }  } |

#### org.springframework.cache.CacheManager is the common Cache abstraction provided by spring to handle all caching related activities. CacheManager controls and manages Caches [ org.springframework.cache.Cache] and can be used to retrieve these for storage

**Slf4j with Logback**

[**SLF4J**](http://www.slf4j.org/) - The Simple Logging Facade for Java (SLF4J) serves as a **simple facade or abstraction**1 for various logging frameworks (e.g. java.util.logging, logback, log4j)

* It is better to use [abstraction](https://javapapers.com/core-java/java-abstraction/), always!
* You can change the implementation as you wish and need not be tied to a dependency.
* To make it clear, if you are using SLF4J along with Log4j, if required Log4j can be swapped with another logging framework like Logback without compiling the source code. Logback is the reference implementation for SLF4J.

By default, all logging goes to console.

### **Configure Logging Levels**

In application.properties, we can use the “logging.level” prefix to set logging levels.

logging.level.some.package.path=DEBUG

logging.level.some.other.package.path=ERROR

**Root logging level can be configured as shown below**

logging.level.root=WARN

### **Configuring a Log File**

You can configure a log file by using logging.file property in application.properties. The logging here would be in addition to the logging in console.

logging.file=\path\_to\logfile.log

### **Custom configuration using logback.xml**

Spring Boot will pick up all custom configuration using logback.xml as long as it is in the application class path.

dependencies to logback-core and the SLF4J API.

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

<version>1.2.3</version>

</dependency>

[**Log4j 1.2**](http://logging.apache.org/log4j/1.2/) - Welcome to Apache log4j, a **logging library** for Java.

[**Logback**](http://logback.qos.ch/) :

Logback is **intended as a successor to the popular log4j** project, picking up where log4j leaves off.

Defalut logging level is info

**Logger Message Describe:**

|  |
| --- |
| 2019-04-01 17:05:33.911 INFO 10768 --- [main] c.m.elk.SpringbootElkLoggerApplication : Starting SpringbootElkLoggerApplication v0.0.1-SNAPSHOT on WKSBAN29RBW1060 with PID 10768 (C:\elk\springboot-elk-logger-0.0.1-SNAPSHOT.jar started by Mekala.Nagendra in C:\elk)  2019-04-01 17:05:33.914 INFO 10768 --- [main] c.m.elk.SpringbootElkLoggerApplication : No active profile set, falling back to default profiles: default  2019-04-01 17:05:34.950 INFO 10768 --- [main] trationDelegate$BeanPostProcessorChecker : Bean 'org.springframework.hateoas.config.HateoasConfiguration' of type [org.springframework.hateoas.config.HateoasConfiguration$$EnhancerBySpringCGLIB$$2ac1bead] is not eligible for getting processed by all BeanPostProcessors (for example: not eligible for auto-proxying)  2019-04-01 17:05:35.462 INFO 10768 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 7777 (http)  2019-04-01 17:05:35.492 INFO 10768 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]  2019-04-01 17:05:35.492 INFO 10768 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.16] |

The above pattern print these listed log message parts with respective color coding applied:

* Date and Time — Millisecond precision.
* 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

**The available logging levels in Logback are:**

* OFF (output no logs)
* ERROR
* WARN
* INFO
* DEBUG
* TRACE

### **Appenders**

* ConsoleAppender – writes messages to the system console
* FileAppender – appends messages to a file
* RollingFileAppender – extends the FileAppender with the ability to roll over log files
* SMTPAppender – sends log messages in an email, by default only for ERROR messages
* DBAppender – adds log events to a database
* SiftingAppender – separates logs based on a runtime attribute

 RollingFileAppender, which rolls over the log file when certain conditions are met. The appender has two components:

<appendername="rollingFile"class="ch.qos.logback.core.rolling.RollingFileAppender">

<rollingPolicyclass="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">

<fileNamePattern>log-%d{yyyy-MM-dd}.log</fileNamePattern>

<maxHistory>30</maxHistory>

<totalSizeCap>3GB</totalSizeCap>

</rollingPolicy>

<triggeringPolicyclass="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">

<maxFileSize>3MB</maxFileSize>

</triggeringPolicy>

<encoder>

<pattern>[%thread] %-5level %logger{36} - %msg%n</pattern>

</encoder>

</appender>

The example above configures the fileNamePattern attribute **based on the day** – which means the name of each file contains the current date, and also that the rollover will happen daily.

Notice how we’re limiting the log data here – maxHistory is set to a value of 30, alongside a totalSizeCap of 3GB – which means that the archived logs will be kept for the past 30 days, up to a maximum size of 3GB.

