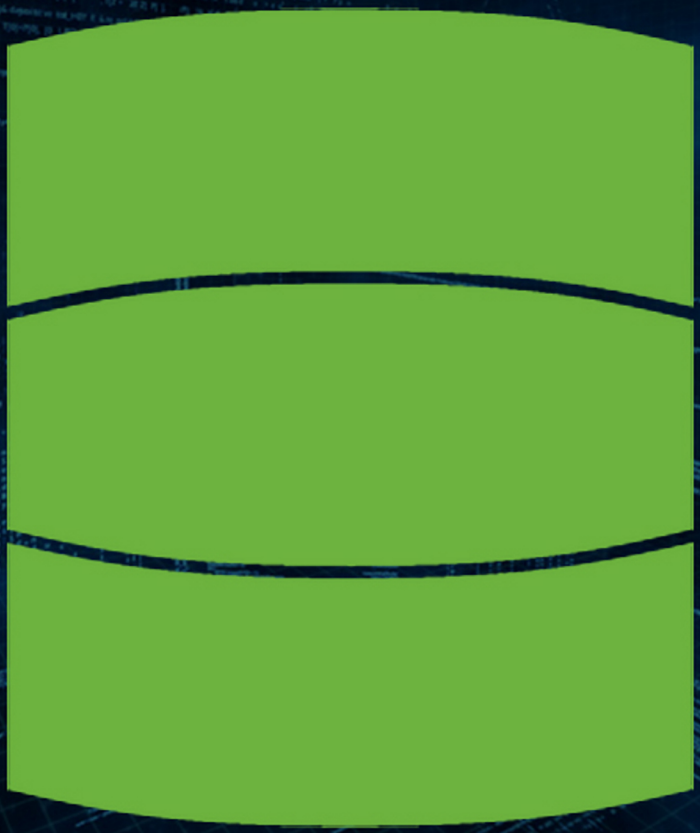
SPRING DATA PROGRAMMING COOKBOOK

Hot Recipes for Spring Data Development



Java Code Geeks
Java 2 Java Developers Resource Center

Spring Data Programming Cookbook

Contents

Preface

Spring Data's mission is to provide a familiar and consistent, Spring-based programming model for data access while still retaining the special traits of the underlying data store.

It makes it easy to use data access technologies, relational and non-relational databases, map-reduce frameworks, and cloud-based data services. This is an umbrella project which contains many subprojects that are specific to a given database. The projects are developed by working together with many of the companies and developers that are behind these exciting technologies. (Source: https://projects.spring.io/spring-data/)

In this ebook, we provide a compilation of Spring Data examples that will help you kick-start your own projects. We cover a wide range of topics, from setting up the environment and creating a basic project, to handling the various modules (e.g. JPA, MongoDB, Redis etc.). With our straightforward tutorials, you will be able to get your own projects up and running in minimum time.

About the Author

JCGs (Java Code Geeks) is an independent online community focused on creating the ultimate Java to Java developers resource center; targeted at the technical architect, technical team lead (senior developer), project manager and junior developers alike.

JCGs serve the Java, SOA, Agile and Telecom communities with daily news written by domain experts, articles, tutorials, reviews, announcements, code snippets and open source projects.

You can find them online at https://www.javacodegeeks.com/

Chapter 1

Beginners Tutorial

In this example, we shall demonstrate how to configure Spring Framework to communicate with database using JPA and Hibernate as the JPA vendor. The benefits of using Spring Data is that it removes a lot of boiler-plate code and provides a cleaner and more readable implementation of `DAO` layer. Also, it helps make the code loosely coupled and as such switching between different JPA vendors is a matter of configuration.

So let's set-up the database for the example. We shall use the MySQL database for this demo. We create a table "employee" with 2 columns as shown:

```
CREATE TABLE `employee` (
`employee_id`  bigint(20) NOT NULL AUTO_INCREMENT,
`employee_name`  varchar(40) ,
PRIMARY KEY (`employee_id`)
)
```

Now that the table is ready, let's have a look at the libraries we will require for this demo :

- antlr-2.7.7
- aopalliance-1.0
- commons-collections-3.2
- commons-logging-1.1
- dom4j-1.6.1
- hibernate-commons-annotations-4.0.2.Final
- hibernate-core-4.2.6.Final
- hibernate-entitymanager-4.2.6.Final
- hibernate-jpa-2.0-api-1.0.1.Final
- javaee-api-5.0-2
- javassist-3.15.0-GA
- jboss-logging-3.1.0.GA
- jta
- log4j-1.2.14
- mysql-connector-java-5.1.11-bin
- slf4j-api-1.5.6

- slf4j-log4j12-1.5.6
- spring-aop-3.2.4.RELEASE
- spring-beans-3.2.4.RELEASE
- spring-context-3.2.4.RELEASE
- spring-context-support-3.2.4.RELEASE
- spring-core-3.2.4.RELEASE
- spring-expression-3.2.4.RELEASE
- spring-jdbc-3.2.4.RELEASE
- spring-orm-3.2.4.RELEASE
- spring-tx-3.2.4.RELEASE

And here's the project structure :

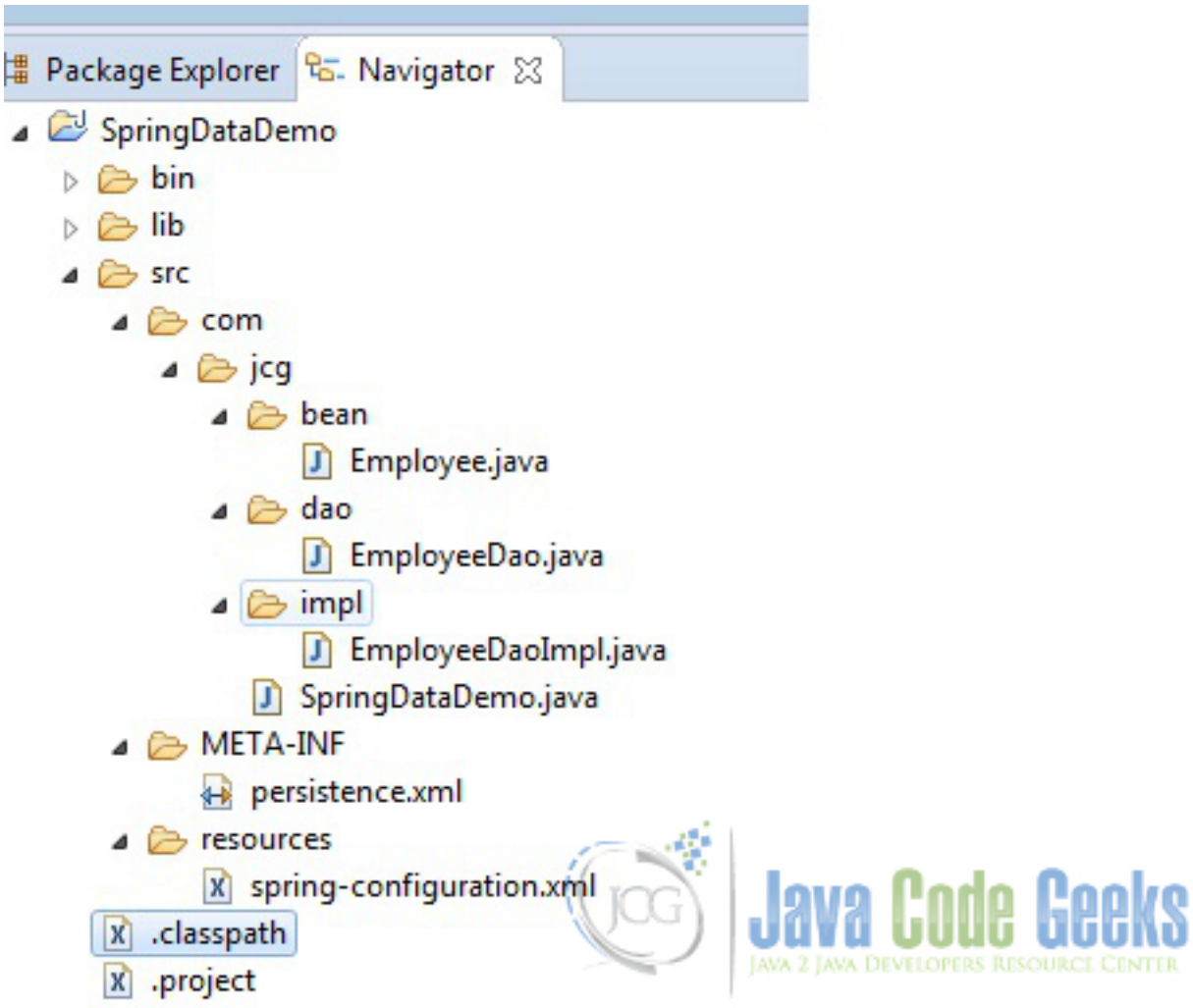


Figure 1.1: Project Structure

Now that the project is all set, we will start writing the code.

First of all, we create the `Employee` class with `employeeId` and `employeeName`. The Person class will be the entity that we will store and retrieve from the database using the JPA.

The `@Entity` marks the class as the JPA Entity. We map the properties of the Employee class with the columns of the employee table and the entity with employee table itself using the `@Table` annotation.

The `toString` method is over-ridden so that we get a meaningful output when we print the instance of the class.

Employee.java

```
package com.jcg.bean;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="employee")
public class Employee
{
	@Id
	@GeneratedValue(strategy=GenerationType.AUTO)
	@Column(name = "employee_id")
	private long employeeId;

	@Column(name="employee_name")
	private String employeeName;

	public Employee()
	{
	}

	public Employee(String employeeName)
	{
		this.employeeName = employeeName;
	}

	public long getEmployeeId()
	{
		return this.employeeId;
	}

	public void setEmployeeId(long employeeId)
	{
		this.employeeId = employeeId;
	}

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

	public void setEmployeeName(String employeeName)
	{
		this.employeeName = employeeName;
	}


	@Override
	public String toString()
```

```
        {
                return "Employee [employeeId=" + this.employeeId + ", employeeName=" + this ↩
                    .employeeName + "]";
        }
}
```

Once the Entity is ready, we define the interface for the storage and retrieval of the entity i.e. we shall create a Data Access Interface.

EmployeeDao.java

```
package com.jcg.dao;

import java.sql.SQLException;

import com.jcg.bean.Employee;

public interface EmployeeDao
{
        void save(Employee employee) throws SQLException;

        Employee findByPrimaryKey(long id) throws SQLException;
}
```

We will then, attempt to implement the Data Access Interface and create the actual Data Access Object which will modify the `Person Entity`.