Finally, the SizeBasedTriggeringPolicy defined configures the rollover of the file whenever it reaches 3 MB. Of course that’s quite a low limit, and [a mature log-viewing tool](https://stackify.com/best-log-viewer-prefix/) can certainly handle a lot more than that

|  |
| --- |
| **LoggerHandler:**  @Component  @ManagedResource(objectName="Logger-Service:name=FedexLoggerInfo",  description="update all log levels")  **publicclass**LoggerHandler {    **privatestaticfinal**ch.qos.logback.classic.Logger***logger*** = (Logger) LoggerFactory.*getLogger*(LoggerHandler.**class**);    **private** String levels = "TRACE, DEBUG, INFO, WARN, ERROR, FATAL";  @ManagedOperation(description = "Sets the level for the named logger. Values: TRACE, DEBUG, INFO, WARN, ERROR, FATAL")  @ManagedOperationParameters( {@ManagedOperationParameter(name = "loggerName", description = "Logger name"),  @ManagedOperationParameter(name = "logLevel", description = "Logger name")})  **public** String setLevel(String loggerName, String logLevel) {  String returnValue = **null**;    **if** (levels.contains(logLevel) && !logLevel.isEmpty()) {  ***logger***.info("updated {} level for the {} logger ", logLevel, loggerName);  Logger root = (Logger) LoggerFactory.*getLogger*(loggerName);  root.setLevel(ch.qos.logback.classic.Level.*toLevel*(logLevel));  returnValue = "Updated logger level for " + loggerName;  } **else** {  returnValue = "Please provide valid logger level";  }  **return**returnValue;  }  @ManagedOperation(description = "Sets the level for all loggers. Values: TRACE, DEBUG, INFO, WARN, ERROR, FATAL")  @ManagedOperationParameters({ @ManagedOperationParameter(name = "level", description = "Logging level") })  **public** String updateAllLogLevels(String logLevel) {  String returnValue = **null**;  **if** (levels.contains(logLevel) && !logLevel.isEmpty() ) {  ***logger***.info("updated {} level for all loggers ", logLevel);  Logger root = (Logger) LoggerFactory.*getLogger*("com.fedex.plefs");  root.setLevel(ch.qos.logback.classic.Level.*toLevel*(logLevel));  returnValue = "Updated all loggers";  } **else** {  returnValue = "Please provide valid logger level";  }  **return**returnValue;  }  } |

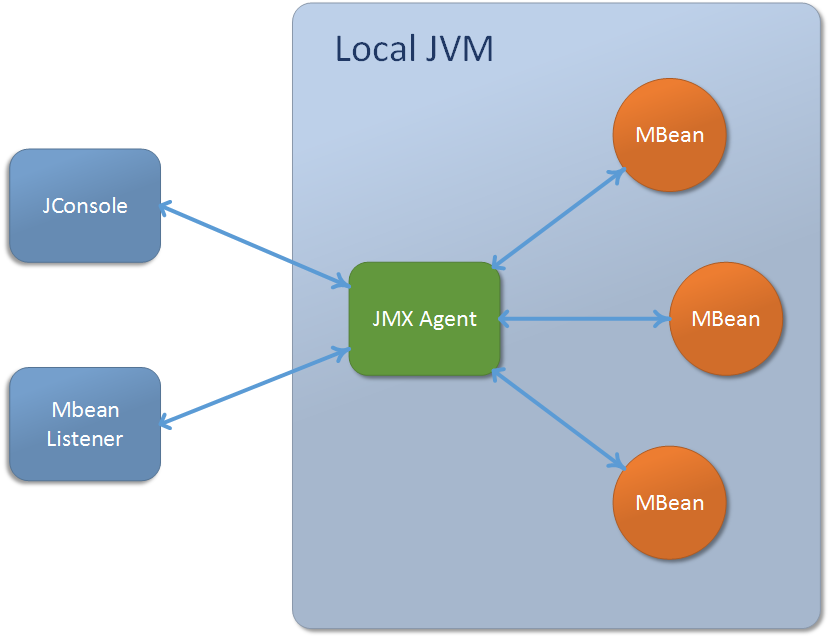
**JMX java:**

Java Management eXtension (**JMX**) is a technology to manage and monitor Java applications remotely.

Java Management Extensions (JMX) is provides managing and monitoring applications resource dynamically at runtime.