EmployeeDaoImpl.java

```
package com.jcg.impl;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

import com.jcg.bean.Employee;
import com.jcg.dao.EmployeeDao;

@Repository("EmployeeDaoImpl")
@Transactional(propagation = Propagation.REQUIRED)
public class EmployeeDaoImpl implements EmployeeDao
{
        @PersistenceContext
        private EntityManager entityManager;


        @Override
        public void save(Employee employee)
        {
                        entityManager.persist(employee);
        }

        @Override
        public Employee findByPrimaryKey(long id)
        {
                        Employee employee = entityManager.find(Employee.class, id);

                        return employee;
        }
```

```
        /**
         * @return the entityManager
         */
        public EntityManager getEntityManager()
        {
                return entityManager;
        }

        /**
         * @param entityManager the entityManager to set
         */
        public void setEntityManager(EntityManager entityManager)
        {
                this.entityManager = entityManager;
        }
}
```

The DAO Implementation class is annotated with `@Repository` which marks is as a Repository Bean and prompts the Spring Bean Factory to load the Bean. `@Transactional` asks the container to provide a transaction to use the methods of this class. `Propagation.REQUIRED` denotes that the same transaction is used if one is available when multiple methods which require transaction are nested. The container creates a single Physical Transaction in the Database and multiple Logical transactions for each nested method. However, if a method fails to successfully complete a transaction, then the entire physical transaction is rolled back. One of the other options is `Propagation.REQUIRES_NEW`, wherein a new physical transaction is created for each method. There are other options which help in having a fine control over the transaction management.

The `@PersistenceContext` annotation tells the container to inject an instance of `entitymanager` in the DAO. The class implements the `save` and `findbyPk` methods which save and fetch the data using the instance of `EntityManager` injected.

Now we define our persistence Unit in the `Persistence.xml` which is put in the `META-INF` folder under `src`. We then mention the class whose instances are to be used Persisted. For this example, it is the `Employee Entity` we created earlier.

persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="https://java.sun.com/xml/ns/persistence"
version="1.0">
<persistence-unit name="jcgPersistence" transaction-type="RESOURCE_LOCAL" >
<class>com.jcg.bean.Employee</class>
</persistence-unit>
</persistence>
```

Now we configure the Spring Container using the `spring-configuration.xml` file.

spring-configuration.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="https://www.springframework.org/schema/beans"
        xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance" xmlns:aop="https://www. ↩
            springframework.org/schema/aop"
        xmlns:context="https://www.springframework.org/schema/context" xmlns:tx="https:// ↩
            www.springframework.org/schema/tx"
        xsi:schemaLocation="https://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans-3.0.xsd
https://www.springframework.org/schema/aop
https://www.springframework.org/schema/aop/spring-aop-3.0.xsd
https://www.springframework.org/schema/context
https://www.springframework.org/schema/context/spring-context-3.0.xsd
https://www.springframework.org/schema/tx
https://www.springframework.org/schema/tx/spring-tx.xsd">

        <context:component-scan base-package="com.jcg" />

        <bean id="dataSource"
```

```
                class="org.springframework.jdbc.datasource.DriverManagerDataSource">
                <property name="driverClassName" value="com.mysql.jdbc.Driver" />
                <property name="url" value="jdbc:mysql://localhost:3306/jcg" />
                <property name="username" value="root" />
                <property name="password" value="toor" />
        </bean>

        <bean id="jpaDialect" class="org.springframework.orm.jpa.vendor.HibernateJpaDialect ↩
            " />

        <bean id="entityManagerFactory"
                class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
                <property name="persistenceUnitName" value="jcgPersistence" />
                <property name="dataSource" ref="dataSource" />
                <property name="persistenceXmlLocation" value="META-INF/persistence.xml" />
                <property name="jpaVendorAdapter" ref="jpaVendorAdapter" />
                <property name="jpaDialect" ref="jpaDialect" />
                <property name="jpaProperties">
         <props>
            <prop key="hibernate.hbm2ddl.auto">validate</prop>
            <prop key="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</prop>
         </props>
      </property>
        </bean>

        <bean id="jpaVendorAdapter"
                class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
        </bean>

        <bean id="txManager" class="org.springframework.orm.jpa.JpaTransactionManager">
                <property name="entityManagerFactory" ref="entityManagerFactory" />
                <property name="dataSource" ref="dataSource" />
                <property name="jpaDialect" ref="jpaDialect" />
        </bean>

        <tx:annotation-driven transaction-manager="txManager" />

</beans>
```

We define the beans we need in the `spring-configuration.xml`. The datasource contains the basic configuration properties like URL, user-name, password and the JDBC Driver class-name.

We create a `EntityManagerFactory` using the `LocalContainerEntityManagerFactoryBean`. The properties are provided like the datasource, persistenceUnitName, persistenceUnitLocation, dialect etc. The instance of `EntityManager` gets injected form this `FactoryBean` into the `EmployeeDaoImpl` instance.

Line 51 in the above XML asks the Spring container to manage Transactions using the Spring Container. The TransactionManagerProvider Class is the `JpaTransactionManager` Class.

Now that we have completed all the hard-work, its time to test the configuration:

The `SpringDataDemo` class extracts the `EmployeeDaoImpl` and attempts to save an instance of `Employee` to the employee table and retrieve the same instance from the database.

SpringDataDemo.java

```
package com.jcg;


import java.sql.SQLException;

import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
```

```
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.jcg.bean.Employee;
import com.jcg.dao.EmployeeDao;


public class SpringDataDemo
{
        public static void main(String[] args)
        {
                try
                {
                        ApplicationContext context = new ClassPathXmlApplicationContext(" ↩
                            resources\\spring-configuration.xml");

                        //Fetch the DAO from Spring Bean Factory
                        EmployeeDao employeeDao = (EmployeeDao)context.getBean(" ↩
                            EmployeeDaoImpl");
                        Employee employee = new Employee("Employee123");
                        //employee.setEmployeeId("1");

                        //Save an employee Object using the configured Data source
                        employeeDao.save(employee);
                        System.out.println("Employee Saved with EmployeeId "+employee. ↩
                            getEmployeeId());

                        //find an object using Primary Key
                        Employee emp = employeeDao.findByPrimaryKey(employee.getEmployeeId ↩
                            ());
                        System.out.println(emp);

                        //Close the ApplicationContext
                        ((ConfigurableApplicationContext)context).close();
                }
                catch (BeansException | SQLException e)
                {
                        e.printStackTrace();
                }
        }
}
```

1.1 Output

```
log4j:WARN No appenders could be found for logger (org.springframework.core.env. ↩
    StandardEnvironment).
log4j:WARN Please initialize the log4j system properly.
Employee Saved with EmployeeId 8
Employee [employeeId=8, employeeName=Employee123]
```

As you can see the employee gets saved and we are able to retrieve the employee Object we saved.

1.2 Download the Source Code

Thus we understood how to configure JPA with Spring and what are the benefits of Spring with JPA over vanilla JPA.

Download You can download the source code of this example here: **SpringDataDemo.zip**

Chapter 2

Gemfire Example

In the previous example, we demonstrated how we can connect to a Relational/SQL database with Spring using the Spring Data. In this example we will see how we can connect the Gemfire using the Spring Data.

Gemfire is a highly scalable, low-latency, in-memory data management platform which stores data in the key-value form. The data maybe persisted onto the disk. Gemfire provides it own set of tools for data management - gfsh, a command line utility tool. But it offers very limited set of control and maneuverability. Spring provides better control through Spring Gemfire module which simplify the Gemfire Data Grid configuration.

So let's start with an example to see how we can achieve the same.

The first step is to setup the `GemFire Cache`. The Cache can be setup using cache.xml or with the help of Spring Data GemFire's XML namespace. However, the preferred way is to use the Spring IoC as it offers a number of benefits from the configuration as well the ease of development point of view.The configuration advantages include Spring `FactoryBean` pattern, modular XML configuration, property placeholders so that the configuration can be externalized etc. The development benefits include auto code completions, real time validations when using intelligent IDEs like eclipse and STS. Given the above benefits, we shall proceed with the example by bootstrapping the GemFire Data Grid through Spring Container.

So we proceed to define the `gfshBean.xml` which contains the basic configuration information for the GemFire Data Grid.

gfshBean.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="https://www.springframework.org/schema/beans"
        xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance" xmlns:context="https://www. ↩
            springframework.org/schema/context"
        xmlns:gfe-data="https://www.springframework.org/schema/data/gemfire"
        xmlns:gfe="https://www.springframework.org/schema/gemfire"
        xsi:schemaLocation="https://www.springframework.org/schema/context https://www. ↩
            springframework.org/schema/context/spring-context-3.0.xsd https://www. ↩
            springframework.org/schema/beans https://www.springframework.org/schema/beans/ ↩
            spring-beans-3.0.xsd https://www.springframework.org/schema/data/gemfire https: ↩
            //www.springframework.org/schema/data/gemfire/spring-data-gemfire.xsd https:// ↩
            www.springframework.org/schema/gemfire https://www.springframework.org/schema/ ↩
            gemfire/spring-gemfire.xsd">

        <!--Spring Components Scan-->
        <context:component-scan base-package="com.jcg.examples"></context:component-scan>

        <!--GemFire Cache -->
        <gfe:cache />

        <!--Region for being used by the Record Bean -->
        <gfe:replicated-region persistent="true" id="record" />

        <!--Scan for annotated GemFire Repositories-->
```

```
        <gfe-data:repositories base-package="com.jcg.examples" />

</beans>
```

- **Line 11:** Basic configuration to create a GemFire `Cache`.
- **Line 15:** Create a GemFire `Region` with type as `replicated`. The `persistent` marks the data to be maintained on the disk as well. The default value is false. It needs to be written to disk when fail-safety is required.
- **Line 18:** Scan the packages for initializing GemFire Bean Repositories.

Now that the GemFire Data Grid is configured we can create a PoJo to map to the GemFire `Region`.

RecordBean.java

```
package com.jcg.examples.bean;


import java.io.Serializable;

import org.springframework.data.annotation.Id;
import org.springframework.data.annotation.PersistenceConstructor;
import org.springframework.data.gemfire.mapping.Region;


@Region("record")
public class RecordBean implements Serializable
{


                private static final long serialVersionUID = 3209342518270638000L;

                @Id
                private String recordId;

                private String recordString;

                public RecordBean(){}

                @PersistenceConstructor
                public RecordBean(String recordId, String recordString)
                {
                                this.recordId = recordId;
                                this.recordString = recordString;
                }

                public String getRecordId()
                {
                                return recordId;
                }

                public void setRecordId(String recordId)
                {
                                this.recordId = recordId;
                }

                public String getRecordString()
                {
                                return recordString;
                }
```