In a nutshell, these are the main participants in a JMX architecture:

* **Managed Beans (MBeans).** Can be managed and monitored remotely via JMX.
* **JMX Agent.** Manages MBeans and provides an interface so that they can be accessed remotely
* **Remote Management Application**. Monitors and interacts with the MBeans via the JMX Agent



# **JMX WITH SPRING FRAMEWORK**

managed beans (MBeans) can be easily resgistered with the JMX Agent using the annotations provided by the Spring Framework.

#### **What is CORS (Cross Origin Resource Sharing)**

Cross Origin Resource Sharing is something that is declared by the w3c on communication between different domains. By CORS, communications between the same domain will be allowed to users and the communications that are cross-originated will be restricted to a few techniques. We can see this when we are talking to APIs mostly. The REST call may give us an error. This is because the **server and the client sides are on different domains and the communication between them are restricted by CORS rules**.

**Importance of CORS**

The importance of the CORS implementation comes with the security aspects. It blocks the calls made by unknown domains and keeps the paths open only to the known domains. So the security is ensured despite the attacking requests.

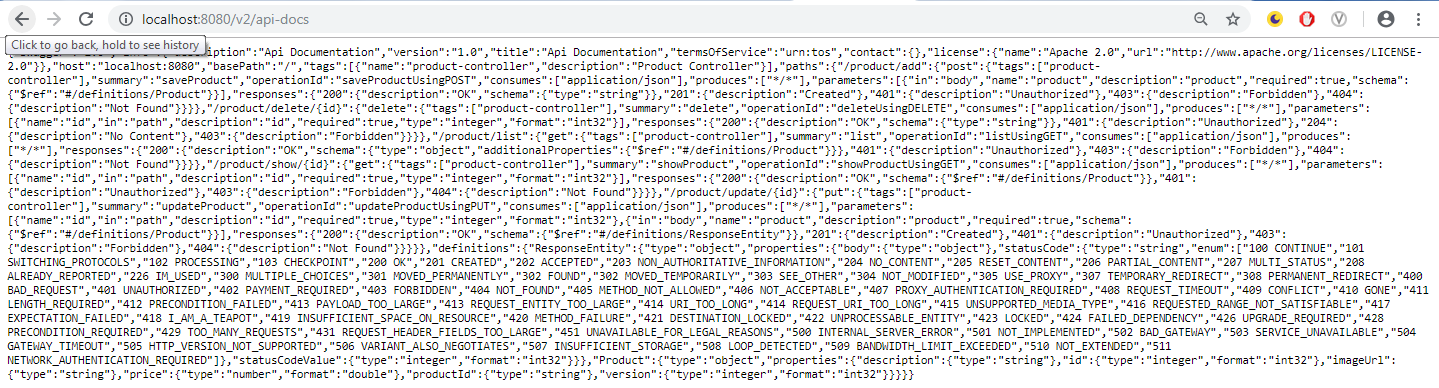
**Spring Rest full web service, Swagger and Junit test case:**

[Swagger](https://swagger.io/) (Swagger 2) is a specification for describing and documenting a REST API. It specifies the format of the REST web services including URL, Resources, methods, etc. Swagger will generate documentation from the application code and handle the rendering part as well.

|  |
| --- |
| Config  @Configuration  @EnableSwagger2  **publicclass**SwaggerConfig {    @Bean  **public** Docket productApi() {  **returnnew**Docket(DocumentationType.***SWAGGER\_2***)  .select().apis(RequestHandlerSelectors.*basePackage*("com.mng.java.controller"))  .paths(*regex*("/product.\*"))  .build().apiInfo(metaData());  }    **private**ApiInfometaData() {  ApiInfoapiInfo = **new**ApiInfo(  "Spring Boot Product REST API",  "Spring Boot Product REST API for Online Store",  "1.0",  "Terms of service",  **new**Contact("Nagendra Mekala", "https://springframework.guru/about/", "nagendra.kldm@gmail.com"),  "Apache License Version 2.0",  "https://www.apache.org/licenses/LICENSE-2.0");  **return**apiInfo;  }  }  **Controller**  RestController  @RequestMapping("/product")  @Api(value="online store", ~~description~~="Operations pertaining to products in Online Store")  @CrossOrigin  **publicclass**ProductController {  @Autowired  **private**ProductServiceproductService;  @Autowired  **publicvoid**setProductService(ProductServiceproductService) {  **this**.productService = productService;  }  //@RequestMapping(value = "/list", method= RequestMethod.GET)  @ApiResponses(value = {  @ApiResponse(code = 200, message = "Successfully retrieved list"),  @ApiResponse(code = 401, message = "You are not authorized to view the resource"),  @ApiResponse(code = 403, message = "Accessing the resource you were trying to reach is forbidden"),  @ApiResponse(code = 404, message = "The resource you were trying to reach is not found")  }  )  @ApiOperation(value="View a list of available products", response=List.**class**)  @GetMapping(value = "/list")  **public**ResponseEntity<List<Product>>list() {  List<Product>productList = productService.listAllProducts();    **if** (productList == **null** || productList.isEmpty()){  System.***out***.println("no product found");  **returnnew**ResponseEntity<List<Product>>(HttpStatus.***NO\_CONTENT***);  }    System.***out***.println("productService.listAllProducts() result: " + productList);  **returnnew**ResponseEntity<List<Product>>(productList, HttpStatus.***OK***);  }  //@RequestMapping(value = "/show/{id}", method= RequestMethod.GET)  @ApiOperation(value = "Search a product with an ID",response = Product.**class**)  @GetMapping(value = "/show/{id}")  **public**ResponseEntity<Product>showProduct(@PathVariable Integer id) {  Product product = productService.getProductById(id);    **if** (product == **null**){  System.***out***.println("no product found");  **returnnew**ResponseEntity<Product>(HttpStatus.***NO\_CONTENT***);  }    System.***out***.println("productService.getProductById() result: " + product);  **returnnew**ResponseEntity<Product>(product, HttpStatus.***OK***);  }  //@RequestMapping(value = "/add", method = RequestMethod.POST)  @ApiOperation(value = "Add a product")  @PostMapping(value = "/add")  **public**ResponseEntity<String>saveProduct(@RequestBody Product product) {  **if**(productService.exists(product.getId())) {  System.***out***.println("a product with name " + product.getId() + " already exists");  **returnnew**ResponseEntity<String>("alredayexixts", HttpStatus.***CONFLICT***);  }  productService.saveProduct(product.getId(),product);  System.***out***.println("productService.saveProduct() result: " + productService.listAllProducts());  **returnnew**ResponseEntity<String>("Product saved successfully", HttpStatus.***CREATED***);  }  //@RequestMapping(value = "/update/{id}", method = RequestMethod.PUT)  @ApiOperation(value = "Update a product")  @PutMapping(value = "/update/{id}")  **public**ResponseEntityupdateProduct(@PathVariable Integer id, @RequestBody Product product) {  Product productObj= productService.getProductById(id);    **if** (productObj == **null**){  System.***out***.println("product with id {} not found");  **returnnew**ResponseEntity<Product>(HttpStatus.***NOT\_FOUND***);  }  productObj.setProductId(product.getProductId());  productObj.setDescription(product.getDescription());  productObj.setImageUrl(product.getImageUrl());  productObj.setPrice(product.getPrice());  productService.saveProduct(product.getId(),productObj);    System.***out***.println("productService.updateProduct() result: " + productService.listAllProducts());  **returnnew**ResponseEntity("Product updated successfully", HttpStatus.***OK***);  }  //@RequestMapping(value="/delete/{id}", method = RequestMethod.DELETE)  @ApiOperation(value = "Delete a product")  @DeleteMapping(value = "/delete/{id}")  **public**ResponseEntity<String>deleteProduct(@PathVariable Integer id) {  System.***out***.println("Before productService.delete() result: " + productService.listAllProducts());  System.***out***.println("Products list size: " + productService.listAllProducts().size());    Product productObj= productService.getProductById(id);  **if**(productObj == **null**) {  System.***out***.println("Unable to delete with Id " + id + " not found");  **returnnew**ResponseEntity<String>("Unable to delete with Id",HttpStatus.***NOT\_FOUND***);  }  productService.deleteProduct(id);  System.***out***.println("after productService.delete() result: " + productService.listAllProducts());  System.***out***.println("Products list size: " + productService.listAllProducts().size());  **returnnew**ResponseEntity<String>("Product deleted successfully", HttpStatus.