```
                public void setRecordString(String recordString)
                {
                        this.recordString = recordString;
                }

                @Override
                public String toString()
                {
                        return "RecordBean [Record Id=" + recordId + ", Record  ↩
                            String=" + recordString + "]";
                }

}
```

The annotation `@Region("record")` is required to tell the Container as to which Region the PoJo maps to, same as we map a PoJo to a table in a relational Database.

The `@Id` annotation marks the property that is to be used as Cache Key for retrieving the values.

The `@PersistenceConstructor` constructor tells the Spring Container as to which `constructor` should be used for creation of entities. When the PoJo has only one `constructor` the annotation is not required. The `toString` method is used to display the bean properties.

Now that the PoJo is ready we need to create the DAO layer. The `RecordRepository` interface does the job of this.

RecordRepository.java

```
package com.jcg.examples.repository;

import java.util.Collection;

import org.springframework.data.gemfire.repository.Query;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

import com.jcg.examples.bean.RecordBean;

/**
 * @author Chandan Singh
 *
 */
@Repository
public interface RecordRepository extends CrudRepository<RecordBean, Integer> {

        RecordBean findByRecordId(String recordId);

        @Query("SELECT * FROM /record")
  Collection<RecordBean> myFindAll();

}
```

The Spring Data provides a number of inbuilt method for manipulating the Data. We need not write the queries for basic data manipulation and reading. It is achieved by extending the `CrudRepository` and declaring the proper `Generics` as per the PoJo.In case the Developer is not satisfied with the existing method, he can create his own method by specifying the Query using the `@query` annotation.

The Spring IoC Container creates an instance of this `Repository` and makes it available to be used as a `Bean`.

Now that all is set, let's run the application and test out the code! Here's Application class that loads the XML file to instantiate the Spring Container and power up the GemFire Data Grid.

Application.java

```
package com.jcg.examples.main;

import java.util.Iterator;

import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.core.io.ClassPathResource;

import com.jcg.examples.bean.RecordBean;
import com.jcg.examples.repository.RecordRepository;


public class Application
{

                public static void main(String[] args)
                {
                        ClassPathXmlApplicationContext context = new  ↩
                            ClassPathXmlApplicationContext();
                        context.setConfigLocation(new ClassPathResource("resources/gfshBean ↩
                            .xml").getPath());
                        context.refresh();
                        RecordRepository recordRepository = context.getBean( ↩
                            RecordRepository.class);
                        RecordBean recordBean = new RecordBean("1", "One");
                        recordRepository.save(recordBean);
                        System.out.println("Successful run!!");

                        RecordBean recordBeanFetched = recordRepository.findByRecordId("2") ↩
                            ;
                        System.out.println("The Fetched record bean is "+recordBeanFetched) ↩
                            ;

                        Iterable<RecordBean> recordCollection = recordRepository.myFindAll ↩
                            ();
                        System.out.println("RecordBeans List : ");
                        for (Iterator<RecordBean> iterator = recordCollection.iterator();  ↩
                            iterator.hasNext();)
                        {
                                RecordBean recordBeannew = (RecordBean) iterator.next();
                                System.out.println(recordBeannew);

                        }
                }
}
```

We create instances of the `RecordBean` and save them using the `RecordRepository` we configured earlier. Then we proceed to fetch the saved data in various ways. The data gets persisted onto the disk, so even if we run the Application Class multiple times the record returns the data saved in the previous runs.

Here's the sample output of the program :

```
Successful run!!
The Fetched record bean is RecordBean [Record Id=2, Record String=Two]
RecordBeans List :
RecordBean [Record Id=3, Record String=Three]
RecordBean [Record Id=2, Record String=Two]
RecordBean [Record Id=1, Record String=One]
[info 2015/07/18 23:22:28.298 IST  tid=0xb] VM is exiting - shutting down distributed  ↩
    system
```

2.1 Download the Source Code

Here we demonstrated how to configure and manage a GemFire Data Repository using Spring Data.

Download You can download the full source code of this example here: **GemFireSpringData.zip**

Chapter 3

Cassandra Example

In the previous example, we demonstrated the configuration of Spring Data with GemFire, an in-memory NoSQL Database. So continuing on the same series, in this example we shall demonstrate how to connect Spring Data Application to Apache Cassandra, a Column based NoSql Database.

Cassandra is a Distributed Database Management System that can handle large amounts of data with data replication across multiple data-centres so that there is no single point of failure. It uses `CQL` as its query language which has syntax quite similar to its homonym `SQL`.

As Cassandra is a new technology, it has its own set of challenges and learning curve. To help with this, `Spring Data` hides the complexity of writing queries and other configuration stuff. Spring Data Cassandra offers the users a familiar interface to those who have used Spring Data in the past.

3.1 Implementation

We need to install Cassandra Database Server, first. For Windows System, Cassandra can be downloaded from plannetcassandra and for Linux System it can be downloaded from the Apache Mirrors.

Once Cassandra is up and running, we need to create a key-space which corresponds to the schema in the `RDBMS` world. We then create a column family in cassandra which can be colloquially referred to a `RDMBS` Table. Then, we need to have following JAR Files to connect to Cassandra Server:

- aopalliancejar
- cassandra-driver-core.jar
- commons-logging.jar
- guava.jar
- metrics-core.jar
- slf4j-api.jar
- spring-aop-RELEASE.jar
- spring-beans-RELEASE.jar
- spring-context-RELEASE.jar
- spring-core-RELEASE.jar
- spring-cql-RELEASE.jar
- spring-data-cassandra-RELEASE.jar

- spring-data-commons-RELEASE.jar
- spring-expression-RELEASE.jar
- spring-tx-RELEASE.jar

Create a project in eclipse or any IDE and add the JAR files downloaded above. Now that the project is setup, we start with the coding phase :

We create a PoJo which maps the Column family and is the basic unit that is to be persisted in the Cassandra Database.

Person.java

```
package com.jcg.examples.entity;

import org.springframework.data.cassandra.mapping.PrimaryKey;
import org.springframework.data.cassandra.mapping.Table;

@Table("Person")
public class Person
{
                @PrimaryKey
                private Integer pId;

                private String name;

                public Integer getpId()
    {
                return pId;
    }

                public void setpId(Integer pId)
    {
                this.pId = pId;
    }

                public String getName()
    {
                return name;
    }

                public void setName(String name)
    {
                this.name = name;
    }

                @Override
    public String toString()
    {
                    return "Person [pId=" + pId + ", name=" + name + "]";
    }

                @Override
    public int hashCode()
    {
                    final int prime = 31;
                    int result = 1;
                    result = prime * result + ((name == null) ? 0 : name.hashCode());
                    result = prime * result + ((pId == null) ? 0 : pId.hashCode());
                    return result;
    }

                @Override
```

```
    public boolean equals(Object obj)
    {
                    if (this == obj)
                                    return true;
                    if (obj == null)
                                    return false;
                    if (getClass() != obj.getClass())
                                    return false;
                    Person other = (Person) obj;
                    if (name == null)
                    {
                                    if (other.name != null)
                                                    return false;
                    }
                    else if (!name.equals(other.name))
                                    return false;
                    if (pId == null)
                    {
                                    if (other.pId != null)
                                                    return false;
                    }
                    else if (!pId.equals(other.pId))
                                    return false;
                    return true;
    }

}
```

We annotate the class with `@Table` annotation to mark it as PoJo which is being mapped and the `column-family` name to which it should correspond in the Key-Space. `@PrimaryKey` simply marks the property as the Primary key.

Now that the PoJo is ready we need to create the DAO layer. The `PersonRepo` interface does the job of this.

PersonRepo.java

```
package com.jcg.examples.repo;

import org.springframework.data.repository.CrudRepository;

import com.jcg.examples.entity.Person;

public interface PersonRepo extends CrudRepository<Person, String>
{

                @Query("Select * from person where pid=?0")
                public Person fetchByPId(int pid);

}
```

The Spring Data provides a number of inbuilt method for manipulating the Data. We need not write the queries for basic data manipulation and reading. It is achieved by extending the `CrudRepository` and declaring the proper Generics as per the PoJo, which in our case is the `Person, String`. In case the Developer is not satisfied with the existing method, he can create his own method by specifying the Query using the `@Query` annotation.

The Spring IoC Container creates an instance of this `Repository` and makes it available to be used as a `Bean`.

The last and the most important part is to configure the `Spring Container` using the `spring-config.xml` :

Spring-config.xml

```
<beans:beans xmlns:beans="https://www.springframework.org/schema/beans"
        xmlns:cassandra="https://www.springframework.org/schema/data/cassandra"
        xmlns:context="https://www.springframework.org/schema/context"
```

```
        xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="https://www.springframework.org/schema/cql https://www. ↩
            springframework.org/schema/cql/spring-cql-1.0.xsd
        https://www.springframework.org/schema/beans https://www.springframework.org/schema ↩
            /beans/spring-beans.xsd
        https://www.springframework.org/schema/data/cassandra https://www.springframework. ↩
            org/schema/data/cassandra/spring-cassandra.xsd
        https://www.springframework.org/schema/context https://www.springframework.org/ ↩
            schema/context/spring-context.xsd">


        <cassandra:cluster id="cassandraCluster"
                contact-points="127.0.0.1" port="9042" />

        <cassandra:converter />

        <cassandra:session id="cassandraSession" cluster-ref="cassandraCluster"
                keyspace-name="jcg" />

        <cassandra:template id="cqlTemplate" />


        <cassandra:repositories base-package="com.jcg.examples.repo" />

        <cassandra:mapping entity-base-packages="com.jcg.examples.entity" />


</beans:beans>
```

- **Line 11 :** Configuring the Cassandra cluster. The default port in 9042.
- **Line 14 :** Cassandra Session can be colloquially referred to as a Sort of connection pool to connect to the cluster. We configure the cassandra session for the `key-space "jcg"`.
- **Line 17 :** Cassandra Template can be used to execute queries. But in this example we are creating only because it is dependency to create the Cassandra Repositories for the PoJos.
- **Line 20 :** Scan the packages for initializing Cassandra Repositories.
- **Line 22 :** Declare Mapping for the PoJos.

Now that all is set, let's run the application and test out the code! Here's Application class that loads the XML file to instantiate the Spring Container and execute a few queries.

Application.java

```
package com.jcg.examples.test;


import java.util.List;

import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.core.io.ClassPathResource;

import com.jcg.examples.entity.Person;
import com.jcg.examples.repo.PersonRepo;


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

```
                {
                        ClassPathXmlApplicationContext context = new  ↩
                            ClassPathXmlApplicationContext(new ClassPathResource("resources/ ↩
                            spring-config.xml").getPath());
                        PersonRepo personRepo = context.getBean(PersonRepo.class);
                        Person personAchilles = new Person();
                        personAchilles.setpId(1);
                        personAchilles.setName("Achilles");
                        personRepo.save(personAchilles);
                        Person personHektor = new Person();
                        personHektor.setpId(2);
                        personHektor.setName("Hektor");
                        personRepo.save(personHektor);

                        Iterable<Person> personList = personRepo.findAll();
                        System.out.println("Person List : ");
                        for (Person person : personList)
            {
                        System.out.println(person);
            }

            System.out.println("Person with Id 1 is "+personRepo.fetchByPId(1));

                context.close();

                }
}
```

In the Application class we created two instances of Person class and persisted them to the Cassandra Database. We then fetch all the records in the Person Column family and print them on the screen. Then we executed a query via the `personRepo` object to fetch the instance by specifying the Primary Key.

Here's the sample output of the program :

```
Aug 02, 2015 2:56:27 AM org.springframework.context.support.ClassPathXmlApplicationContext  ↩
    prepareRefresh
INFO: Refreshing org.springframework.context.support. ↩
    ClassPathXmlApplicationContext@78221c75: startup date [Sun Aug 02 02:56:27 IST 2015];  ↩
    root of context hierarchy
Aug 02, 2015 2:56:27 AM org.springframework.beans.factory.xml.XmlBeanDefinitionReader  ↩
    loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [resources/spring-config.xml]
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See https://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Person List :
Person [pId=1, name=Achilles]
Person [pId=2, name=Hektor]
Person with Id 1 is Person [pId=1, name=Achilles]
Aug 02, 2015 2:56:28 AM org.springframework.context.support.ClassPathXmlApplicationContext  ↩
    doClose
INFO: Closing org.springframework.context.support.ClassPathXmlApplicationContext@78221c75:  ↩
    startup date [Sun Aug 02 02:56:27 IST 2015]; root of context hierarchy
```

3.2 Download the Source Code

Here we demonstrated how to configure and manage a Cassandra Data Repository using Spring Data.

Download You can download the full source code of this example here: **CassandraSpringData.zip**

Chapter 4

Redis Example

In the past few examples, we have been integrating Spring Data with the NoSql Databases. In this example, we shall integrate Spring Data with Redis, a key-value based NoSQL Database.

Spring Data offers a level of abstraction over the traditional way of executing query by exposing a `Repository`. As such, the user need not write queries and call specific methods, depending upon the underlying Database.

Redis employs a key-store Data Structure to store data. It can be used to store complex data structure like `List`, `Set`, `Hashes` etc, which is why it is also referred to as Data-Structure Server. Like Gemfire, Redis too uses in-memory `datasets` for quicker access.

4.1 Implementation

Redis can be downloaded from here for Linux systems. We are using `Redis 3.0.3` for this demonstration. Once the Redis Server is up and running we can start connecting to it through Spring Data.

Then, we need to have following JAR Files to connect to Redis Server:

- commons-logging.jar
- commons-pool.jar
- jackson-core-asl.jar
- jackson-mapper.jar
- jedis.jar
- spring-asm.jar
- spring-beans.jar
- spring-context.jar
- spring-core.jar
- spring-data-redis.jar
- spring-expression.jar
- spring-tx.jar

Create a project in eclipse or any IDE and add the JAR files downloaded above. Now that the project is setup, we start with the coding phase :

First, we create an Entity that is to be persisted in the Redis Database.

Person.java

```
package com.jcg.examples.bean;

import java.io.Serializable;

public class Person implements Serializable
{

        private static final long serialVersionUID = -8243145429438016231L;

        public enum Gender{Male, Female}

        private String id;

        private String name;

        private Gender gender;

        private int age;

        public String getId()
        {
                        return id;
        }

        public void setId(String id)
        {
                        this.id = id;
        }

        public String getName()
        {
                        return name;
        }

        public void setName(String name)
        {
                        this.name = name;
        }

        public Gender getGender()
        {
                        return gender;
        }

        public void setGender(Gender gender)
        {
                        this.gender = gender;
        }

        public int getAge()
        {
                        return age;
        }

        public void setAge(int age)
        {
                        this.age = age;
        }

        @Override
        public int hashCode()
```

```
        {
                        final int prime = 31;
                        int result = 1;
                        result = prime * result + age;
                        result = prime * result + ((gender == null) ? 0 : gender.hashCode() ↩
                            );
                        result = prime * result + ((id == null) ? 0 : id.hashCode());
                        result = prime * result + ((name == null) ? 0 : name.hashCode());
                        return result;
        }

        @Override
        public boolean equals(Object obj)
        {
                        if (this == obj)
                                        return true;
                        if (obj == null)
                                        return false;
                        if (getClass() != obj.getClass())
                                        return false;
                        Person other = (Person) obj;
                        if (age != other.age)
                                        return false;
                        if (gender == null)
                        {
                                        if (other.gender != null)
                                                        return false;
                        }
                        else if (!gender.equals(other.gender))
                                        return false;
                        if (id == null)
                        {
                                        if (other.id != null)
                                                        return false;
                        }
                        else if (!id.equals(other.id))
                                        return false;
                        if (name == null)
                        {
                                        if (other.name != null)
                                                        return false;
                        }
                        else if (!name.equals(other.name))
                                        return false;
                        return true;
        }

        @Override
        public String toString()
        {
                        return "Person [id=" + id + ", name=" + name + ", gender=" + gender ↩
                             + ", age=" + age + "]";
        }

}
```

One difference here from the previous confgiuration of PoJos the reader will find is that there is no configuration information provided in the Entity. The Data-base simply serializes and stores the PoJo against the Key passed. That is why it is important to implement the `Serializable` interface. Not implementing `Serializable` interface leads to silly Serialization Exceptions at the time of persisting to the Database.

Next, we configure the repository which will help us in persisting the PoJo to the Redis Server:

PersonRepo.java

```
package com.jcg.examples.repo;

import java.util.Map;

import com.jcg.examples.bean.Person;

public interface PersonRepo
{
        public void save(Person person);

        public Person find(String id);

        public Map<Object, Object> findAll();

        public void delete(String id);
}
```

Next, we implement `PersonRepo` in the `PersonRepoImpl` class:

PersonRepoImpl.java

```
package com.jcg.examples.repo.impl;

import java.util.Map;

import org.springframework.data.redis.core.RedisTemplate;

import com.jcg.examples.bean.Person;
import com.jcg.examples.repo.PersonRepo;

public class PersonRepoImpl implements  PersonRepo
{

        private RedisTemplate<String, Person> redisTemplate;

        private static String PERSON_KEY = "Person";

        public RedisTemplate<String, Person> getRedisTemplate()
        {
                        return redisTemplate;
        }

        public void setRedisTemplate(RedisTemplate<String, Person> redisTemplate)
        {
                this.redisTemplate = redisTemplate;
        }

        @Override
        public void save(Person person)
        {
                this.redisTemplate.opsForHash().put(PERSON_KEY, person.getId(), person);
        }

        @Override
        public Person find(String id)
        {
                return (Person)this.redisTemplate.opsForHash().get(PERSON_KEY, id);
        }

        @Override
        public Map<Object,Object> findAll()
```

```
        {
                return this.redisTemplate.opsForHash().entries(PERSON_KEY);
        }

        @Override
        public void delete(String id)
        {
                this.redisTemplate.opsForHash().delete(PERSON_KEY,id);
        }

}
```

`PersonRepoImpl` uses the `RedisTemplate` to communicate with the Redis Server. Since we are using Hash based Operations, we are using the `Redistemplate#opsForHash()`. The method returns an instance of `HashOperations` class. We use the methods in this class to store retrieve the `Keys`.

Next is `XML` configuration.

spring-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="https://www.springframework.org/schema/beans"
        xmlns:context="https://www.springframework.org/schema/context"
        xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
        xmlns:tx="https://www.springframework.org/schema/tx",
        xmlns:p="https://www.springframework.org/schema/p"
        xsi:schemaLocation="https://www.springframework.org/schema/beans https://www. ↩
            springframework.org/schema/beans/spring-beans.xsd
                https://www.springframework.org/schema/context https://www.springframework. ↩
                    org/schema/context/spring-context.xsd
                https://www.springframework.org/schema/tx https://www.springframework.org/ ↩
                    schema/tx/spring-tx.xsd">

        <!-- Redis Connection Factory -->
        <bean id="jedisConnFactory" class="org.springframework.data.redis.connection.jedis. ↩
            JedisConnectionFactory"
                        p:use-pool="true" />

        <!-- Redis Template Configuration-->
        <bean id="redisTemplate" class="org.springframework.data.redis.core.RedisTemplate"
                        p:connection-factory-ref="jedisConnFactory" />

        <bean id="personRepo" class="com.jcg.examples.repo.impl.PersonRepoImpl">
                <property name="redisTemplate" ref="redisTemplate" />
        </bean>

</beans>
```

The `RedisTemplate` injected into the `PersonRepoImpl` class by the Spring `BeanFactory`. The `RedisTemplate` requires the `JedisConnectionFactory` instance from the `Jedis` JAR. Next we inject the `RedisTemplate` instance into the `PersonRepoImpl` bean as a reference. We can also use `@Autowired` to configure the same and add the component-scan directive in the XML.

Now that all is set, let's run the application and test out the code! Here's Application class that loads the XML file to instantiate the Spring Container and run `CRUD` commands on the server.

Application.java

```
package com.jcg.examples.test;


import java.util.Map;
```

```
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.core.io.ClassPathResource;

import com.jcg.examples.bean.Person;
import com.jcg.examples.bean.Person.Gender;
import com.jcg.examples.repo.PersonRepo;


public class Application
{
        public static void main(String[] args)
        {
                ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext ↩
                    (new ClassPathResource("resources/spring-config.xml").getPath());
                PersonRepo personRepo = (PersonRepo)context.getBean("personRepo");

                Person person = new Person();
                person.setId("1");
                person.setAge(55);
                person.setGender(Gender.Female);
                person.setName("Oracle");

                personRepo.save(person);

                Person person2 = new Person();
                person2.setId("2");
                person2.setAge(60);
                person2.setGender(Gender.Male);
                person2.setName("TheArchitect");

                personRepo.save(person2);

                Person person3 = new Person();
                person3.setId("3");
                person3.setAge(25);
                person3.setGender(Gender.Male);
                person3.setName("TheOne");

                personRepo.save(person3);

                System.out.println("Finding the One : "+personRepo.find("3"));

                Map <Object,Object> personMatrixMap = personRepo.findAll();

                System.out.println("Currently in the Redis Matrix");