***OK***);  }    @GetMapping(value="/getAll",produces= {"application/json"} )  **public** List<Product>getProductList(){  **return**productService.getProductList();  }  }  Model class:  **publicclass** Product {  @ApiModelProperty(notes = "The database generated product ID")  **private** Integer id;  @ApiModelProperty(notes = "The auto-generated version of the product")  **private** Integer version;  @ApiModelProperty(notes = "The application-specific product ID")  **private** String productId;  @ApiModelProperty(notes = "The product description")  **private** String description;  @ApiModelProperty(notes = "The image URL of the product")  **private** String imageUrl;  @ApiModelProperty(notes = "The price of the product", required = **true**)  **privatedouble**price;  setters / getters  }  **Impl class:**  @Repository  **publicclass**ProductServiceImpl**implements**ProductService {    Map<Integer,Product>products = **new** HashMap<>();    @PostConstruct  **publicvoid**generateProducts() {    Product p1 = **new**Product();  p1.setId(001);  p1.setProductId("mock laptop");  p1.setDescription("mock product configuration info");  p1.setImageUrl("mock://localhost:8080/delImage");  p1.setPrice(50000);  p1.setVersion(5);  Product p2 = **new**Product();  p2.setId(002);  p2.setProductId("mock laptop");  p2.setDescription("mock product configuration info");  p2.setImageUrl("http://localhost:8080/lenovoImage");  p2.setPrice(40000);  p2.setVersion(4);  products.put(001, p1);  products.put(002, p2);    }  @Override  **public** List<Product>listAllProducts() {    List<Product>list = **new**ArrayList<Product>();  // using values() for iteration over keys  **for** (Product obj :products.values()) {  list.add(obj);  }  **return**list;  }  @Override  **public** Product getProductById(Integer id) {  **return**products.get(id);  }  @Override  **publicvoid**saveProduct(Integer id, Product product) {  products.put(id,product);  }  @Override  **publicvoid**deleteProduct(Integer id) {  products.remove(id);  }  @Override  **publicboolean**exists(Integer id) {  // **TODO** Auto-generated method stub  **return**products.containsKey(id);  }  @Override  **public** List<Product>getProductList() {    List<Product>al = **new**ArrayList<>();    Product p1 = **new**Product();  p1.setId(001);  p1.setProductId("mock laptop");  p1.setDescription("mock product configuration info");  p1.setImageUrl("mock://localhost:8080/delImage");  p1.setPrice(50000);  p1.setVersion(5);    al.add(p1);  Product p2 = **new**Product();  p2.setId(002);  p2.setProductId("mock laptop");  p2.setDescription("mock product configuration info");  p2.setImageUrl("http://localhost:8080/lenovoImage");  p2.setPrice(40000);  p2.setVersion(4);    al.add(p2);    **return**al;  }  }  Service Interface:  **publicinterface**ProductService {  List<Product>listAllProducts();  Product getProductById(Integer id);  **void**saveProduct(Integer id,Productproduct);  **void**deleteProduct(Integer id);  **boolean**exists(Integer id);  List<Product>getProductList();  }  Controller Test Class:  @RunWith(SpringRunner.**class**)  @EnableAutoConfiguration  **publicclass**ProductControllerTest {  @Mock  **private**ProductServiceproductService;  **private**MockMvcmockMvc;  @InjectMocks  **private**ProductControllerproductController;  @Before  **publicvoid**setUp() {  MockitoAnnotations.*initMocks*(**this**);  mockMvc = MockMvcBuilders.*standaloneSetup*(productController).build();  }  @AfterClass  **publicstaticvoid**tearDownAfterClass() {  }    @Test  **publicvoid**list() **throws** Exception {  List<Product>productList =**new**ArrayList<>();  Product p1 = **new**Product();  p1.setId(001);  p1.setProductId("mock laptop");  p1.setDescription("mock product configuration info");  p1.setImageUrl("mock://localhost:8080/delImage");  p1.setPrice(50000);  p1.setVersion(5);    productList.add(p1);    *when*(productService.listAllProducts()).thenReturn(productList);  MvcResultmvcResults = mockMvc  .perform(*get*("http://localhost:8080/product/list").accept(MediaType.***APPLICATION\_JSON\_UTF8\_VALUE***))  .andReturn();    String results = mvcResults.getResponse().getContentAsString();  *assertEquals*(HttpStatus.***OK***.value(), mvcResults.getResponse().getStatus());  *assertNotNull*(results);  System.***out***.println("results: " + results);  mockMvc.perform(*get*("http://localhost:8080/product/list")).andExpect(*status*().isOk())  .andExpect(*content*().contentType(MediaType.***APPLICATION\_JSON\_UTF8\_VALUE***))  .andExpect(*jsonPath*("$", *hasSize*(1)))  .andExpect(*jsonPath*("$[0].id", *is*(001)))  .andExpect(*jsonPath*("$[0].productId", *is*("mock laptop")))  .andExpect(*jsonPath*("$[0].description", *is*("mock product configuration info")))  .andExpect(*jsonPath*("$[0].imageUrl", *is*("mock://localhost:8080/delImage")))  .andExpect(*jsonPath*("$[0].price", *is*(50000.0))).andExpect(*jsonPath*("$[0].version", *is*(5)));  }    @Test  **publicvoid**showProduct() **throws** Exception {  Product p1 = **new**Product();  p1.setId(111);  p1.setProductId("mock laptop obj");  p1.setDescription("mock product configuration info msg");  p1.setImageUrl("mock://localhost:8080/delImage");  p1.setPrice(50000);  p1.setVersion(5);    *when*(productService.getProductById(111)).thenReturn(p1);  mockMvc.perform(*get*("http://localhost:8080/product/show/{id}",111).accept(MediaType.