                System.out.println(personMatrixMap);

                System.out.println("Deleting The Architect ");

                personRepo.delete("2");

                personMatrixMap = personRepo.findAll();

                System.out.println("Remnants .. : ");

                System.out.println(personMatrixMap);

                context.close();

        }
```

```
}
```

In `Application` class, we are creating an instances of Person Class and save them into the Redis database. We can then retrieve and delete them.

Here's the sample output of the program :

```
Aug 09, 2015 4:02:57 PM org.springframework.context.support.AbstractApplicationContext ↩
    prepareRefresh
INFO: Refreshing org.springframework.context.support. ↩
    ClassPathXmlApplicationContext@42b1b290: startup date [Sun Aug 09 16:02:57 IST 2015]; ↩
    root of context hierarchy
Aug 09, 2015 4:02:57 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader ↩
    loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [resources/spring-config.xml]
Aug 09, 2015 4:02:57 PM org.springframework.beans.factory.support. ↩
    DefaultListableBeanFactory preInstantiateSingletons
INFO: Pre-instantiating singletons in org.springframework.beans.factory.support. ↩
    DefaultListableBeanFactory@322558e: defining beans [jedisConnFactory,redisTemplate, ↩
    personRepo]; root of factory hierarchy
Finding the One : Person [id=3, name=TheOne, gender=Male, age=25]
Currently in the Redis Matrix
{1=Person [id=1, name=Oracle, gender=Female, age=55], 3=Person [id=3, name=TheOne, gender= ↩
    Male, age=25], 2=Person [id=2, name=TheArchitect, gender=Male, age=60]}
Deleting The Architect
Remnants .. :
{1=Person [id=1, name=Oracle, gender=Female, age=55], 3=Person [id=3, name=TheOne, gender= ↩
    Male, age=25]}
Aug 09, 2015 4:02:58 PM org.springframework.context.support.AbstractApplicationContext ↩
    doClose
INFO: Closing org.springframework.context.support.ClassPathXmlApplicationContext@42b1b290: ↩
    startup date [Sun Aug 09 16:02:57 IST 2015]; root of context hierarchy
Aug 09, 2015 4:02:58 PM org.springframework.beans.factory.support. ↩
    DefaultSingletonBeanRegistry destroySingletons
INFO: Destroying singletons in org.springframework.beans.factory.support. ↩
    DefaultListableBeanFactory@322558e: defining beans [jedisConnFactory,redisTemplate, ↩
    personRepo]; root of factory hierarchy
```

4.2 Download the Source Code

Here we demonstrated how to configure and manage a Redis Data Repository using Spring Data.

Download You can download the full source code of this example here: **SpringDataRedisExample.zip**

Chapter 5

MongoDB REST Example

In this tutorial, I will show you how to develop a RESTful web application by using Spring and MongoDB without implementing boring uniform controllers.

5.1 Introduction

When it comes to RESTful application, we need an api that mainly handles CRUD operations. Let me give you example to clearify what I mean by sentence above. Let say that, you have a REST endpoint `/products` that allows you to perform product related operations like **Product Create**, **Product Update**, **Product View**, **Product Delete,** or **Product Search**. The minimum code requirements for designing such an endpoint is nearly same for every resources like below:

- Design your domain model (Product)
- Implement Repository
- Implement Controller

Quick Tip You will see the term `resource` in this article frequently. It is one of the main components of the RESTful design and you can refer here to learn more about what a resource is.

By using above components, you can handle requests with **controllers**, validate and convert request to **Entity**, and finaly perform CRUD operation by using **Service** that use **DAO** classes for each entity. In this situation, you need to code 4 endpoint for each resource(Product is a resource here). Let say that you have 10 resources in your project like User, Comment, History, etc.. You will have 10 x 4 endpoint needs to be implemented. Those endpoints are mainly same for all resources. What if we do not need to implement those CRUD enpoints for each resources? We will use Spring Data Rest project for lead to implement RESTful services by implementing only Repository and Domain Model.

5.2 Requirements

In this project, we will need following requirements;

- Maven 3.x
- Spring Boot 1.2.5
- Valid MongoDB database

I have prefered Spring Boot for fast web development with Spring. If you have never heard about Spring Boot, you can have a look at here. I have specified the version according to current time, but you don't need to do anything with versions, I will provide pom.xml for dependencies. You can see example `pom.xml` below;

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↩
    XMLSchema-instance"
         xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/ ↩
             xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>huseyinbabal.net</groupId>
    <artifactId>spring-mongodb-data-rest</artifactId>
    <version>0.1.0</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.2.5.RELEASE</version>
    </parent>

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

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-rest</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-mongodb</artifactId>
        </dependency>
    </dependencies>

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

</project>
```

In above dependecy xml, we have only 2 dependecies for REST and MongoDB related operations. spring-boot-starter-parent is for minimum dependency for Spring Boot to allow us to develop web projects. spring-boot-maven-plugin is for running Spring Boot projects by using maven. Let see all of them in action with a demo application.

5.3 Domain Model

We will develop a RESTful service for Product related operations. Let's create a domain object for Product.

Product.java

```
package main.java.springmongodbdatarest;

import org.springframework.data.annotation.Id;

public class Product {
```

```
        @Id
        private String id;

        private String name;
        private String title;
        private String description;
        private String imageUrl;
        private double price;
        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }
        public String getTitle() {
                return title;
        }
        public void setTitle(String title) {
                this.title = title;
        }
        public String getDescription() {
                return description;
        }
        public void setDescription(String description) {
                this.description = description;
        }
        public String getImageUrl() {
                return imageUrl;
        }
        public void setImageUrl(String imageUrl) {
                this.imageUrl = imageUrl;
        }
        public double getPrice() {
                return price;
        }
        public void setPrice(double price) {
                this.price = price;
        }
}
```

This object will be used to persist product data to MongoDB and also when we read a product data from MongoDB, it will be authomatically casted to this object. We will not do anything about persistence, Spring will handle all of the MongoDB operations. And `@Id` annotation in product model is for generating automatic id for MongoDB. Let's continue with Repository.

5.4 Repository

We need a repository interface to make MongoDB related operations by using Product model. We can use `ProductRepository` for our case. You can see an example repository interface below.

ProductRepository.java

```
package main.java.springmongodbdatarest;

import org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

@RepositoryRestResource(collectionResourceRel = "products", path = "products")
public interface ProductRepository extends MongoRepository<Product, String> {

}
```

As you can see, this simple interface will handle all of your db related operations, and it will also implement REST endpoints with some annotations. But, how?

We have spring-data-rest dependency and this will make an implementation of above interface at runtime. By using `@RepositoryRestResource`, it will create an endpoint `/products`. `collectionResourceRel` is for MongoDB collection name, that means, when you create a product, it will be persisted to MongoDB database inside **products** collection as we stated with `collectionResourceRel`. `path` is for rest endpoint name for the product model. ProductRepository interface extends MongoRepository to inhert some DB related operations, so Spring will automatically use `save()`, `find()`, etc.. kind methods for your model. Lastly, we have provided Product and String to MongoRepository and this means that this repository will make db operations through **Product** model, and data type of primary id field of this model is **String**

5.5 MongoDB

In order to run your project successfully, you need valid MongoDB database. Spring Boot uses `localhost:27017/test` by default. If you want to override database properties, create `application.properties` in `resources` folder and put following content in it.

MongoDB URL in property file

```
spring.data.mongodb.uri=mongodb://localhost:27017/test
```

You can also see example in sample project.

5.6 Boot Up

In order to make this REST service up, we will use following bootstrap class;

Application.java

```
package main.java.springmongodbdatarest;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {

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

We are saying that, this is an SpringBootApplication and go get the dependencies and run required components. You can put all 3 classes inside same package.

In order to run project, go to project folder and run `mvn spring-boot:run`. This will run project and you will be able to access REST service endpoints on `8080` port.

5.7 Let See It In Action!

When you make a request to application root(In my case: `https://localhost:8080`) you will see the available endpoints for all resources like below

Sample REST Response

```
{
    "_links": {
        "products": {
            "href": "https://localhost:8080/products{?page,size,sort}",
            "templated": true
        },
        "profile": {
            "href": "https://localhost:8080/alps"
        }
    }
}
```

As you can see, there is only one resource `products` and you can see list of products by going url in the `href`. Note that, you can manipulate product results by using `page`, `size`, and `sort`. Let's add one product to the database.

Product model has 5 fields, and we will post that details in the request payload. Before proceeding data creation, HTTP methods has following roles in RESTful services.

- **POST:** Create
- **PUT:** Total update
- **PATCH:** Partial update
- **DELETE:** Delete
- **GET:** View

So, If we want to create a record, we can make a `POST` request to `https://localhost:8080/products` with product payload. You can see example below:

Request Payload for Product Creation

```
{
    "name": "Glass",
    "title": "Beatiful Glass",
    "descriptions": "This is one of the greatest glass",
    "imageUrl": "https://img.com",
    "price": "4$"
}
```

When you make this request, you will get a response with status code 201(Created). We have a product data now, and you can list products now by going url `https://localhost:8080/products`. The response will be like below

Product Detail Response

```
{
  "_links" : {
    "self" : {
      "href" : "https://localhost:8080/products{?page,size,sort}",
      "templated" : true
    }
  },
  "_embedded" : {
    "products" : [ {
      "name" : "Glass",
      "title" : "Beatiful Glass",
      "description" : null,
      "imageUrl" : "https://img.com",
      "price" : 4.0,
      "_links" : {
        "self" : {
```

```
          "href" : "https://localhost:8080/products/55cc79774568615d83f718be"
        }
      }
    } ]
  },
  "page" : {
    "size" : 20,
    "totalElements" : 1,
    "totalPages" : 1,
    "number" : 0
  }
}
```

As you can see, there is a url for each resource to make resource specific operations. For example, you can delete resource by performing DELETE request to `https://localhost:8080/products/55cc79774568615d83f718be`, you will get 204(No content). The record deleted and returned no content.

5.8 Summary

RESTful API needs to be designed very carefully. Spring Data Rest project handles RESTful API concepts very good. In this article, I have tried to explain you how to use Spring Data Rest within your projects. Hope this is helpful for you.

5.9 Download the Source Code

Here we demonstrated how to configure and manage a Redis Data Repository using Spring Data.

Download You can download the full source code of this example here: **SpringMongoDBDataRestExample**

Chapter 6

Solr Example

In this example, we will demonstrate how to integrate Spring data with Apache Solr.

`Solr` is a search engine built on top of Apache Lucene library. It can be communicated with a REST like HTTP API because of which it can be easily be consumed as a REST-ful web-service irrespective of the underlying programming language used in the application which is calling the Solr Server. However, for the sake of this example we will be using Java as the programming language and Spring Data as the framework.

The Spring Data Solr is the module of Spring Data that provides support for Solr. As with the other examples in this series, this module supports both for derived queries(based on the method name) and the annotated query.

6.1 Implementation

Download the Apache Solr from here. The version at the time of publishing this blog was `5.2.1`. Unzip the downloaded file, change directory to location where Solr `bin` is unzipped and run the following commands :

```
solr start -p 8983

solr create -c jcg
```

The first command start the solr server, while the second command creates a core, an index. Verify if the server is up by hitting the URL : https://localhost:8983/solr. In version `5.2` of Solr, the `solrconfig.xml` uses the `ManagedIndexSchemaFactory` as the `schemaFactory`. However, we will be using the `ClassicIndexSchemaFactory` for this example. Make the following changes to do so:

- Go to $(Solr_Home)/server/solr/$(core_name)/conf and rename the managed-schema to `schema.xml`.
- Comment out the existing `schameFactory` tag and add the following line :

```
<schemaFactory class="ClassicIndexSchemaFactory"/>
```

- Reload the core at the URL mentioned above in the `Cores` tab

Now that the core is setup, we need to add our fields to the `schema.xml`.