***APPLICATION\_JSON\_UTF8\_VALUE***))  .andExpect(*jsonPath*("$.productId", *is*("mock laptop obj")))  .andExpect(*jsonPath*("$.description", *is*("mock product configuration info msg")))  .andExpect(*jsonPath*("$.imageUrl", *is*("mock://localhost:8080/delImage")))  .andExpect(*jsonPath*("$.price", *is*(50000.0)))  .andExpect(*jsonPath*("$.version", *is*(5)))  .andReturn();    // verify that service method was called once  *verify*(productService).getProductById(111);    // second approach  *when*(productService.getProductById(111)).thenReturn(p1);  MvcResultmvcResults = mockMvc.perform(*get*("http://localhost:8080/product/show/{id}",111).accept(MediaType.***APPLICATION\_JSON\_UTF8\_VALUE***))  .andReturn();  *assertEquals*(HttpStatus.***OK***.value(), mvcResults.getResponse().getStatus());  *assertNotNull*(mvcResults.getResponse());  System.***out***.println("\*\*\*\*JSON REsponse\*\*\*\*\*"+ mvcResults.getResponse());    System.***out***.println("response status:" + mvcResults.getResponse().getStatus());  System.***out***.println("results: " + mvcResults.getResponse().getContentAsString());  }    @Test  **publicvoid**saveProduct() **throws** Exception {  Product p1 = **new**Product();  p1.setId(111);  p1.setProductId("mock laptop obj");  p1.setDescription("mock product configuration info msg");  p1.setImageUrl("mock://localhost:8080/delImage");  p1.setPrice(50000);  p1.setVersion(5);    String exJson ="{\"id\":111,\"version\":5,\"productId\":\"mock laptop obj\",\"description\":\"mock product configuration info msg\",\"imageUrl\":\"mock://localhost:8080/delImage\",\"price\":50000.0}";    *doAnswer*((i) ->{  System.***out***.println("\*\*\*\*do answer method\*\*\*\*\*\*\*: "+i);  **returnnull**;  }).when(productService).saveProduct(101, p1);;    *when*(productService.exists(101)).thenReturn(**true**);    // Send course as body to /students/Student1/courses  RequestBuilderrequestBuilder = MockMvcRequestBuilders  .*post*("http://localhost:8080/product/add")  .accept(MediaType.***APPLICATION\_JSON***).content(exJson)  .contentType(MediaType.***APPLICATION\_JSON***);    MvcResultmvcResult = mockMvc.perform(requestBuilder).andReturn();    *assertEquals*("Product saved successfully", mvcResult.getResponse().getContentAsString());  *assertEquals*(HttpStatus.***CREATED***.value(), mvcResult.getResponse().getStatus());  }  @Test  **publicvoid**updateProduct() **throws** Exception {  Product p1 = **new**Product();  p1.setId(111);  p1.setProductId("mock laptop obj");  p1.setDescription("mock product configuration info msg");  p1.setImageUrl("mock://localhost:8080/delImage");  p1.setPrice(50000);  p1.setVersion(5);    String exJson ="{\"id\":111,\"version\":5,\"productId\":\"mock laptop obj\",\"description\":\"mock product configuration info msg\",\"imageUrl\":\"mock://localhost:8080/delImage\",\"price\":50000.0}";    *doAnswer*((i) ->{  System.***out***.println("\*\*\*\*do answer method\*\*\*\*\*\*\*: "+i);  **returnnull**;  }).when(productService).saveProduct(101, p1);;    *when*(productService.getProductById(101)).thenReturn(p1);      // Send course as body to /students/Student1/courses  RequestBuilderrequestBuilder = MockMvcRequestBuilders  .*put*("http://localhost:8080/product/update/{id}",101)  .accept(MediaType.***APPLICATION\_JSON***).content(exJson)  .contentType(MediaType.***APPLICATION\_JSON***);    MvcResultmvcResult = mockMvc.perform(requestBuilder).andReturn();    *assertEquals*("Product updated successfully", mvcResult.getResponse().getContentAsString());  *assertEquals*(HttpStatus.***OK***.value(), mvcResult.getResponse().getStatus());  }    @Test  **publicvoid**deleteProduct() **throws** Exception {  Product p1 = **new**Product();  p1.setId(111);  p1.setProductId("mock laptop obj");  p1.setDescription("mock product configuration info msg");  p1.setImageUrl("mock://localhost:8080/delImage");  p1.setPrice(50000);  p1.setVersion(5);    *doAnswer*((i) ->{  System.***out***.println("\*\*\*\*do answer method\*\*\*\*\*\*\*: "+i);  **returnnull**;  }).when(productService).deleteProduct(101);;    *when*(productService.getProductById(101)).thenReturn(p1);    MvcResultmvcResults = mockMvc  .perform(*delete*("http://localhost:8080/product/delete/{id}",101).accept(MediaType.***APPLICATION\_JSON\_UTF8\_VALUE***))  .andReturn();    *assertEquals*("Product deleted successfully", mvcResults.getResponse().getContentAsString());  *assertEquals*(HttpStatus.***OK***.value(), mvcResults.getResponse().getStatus());  }  @Test  **publicvoid**getProductList() **throws** Exception {  List<Product>al = **new**ArrayList<>();  Product p1 = **new**Product();  p1.setId(001);  p1.setProductId("mock laptop");  p1.setDescription("mock product configuration info");  p1.setImageUrl("mock://localhost:8080/delImage");  p1.setPrice(50000.0);  p1.setVersion(5);  al.add(p1);  *when*(productService.getProductList()).thenReturn(al);  mockMvc.perform(*get*("http://localhost:8080/product/getAll")).andExpect(*status*().isOk())  .andExpect(*content*().contentType(MediaType.***APPLICATION\_JSON\_UTF8\_VALUE***))  .andExpect(*jsonPath*("$", *hasSize*(1))).andExpect(*jsonPath*("$[0].id", *is*(001)))  .andExpect(*jsonPath*("$[0].productId", *is*("mock laptop")))  .andExpect(*jsonPath*("$[0].description", *is*("mock product configuration info")))  .andExpect(*jsonPath*("$[0].imageUrl", *is*("mock://localhost:8080/delImage")))  .andExpect(*jsonPath*("$[0].price", *is*(50000.0))).andExpect(*jsonPath*("$[0].version", *is*(5)));  }  } |