```
<field name="id" type="string" indexed="true" stored="true" required="true" multiValued=" ↩
    false" />
        <field name="title" type="string" indexed="true" stored="true" required="false"  ↩
            multiValued="false" />
        <field name="description" type="string" indexed="true" stored="true" required=" ↩
            false" multiValued="false" />
```

That's all on the Solr side. The core is now up and ready to use. Let's start coding on the Application side.

We need to have following JAR Files to connect to Solr Server:

- aopalliance-1.0
- commons-io-1.3.2
- commons-lang3-3.4
- commons-logging-1.1.3
- httpclient-4.3.6
- httpcore-4.4.1
- httpmime-4.3.6
- noggit-0.7
- slf4j-api-1.7.5
- solr-solrj-4.10.3
- spring-aop-4.1.4.RELEASE
- spring-beans-4.1.4.RELEASE
- spring-core-4.1.4.RELEASE
- spring-context-4.1.4.RELEASE
- spring-data-commons-1.10.2.RELEASE
- spring-data-solr-1.4.2.RELEASE
- spring-expression-4.2.0.RELEASE
- spring-tx-3.1.1.RELEASE

Create a project in eclipse or any IDE and add the JAR files downloaded above. Now that the project is setup, we start with the coding phase :

First, we create an Entity that is to be persisted in the `Solr` for searching later.

Book.java

```
package com.jcg.examples.entity;


import java.io.Serializable;

import org.apache.solr.client.solrj.beans.Field;
import org.springframework.data.annotation.Id;

public class Book implements Serializable
{

                private static final long serialVersionUID = -8243145429438016231L;

                @Id
                @Field
                private String id;

                @Field
                private String title;
```

```
@Field
private String description;

public String getId() {
        return id;
}

public void setId(String id) {
        this.id = id;
}

public String getTitle()
{
                return title;
}

public void setTitle(String title)
{
                this.title = title;
}

public String getDescription()
{
                return description;
}

public void setDescription(String description)
{
                this.description = description;
}



@Override
public int hashCode()
{
                final int prime = 31;
                int result = 1;
                result = prime * result + ((description == null) ? 0 :  ↩
                    description.hashCode());
                result = prime * result + ((id == null) ? 0 : id.hashCode() ↩
                    );
                result = prime * result + ((title == null) ? 0 : title. ↩
                    hashCode());
                return result;
}

@Override
public boolean equals(Object obj)
{
                if (this == obj)
                                return true;
                if (obj == null)
                                return false;
                if (getClass() != obj.getClass())
                                return false;
                Book other = (Book) obj;
                if (description == null)
                {
                                if (other.description != null)
                                                return false;
```

```
                                }
                                else if (!description.equals(other.description))
                                                return false;
                                if (id == null)
                                {
                                                if (other.id != null)
                                                                return false;
                                }
                                else if (!id.equals(other.id))
                                                return false;
                                if (title == null)
                                {
                                                if (other.title != null)
                                                                return false;
                                }
                                else if (!title.equals(other.title))
                                                return false;
                                return true;
                }

                @Override
                public String toString()
                {
                                return "Book [id=" + id + ", title=" + title + ",  ↩
                                    description=" + description + "]";
                }

}
```

The `id` field is the unique/Primary field defined in the `schema.xml` and the same is annotated with `@Id`. The `@Field` is used to mark the other fields in the schema. In case the name of the field is different in the `schema.xml`, we pass the name of the field in the value attribute of the `@Field` annotation.

Next, we configure the repository which will help us in persisting the `Book` Entity to the Solr Server:

BookRepo.java

```
package com.jcg.examples.repo;

import org.springframework.data.repository.CrudRepository;
import org.springframework.data.solr.repository.Query;

import com.jcg.examples.entity.Book;

public interface BookRepo extends CrudRepository<Book, Long>
{

                @Query("title:?0")
                public Book findByBookTitle(String name);

}
```

The Spring Data provides a number of inbuilt method for manipulating the Data. We need not write the queries for basic data manipulation and reading. It is achieved by extending the CrudRepository and declaring the proper Generics as per the Entity, which in our case is the <Book, Long>.

In case the Developer is not satisfied with the existing method, he can create his own method by specifying the Query using the `@Query` annotation. In the `BookRepo` class, the `findByBookTitle` method searches for the argument passed in the `title` field of the `jcg` Solr Core.

The Spring IoC Container creates an instance of this Repository and makes it available to be used as a Factory Bean.

The last and the most important part is to configure the Spring Container using the spring-config.xml:

spring-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="https://www.springframework.org/schema/beans"
       xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="https://www.springframework.org/schema/context"
       xmlns:solr="https://www.springframework.org/schema/data/solr"
       xsi:schemaLocation="https://www.springframework.org/schema/beans https://www. ↩
           springframework.org/schema/beans/spring-beans.xsd
       https://www.springframework.org/schema/context https://www.springframework.org/ ↩
           schema/context/spring-context-3.2.xsd https://www.springframework.org/schema/data ↩
           /solr https://www.springframework.org/schema/data/solr/spring-solr.xsd">


    <solr:repositories base-package="com.jcg.examples.repo"/>

        <!-- Define HTTP Solr server -->
        <solr:solr-server id="solrServer"  url="https://localhost:8983/solr/jcg"/>

        <!-- Define Solr template -->
        <bean id="solrTemplate"  class="org.springframework.data.solr.core.SolrTemplate">
            <constructor-arg index="0" ref="solrServer"/>
        </bean>
</beans>
```

- **Line 10:** Scan the packages for initializing Cassandra Repositories.
- **Line 13:** Provide the port ,host and core for instance of Solr Server we wish to connect with the Application.
- **Line 16:** reate an instance of SolrTemplate which will communicate with the Solr Server to execute the queries.

Now that all is set, let's run the application and test out the code! Here's Application class that loads the XML file to instantiate the Spring Container and execute a few queries.

Application.java

```
package com.jcg.examples.test;


import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.core.io.ClassPathResource;

import com.jcg.examples.entity.Book;
import com.jcg.examples.repo.BookRepo;


public class Application
{
                public static void main(String[] args)
                {
                                ClassPathXmlApplicationContext context = new  ↩
                                    ClassPathXmlApplicationContext(new ClassPathResource(" ↩
                                    resources/spring-config.xml").getPath());
                                BookRepo bookRepo = context.getBean(BookRepo.class);

                                Book rrs = new Book();
                                rrs.setId("1");
                                rrs.setTitle("Red Storm Rising");
                                rrs.setDescription("World War III");
                                bookRepo.save(rrs);
```

```
                                Book hobbit = new Book();
                                hobbit.setId("3");
                                hobbit.setTitle("Hobbit");
                                hobbit.setDescription("Prelude to LOTR");
                                bookRepo.save(hobbit);


                                System.out.println(bookRepo.findOne(1l));
                                System.out.println(bookRepo.findByBookTitle("Hobbit"));

                                context.close();

                }
}
```

In the `Application` class we created two instances of Book class and persisted them to the Solr Server. We then fetch the record from the core by the unique key. Next we fetch the data by executing the explicit query in the `BookRepo` class.

Here's the sample output of the program:

```
Aug 17, 2015 12:56:56 AM org.springframework.context.support.ClassPathXmlApplicationContext ↩
     prepareRefresh
INFO: Refreshing org.springframework.context.support. ↩
    ClassPathXmlApplicationContext@28fa1b85: startup date [Mon Aug 17 00:56:56 IST 2015]; ↩
    root of context hierarchy
Aug 17, 2015 12:56:56 AM org.springframework.beans.factory.xml.XmlBeanDefinitionReader ↩
    loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [resources/spring-config.xml]
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See https://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Book [id=1, title=Red Storm Rising, description=World War III]
Book [id=3, title=Hobbit, description=Prelude to LOTR]
Aug 17, 2015 12:56:57 AM org.springframework.context.support.ClassPathXmlApplicationContext ↩
     doClose
INFO: Closing org.springframework.context.support.ClassPathXmlApplicationContext@28fa1b85: ↩
    startup date [Mon Aug 17 00:56:56 IST 2015]; root of context hierarchy
```

6.2 Download the Source Code

Here we demonstrated how to configure and manage a Apache Solr Search Engine using Spring Data.

Download You can download the full source code of this example here: **SpringDataSolrExample.zip**

Chapter 7

MongoDB Example

In this example, we will demonstrate how to connect Spring Data with MongoDb. MongoDb is also a document based NoSql Database like Solr, which we demonstrated past week.

Spring Data MongoDb is the module of Spring Data that provides support for MongoDb. As with the other modules demonstrated in this series, this module too provides supports both for derived queries(based on the method name) and the annotated query.

Let's get started with the setup:

7.1 How to configure and manage a MongoDb using Spring Data

Install Mongodb depending upon your system from here.

Point to the `bin` of the installed MongoDb, which is `Program Files` in Windows. Then run the following command:

```
mongo --dbpath C:\MongoDb\Data
```

This command starts the MongoDb Server with the repository location at the path specified in the command above. However, the folders should already be present otherwise the server will throw `Invalid Directory` error. The default port for MongoDb is `271017`.

Now that the `Mongo` server is up and running, we will setup the application environment.