* @RestController was introduced in Spring MVC 4 and is a convenience annotation that itself is annotated with @Controller and @ResponseBody annotation. This eliminates the need of annotating your controller methods with @ResponseBody individually.
* @RequestMapping annotation is used to map URLs such as /users onto an entire class or a particular handler method. A class-level annotation like /users maps a specific request path onto a controller, with additional method-level annotations further narrowing the mapping for a specific HTTP request method like “GET”, “PUT”, “POST” or “DELETE” etc.
* RequestMethod is an enumeration of possible HTTP request methods like GET, PUT, POST, DELETE, etc.
* @PathVariable annotation indicates that a method parameter should be bound to a URI template variable. To process the @PathVariable annotation, Spring MVC needs to find the matching URI template variable by name. You can specify it in the annotation or, if the URI template variable name matches the method argument name, you can omit that detail.
* @RequestBody annotation indicates a method parameter should be bound to the body of the HTTP web request. Behind the scene, a HttpMessageConverter is responsible for converting from the HTTP request message to an object and converting from an object to the HTTP response body.
* @ResponseBody annotation indicates a method return value should be bound to the HTTP response body. By annotating your class with @RestController, you no longer need to add this annotation to every method individually.
* ResponseEntity extends from HttpEntity which allows you to add a HttpStatus code directly to the response. The ResponseEntity represents the entire HTTP response. You can add header information, status codes and add content to the body.
* HttpHeaders represents HTTP request and response headers. This class has some convenience methods for setting popular header types like Content-Type, Access-Control-Allow-Headers, etc.