Create a simple Maven Project in eclipse IDE. We are using the below `pom.xml` to manage the dependencies for MongoDb from Spring data.

pom.xml

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↩
    XMLSchema-instance" xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven ↩
    .apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>SpringDataMongoDbExample</groupId>
  <artifactId>SpringDataMongoDbExample</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <dependencies>
    <dependency>
        <groupId>org.springframework.data</groupId>
        <artifactId>spring-data-mongodb</artifactId>
        <version>1.7.2.RELEASE</version>
    </dependency>
</dependencies>

</project>
```

Eclipse will download the required JAR files and add the dependencies in the project classpath. Now that the project is setup and dependencies imported, we can begin writing the actual code.

We start by creating the entities that will be persisted to the Mongo Database.

Person.java

```
package com.jcg.examples.entity;

import java.util.ArrayList;
import java.util.List;

import org.springframework.data.annotation.Id;
import org.springframework.data.annotation.PersistenceConstructor;
import org.springframework.data.mongodb.core.mapping.DBRef;
import org.springframework.data.mongodb.core.mapping.Document;

@Document(collection="person")
public class Person
{

		@Id
		private Long personId;

		private String name;

		private int age;

		@DBRef(db="address")
		private List<Address> addresses = new ArrayList<>();

		public Person()
		{}


		@PersistenceConstructor
		public Person(Long personId, String name, int age)
		{
				super();
				this.personId = personId;
				this.name = name;
				this.age = age;
		}

		public Long getPersonId()
		{
				return personId;
		}

		public void setPersonId(Long personId)
		{
				this.personId = personId;
		}

		public String getName()
		{
				return name;
		}

		public void setName(String name)
		{
				this.name = name;
		}
```

```
		public int getAge()
		{
			return age;
		}

		public void setAge(int age)
		{
			this.age = age;
		}

		public List<Address> getAddresses()
		{
			return addresses;
		}


		public void setAddresses(List<Address> addresses)
		{
			this.addresses = addresses;
		}


		@Override
		public String toString()
		{
			return "Person [personId=" + personId + ", name=" + name +  ↩
			    ", age=" + age + ", addresses=" + addresses + "]";
		}

}
```

`@Document` is used to denote the collection in which the data will be persisted. If it is not mentioned, the data is saved in the collection which has the same name as the Entity Class Name.

`@Id` maps the property annotated with it to the `_id` column of the collection. In case no property is annotated with `@Id`, the property with the name id will be mapped to the `_id`. In case there is now property with that name, a column will be generated by the Mongo Driver, but the value will not available in the PoJo.

`@DBRef` is used to relate an existing entity to the current entity. However, unlike the case with Relational Databases, if we save the host entity it does not save the related entity. It has to be persisted separately.

`@PersistenceConstructor` is used to mark the constructor which is to be used for creating entities when fetching data from the Mongo Server.

Here is the linked entity `Address`.

Address.java

```
package com.jcg.examples.entity;

import org.springframework.data.annotation.Id;
import org.springframework.data.annotation.PersistenceConstructor;
import org.springframework.data.mongodb.core.mapping.Document;

@Document(collection="address")
public class Address
{

		@Id
		private long addressId;

		private String address;
```

```
        private String city;

        private String state;

        private long zipcode;

        public Address()
        {

                        System.out.println("CAlling default cons");
        }

        @PersistenceConstructor
        public Address(long addressId, String address, String city, String state,  ↩
            long zipcode)
        {
                        super();
                        this.addressId = addressId;
                        this.address = address;
                        this.city = city;
                        this.state = state;
                        this.zipcode = zipcode;
        }




        public String getAddress()
        {
                        return address;
        }

        public void setAddress(String address)
        {
                        this.address = address;
        }

        public String getCity()
        {
                        return city;
        }

        public void setCity(String city)
        {
                        this.city = city;
        }

        public String getState()
        {
                        return state;
        }

        public void setState(String state)
        {
                        this.state = state;
        }

        public long getZipcode()
        {
                        return zipcode;
        }
```

```
                public void setZipcode(long zipcode)
                {
                        this.zipcode = zipcode;
                }

                @Override
                public String toString()
                {
                        return "Address [address=" + address + ", city=" + city + " ↩
                            , state=" + state + ", zipcode=" + zipcode + "]";
                }

}
```

Now we will create a Repository for each of the entity defined above which will help us in persisting the respective Entities to the MongoDb Server.

PersonRepo.java

```
package com.jcg.examples.repo;

import org.springframework.data.mongodb.repository.Query;
import org.springframework.data.repository.CrudRepository;

import com.jcg.examples.entity.Person;

public interface PersonRepo extends CrudRepository<Person, Long>
{
                @Query("{'name' : ?0}")
                public Iterable<Person> searchByName(String personName);

}
```

Spring Data Module provides us with a number of inbuilt method for manipulating the Data. We need not write the queries for basic data manipulation and reading. It is achieved by extending the `CrudRepository` interface and declaring the proper Generics as per the Entity, which in our case is the `<Person, Long>`.

For executing custom written queries, the developer can create his own method by specifying the Query using the `@Query` annotation. In the above class, we have annotated the method, `searchByName` with the said annotation. This method returns the Person Entity by querying the name field of the `Person` Collection from the Server.

Here's the `AddressRepo`

AddressRepo.java

```
package com.jcg.examples.repo;

import org.springframework.data.repository.CrudRepository;

import com.jcg.examples.entity.Address;

public interface AddressRepo extends CrudRepository
```

The last and the most important part is to configure the Spring Container using the `spring-config.xml`:

Spring-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="https://www.springframework.org/schema/beans"
          xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
          xmlns:context="https://www.springframework.org/schema/context"
          xmlns:mongo="https://www.springframework.org/schema/data/mongo"
```

```
          xsi:schemaLocation=
          "https://www.springframework.org/schema/context
          https://www.springframework.org/schema/context/spring-context-3.0.xsd
          https://www.springframework.org/schema/data/mongo
          https://www.springframework.org/schema/data/mongo/spring-mongo-1.0.xsd
          https://www.springframework.org/schema/beans
          https://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <!-- Configure the Mongo Server -->
    <mongo:mongo id="mongo" host="localhost" port="27017"/>

    <bean id="mongoTemplate" class="org.springframework.data.mongodb.core.MongoTemplate">
    <constructor-arg ref="mongo"/>
    <constructor-arg name="databaseName" value="jcg"/>
  </bean>

    <mongo:repositories base-package="com.jcg.examples.repo"></mongo:repositories>


</beans>
```

- **Line 15:** Configure the Mongo Server by providing the server and the Port on which it is running.
- **Line 17:** The `mongoTemplate` is used as a dependency for creating the repositories which we discussed above.
- **Line 22:** Scan the packages for initializing Mongo Bean Repositories.

Now that all is set, let's run the application and test out the code! Here's `Application` class that loads the `XML` file to instantiate the Spring Container and execute a few queries.

Application.java

```
package com.jcg.examples.test;


import java.util.List;

import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.core.io.ClassPathResource;

import com.jcg.examples.entity.Address;
import com.jcg.examples.entity.Person;
import com.jcg.examples.repo.AddressRepo;
import com.jcg.examples.repo.PersonRepo;


public class Application
{
                public static void main(String[] args)
                {
                                ClassPathXmlApplicationContext context = new  ↩
                                    ClassPathXmlApplicationContext(new ClassPathResource(" ↩
                                    spring-config.xml").getPath());
                                PersonRepo personRepo = context.getBean(PersonRepo.class);
                                AddressRepo addressRepo = context.getBean(AddressRepo.class ↩
                                    );

                                Person personAchilles = new Person();
                                personAchilles.setPersonId(11);
                                personAchilles.setName("Achilles");
                                personRepo.save(personAchilles);
```

```
                                Person personHektor = new Person();
                                personHektor.setPersonId(2l);
                                personHektor.setName("Hektor");

                                Address address = new Address(1,"221b Baker Street","London ↩
                                     NW1","London",12345l);
                                List<Address> addresses = personHektor.getAddresses();
                                addresses.add(address);
                                personAchilles.setAddresses(addresses);

                                addressRepo.save(address);
                                personRepo.save(personHektor);

                                Iterable<Person> personList = personRepo.findAll();
                                System.out.println("Person List : ");
                                for (Person person : personList)
        {
                        System.out.println(person);
        }

                                System.out.println("Person Record with name Hektor  is "+ ↩
                                    personRepo.searchByName("Hektor"));

                                context.close();

                }
}
```

In the `Application` we create two instances of `Person` and persist them. However, the second instance also has a linked `Address` instance to it, which we persist separately by calling the `save` method of the `AddressRepo` class.

Then we iterate all the elements stored in the Person collection. We have also successfully searched for the Person named Hektor from the collection using custom query and the `@Query` annotation.

Here's the sample output of the program:

```
Person List :
Person [personId=1, name=Achilles, age=0, addresses=[]]
Person [personId=2, name=Hektor, age=0, addresses=[Address [address=221b Baker Street, city ↩
    =London NW1, state=London, zipcode=12345]]]
Person Record with name Hektor  is [Person [personId=2, name=Hektor, age=0, addresses=[ ↩
    Address [address=221b Baker Street, city=London NW1, state=London, zipcode=12345]]]]
```

7.2 Download the Source Code

Here we demonstrated how to configure and manage a MongoDb using Spring Data.

Download You can download the full source code of this example here: **SpringDataMongoDbExample.zip**

Chapter 8

REST Example

In the previous few examples we demonstrated how the different spring data modules are configured and their support for the `NoSql` databases. In this example, I will demonstrate how we can expose our `CRUD Repository` as a REST Web-Service over the HTTP.

We will be using MySQL as the database for this project . However, the application which we shall build here will be largely independent of the underlying Database technology used. This is because of the abstraction layer added by the Spring Data Module.