On pointing your browser to http://localhost:8080/swagger-ui.html, you will see the generated documentation rendered by Swagger UI, like this:



**Spring test and DBUnit framework**

Spring DBUnit provides integration between the Spring testing framework and the popular DBUnit project. It allows you to setup and teardown database tables using simple annotations as well as checking expected table contents once a test completes.

The project can be configured to run DBUnit tests using a Spring TestExecutionListener.

|  |
| --- |
| @RunWith(SpringJUnit4ClassRunner.**class**)  @EnableAutoConfiguration  @ContextConfiguration(locations = { "classpath:test-application-context.xml" })  @TestExecutionListeners({ DependencyInjectionTestExecutionListener.**class**, DirtiesContextTestExecutionListener.**class**,  TransactionDbUnitTestExecutionListener.**class** })  @DbUnitConfiguration(dataSetLoader = DbUnitDataReplacer.**class**)  @Transactional  @Rollback  Public class DbunitExample {  @Test  @DatabaseSetup({ "/dbunit-data/emp.dbunit.data.xml", "/dbunit-data/dept.dbunit.data.xml" })  Public void testEmplTable {  ------------  ---------------  ---------------  }  } |

**Dependency:**

We can get the required dependencies with Maven by declaring the following dependencies in our pom.xml file:

* [JUnit](http://junit.org/) (version 4.11).
* [AssertJ Core](http://joel-costigliola.github.io/assertj/assertj-core.html) (version 3.2.0). We use AssertJ for ensuring that the tested method returns the correct information.
* [Spring Test](http://docs.spring.io/spring/docs/4.1.x/spring-framework-reference/html/testing.html) (version 4.1.6.RELEASE).
* [DbUnit](http://dbunit.sourceforge.net/) (version 2.5.1). Remember to exclude the JUnit dependency. We use DbUnit for initializing our database into a known state before each test case is invoked.
* [Spring Test DbUnit](http://springtestdbunit.github.io/spring-test-dbunit/) (version 1.2.1) integrates DbUnit with the Spring Test framework.

**Configuring Our Integration Tests**

We can configure our integration tests by following these steps:

1. Run integration tests by using the *SpringJUnit4ClassRunner* class. It is a custom JUnitrunner that integrates theSpring Test framework with JUnit. We can configure the used JUnit runner by annotating our test class with the @RunWith annotation.
2. Configure the application context configuration class (or XML configuration file) that configures the application context used by our integration tests.We can configure the used application context configuration class (or XML configuration file) by annotating our test class with the @ContextConfiguration annotation.

1. Configure the test execution listeners which react to the test execution events that are published by the Spring Test framework. We have to configure the following test execution listeners:
   * The DependencyInjectionTestExecutionListener provides dependency injection for the test object.
   * The TransactionalTestExecutionListener adds transaction support (with default rollback semantics) into our integration tests.
   * The DbUnitTestExecutionListener adds support for the features provided by the Spring Test DbUnit library.

<beanid="dataSource"class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<propertyname="driverClassName"value="org.hsqldb.jdbcDriver"/>

<propertyname="url"value="jdbc:hsqldb:mem:paging"/>

<propertyname="username"value="sa"/>

<propertyname="password"value=""/>

</bean>

# Setup and TearDown

The @DatabaseSetup and @DatabaseTearDown annotations can be used to configure database table before tests execute and reset them once tests have completed.

Here is a typical setup annotation. In this case a file named sampleData.xml is contained in the same package as the test class.

@DatabaseSetup("sampleData.xml")

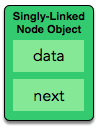
It is also possible to reference specific resource locations, for example:

@DatabaseSetup("/META-INF/dbtest/sampleData.xml")

By default setup will perform a CLEAN\_INSERT operation, this means that all data from tables referenced in the DataSet XML will be removed before inserting new rows. The standard

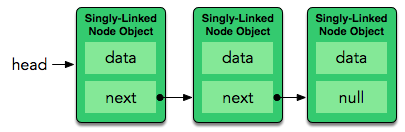
**Singly-LinkedList:**

A singly-linked list is a data structure having a list of elements where each element has a reference pointing to the next element in the list. Its elements are generally referred to as *nodes*; each node has a *data* field containing a data value and a *next* field pointing to the next element in the list (or null if it is the last element in the list).



The image below depicts a singly-linked Node object:

A singly-linked list of length  would look like this:



Observe that the  pointer for the last element in the list contains a null reference.

**Doubly-linkedList:**

This is the same as a singly-linked list, except that each node has an additional pointer to the previousnode. This allows you to traverse the list in both the forward and backward directions.

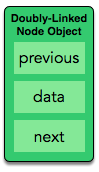
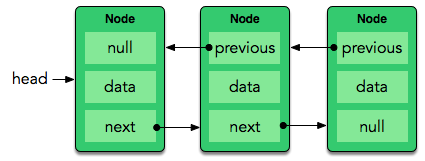


image below depicts a doubly-linked Node object:

A doubly-linked list of length  would look like this:



Observe that theprevious  pointer for the first element in the list and the  pointer for the last element in the list both contain null references.