So let's get started with the project setup:

We will create a new Maven project with archetype as `maven-archetype-webapp`. Update the `pom.xml` with the below file :

pom.xml

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↩
    XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/maven- ↩
      v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>SpringDataRestExample1</groupId>
  <artifactId>SpringDataRestExample1</artifactId>
  <packaging>war</packaging>
  <version>0.0.1-SNAPSHOT</version>
  <name>SpringDataRestExample</name>
  <url>https://maven.apache.org</url>
  <dependencies>

                <dependency>
                        <groupId>org.springframework.data</groupId>
                        <artifactId>spring-data-jpa</artifactId>
                        <version>1.9.0.RELEASE</version>
                </dependency>

                <dependency>
                        <groupId>org.springframework.data</groupId>
                        <artifactId>spring-data-rest-webmvc</artifactId>
                        <version>2.3.2.RELEASE</version>
                </dependency>

                <dependency>
                        <groupId>mysql</groupId>
                        <artifactId>mysql-connector-java</artifactId>
                        <version>5.1.36</version>
                </dependency>
```

```
                <dependency>
                        <groupId>org.hibernate</groupId>
                        <artifactId>hibernate-core</artifactId>
                        <version>4.3.6.Final</version>
                </dependency>
                <dependency>
                        <groupId>org.hibernate</groupId>
                        <artifactId>hibernate-entitymanager</artifactId>
                        <version>4.3.6.Final</version>
                </dependency>

        </dependencies>

        <build>
                <plugins>
                        <plugin>
                                <groupId>org.apache.maven.plugins</groupId>
                                <artifactId>maven-eclipse-plugin</artifactId>
                                <version>3.3</version>
                                <configuration>
                                        <downloadSources>true</downloadSources>
                                        <downloadJavadocs>true</downloadJavadocs>

                                </configuration>

                        </plugin>
                           <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>
                </plugins>
                <finalName>SpringDataRestExample</finalName>
        </build>
</project>
```

This will add the required dependencies for the archives we will need for setting up this project. Let's start with the implementation now.

We start by creating the entities that will be persisted to the `MySql` server.

Person.java

```
package com.jcg.examples.entity;

import java.io.Serializable;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity(name="person")
public class Person implements Serializable
{

                private static final long serialVersionUID = -5378396373373165919L;
```

```
	@Id
	@GeneratedValue(strategy=GenerationType.IDENTITY)
	private Long pId;

	@Column
	private String personName;

	@Column
	private double personAge;

	public Long getpId()
	{
		return pId;
	}

	public void setpId(Long pId)
	{
		this.pId = pId;
	}

	public String getPersonName()
	{
		return personName;
	}

	public void setPersonName(String personName)
	{
		this.personName = personName;
	}

	public double getPersonAge()
	{
		return personAge;
	}

	public void setPersonAge(double personAge)
	{
		this.personAge = personAge;
	}

	@Override
	public String toString()
	{
		return "Person [pId=" + pId + ", personName=" + personName
			+ ", personAge=" + personAge + "]";
	}
}
```

We are using annotations to map the entity properties to the database table columns. Here is a brief description of the annotations we have used for mapping:

`@Id` is used to mark the primary key of the entity. `@Column` is used to map the properties of the entity to those of the table columns. In case, the name of the column happens to be different from the name of the property we can use the `name` attribute of the `@Column` annotation and pass the name of the table column.

Next we create a repository to persist the entity defined above to the database.

PersonRepo.java

```
package com.jcg.examples.repo;

import java.util.List;
```

```
import org.springframework.data.repository.CrudRepository;
import org.springframework.data.repository.query.Param;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;
import org.springframework.data.rest.core.annotation.RestResource;

import com.jcg.examples.entity.Person;

@RepositoryRestResource
public interface PersonRepo extends CrudRepository<Person, Long>
{
                @RestResource(path="byName")
                public List findByPersonName(@Param("name") String personName);
}
```

Spring Data Module provides us with a number of inbuilt method for manipulating the Data. We need not write the queries for basic data manipulation and reading. It is achieved by extending the `CrudRepository` interface and declaring the proper Generics as per the Entity, which in our case is the <Person, Long>.

Also, we have a created a custom a method that searches based on the `name` property of the `Person`.

Now, we have annotated the repository with the `@RepositoryRestResource` annotation. This marks the resource to be exported as a `REST` resource which is available over the `HTTP`. We may also choose to hide some methods by adding the annotation : `@RestResource(exported =false)`. The same annotation can be used to annotate an entity property so that it is not transmitted over the network.

Now we create the `spring-config.xml` and place it in the `WEB-INF` folder. This file contains the configuration of the database and other required beans. Line 13 causes the spring container to scan the repo package for the classes annotated with the `@RepositoryRestResource` to be exported.We can also use the path attribute of the `@RestResource` to modify the path of the method.

spring-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="https://www.springframework.org/schema/beans"
        xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
        xmlns:jpa="https://www.springframework.org/schema/data/jpa"
        xmlns:context="https://www.springframework.org/schema/context"
        xsi:schemaLocation="https://www.springframework.org/schema/beans
    https://www.springframework.org/schema/beans/spring-beans.xsd
    https://www.springframework.org/schema/data/jpa
    https://www.springframework.org/schema/data/jpa/spring-jpa.xsd
    https://www.springframework.org/schema/context
https://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:component-scan base-package="com.jcg.examples.repo" />

    <bean id="dataSource" class="org.springframework.jdbc.datasource. ↩
        DriverManagerDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://localhost/test"/>
        <property name="username" value="root"/>
        <property name="password" value="toor"/>
    </bean>

    <bean id="jpaVendorAdapter" class="org.springframework.orm.jpa.vendor. ↩
        HibernateJpaVendorAdapter">
        <property name="showSql" value="true"/>
        <property name="generateDdl" value="true"/>
        <property name="database" value="MYSQL"/>
    </bean>
```

```
    <bean id="entityManagerFactory" class="org.springframework.orm.jpa. ↩
        LocalContainerEntityManagerFactoryBean">
        <property name="dataSource" ref="dataSource"/>
        <property name="jpaVendorAdapter" ref="jpaVendorAdapter"/>
        <property name="packagesToScan" value="com.jcg.examples.entity"/>
    </bean>

    <bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager" ↩
        />

    <jpa:repositories base-package="com.jcg.examples.repo" />

</beans>
```

Next is the most important `web.xml`. We configure the `ContextLoaderListener` to load the `spring-config.xml` file and bind the `Applicationcontext` lifecycle to that of the `ServletContext`.This initializes the beans we have defined in the `xml`

The `RepositoryRestDispatcherServlet` is used to expose the REST Resources over the network. The non-zero value of `load-on-startup` marks the servlet to be eagerly loaded during the web-container initialization.

web.xml

```
<web-app xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
        xmlns="https://java.sun.com/xml/ns/javaee"
        xsi:schemaLocation="https://java.sun.com/xml/ns/javaee https://java.sun.com/xml/ns/ ↩
            javaee/web-app_3_0.xsd"
        id="WebApp_ID" version="3.0">

        <display-name>Archetype Created Web Application</display-name>
        <listener>
                <listener-class>org.springframework.web.context.ContextLoaderListener</ ↩
                    listener-class>
        </listener>

        <context-param>
                <param-name>contextConfigLocation</param-name>
                <param-value>/WEB-INF/spring-config.xml</param-value>
        </context-param>

        <servlet>
                <servlet-name>rest</servlet-name>
                <servlet-class>org.springframework.data.rest.webmvc. ↩
                    RepositoryRestDispatcherServlet</servlet-class>
                <load-on-startup>1</load-on-startup>
        </servlet>

        <servlet-mapping>
                <servlet-name>rest</servlet-name>
                <url-pattern>/api/*</url-pattern>
        </servlet-mapping>

</web-app>
```

Deploy the above application to the Web-Server/Application Server you have configured and the REST service is ready to be consumed.

Here is the sample output when different HTTP methods are used. For the sake of this project I have used `curl`, a command line tool, to test our REST Web-Service.

Command executed for `GET`:

```
curl https://localhost:8080/SpringDataRestExample/api/persons
```

Figure 8.1: CURL

Command executed for `POST` :

```
curl -i -X POST -H "Content-Type:application/json" -d "{ \"personName\" : \"Krishna\" , \" ↩
    personAge\" : \"120\" }" https://localhost:8080/SpringDataRestExample/api/persons
```

```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\cshambhunathsingh>curl -i -X POST -H "Content-Type:application/json" -d "{ \"personN
HTTP/1.1 201 Created
Server: Apache-Coyote/1.1
Location: http://localhost:8080/SpringDataRestExample/api/persons/1
Content-Type: application/hal+json
Transfer-Encoding: chunked
Date: Mon, 07 Sep 2015 09:57:43 GMT

{
  "personName" : "Krishna",
  "personAge" : 120.0,
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/SpringDataRestExample/api/persons/1"
    }
  }
}
```

Figure 8.2: POST

In the windows console, single quote is not recognized as such we are forced to use the double quotes and escape sequence for passing the values.

Command executed for `Search` :

```
curl https://localhost:8080/SpringDataRestExample/api/persons/search/byName?name=Krishna
```

```
C:\Users\cshambhunathsingh>curl http://localhost:8080/SpringDataRestExample/api/person
{
  "_embedded" : {
    "persons" : [ {
      "personName" : "Krishna",
      "personAge" : 120.0,
      "_links" : {
        "self" : {
          "href" : "http://localhost:8080/SpringDataRestExample/api/persons/1"
        }
      }
    } ]
  }
}
```

Figure 8.3: Search

We search using the query method `findByPersonName` we wrote in `PersonRepo` class. Had we not changed the `path` of the method, it would be accessed using the actual method name.

Command executed for `Delete` :

```
curl -X DELETE https://localhost:8080/SpringDataRestExample/api/persons/3
```

Figure 8.4: Delete

As you can see, this command deletes an entity with the id = 3.

8.1 Download the Source Code

Thus we demonstrated how the Spring Data REST repository is configured and we can use it for CRUD operation. However, the reader should note that this style of architecture is more suited for smaller to medium scale applications. In large scale applications omitting the service layer, altogether, may **not** be advisable.

Download You can download the full source code of this example here: **SpringDataRestExample.zip**

Chapter 9

Neo4j Example

In this example we shall demonstrate how to integrate Neo4J, a graph based NoSql database with Spring Data.

9.1 Introduction

Neo4j is an open source, graph based NoSQL database developed in Java and Scala. Like traditional relational Databases, Neo4J offers support to ACID properties. The graph based databases find their uses in use cases where the focus is strongly on inter-relationship between the entities of the domain like match-making,social networks, routing.

9.2 Installation

Neo4J can be downloaded from here. For the purpose of this demo we are using the community edition.

The user can install the Neo4j by simply following the steps provided by the installer, downloaded earlier.

9.3 Project Set-up

We shall use `Maven` to setup our project. Open Eclipse and create a simple Maven project and check the `skip archetype selection` checkbox on the dialogue box that appears. Replace the content of the existing `pom.xml` with the one provided below:

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↩
    XMLSchema-instance" xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven ↩
    .apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.jcg.springNeo4J</groupId>
  <artifactId>SpringDataNeo4JExample</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <dependencies>
    <dependency>
        <groupId>org.springframework.data</groupId>
        <artifactId>spring-data-neo4j</artifactId>
        <version>4.0.0.RELEASE</version>
    </dependency>

     <dependency>
         <groupId> org.neo4j </groupId>
         <artifactId> neo4j-kernel </artifactId>
```

```
         <version> 2.1.3 </version>
      </dependency>

      <dependency>
         <groupId> javax.transaction </groupId>
         <artifactId> jta </artifactId>
         <version> 1.1 </version>
      </dependency>

      <dependency>
         <groupId>javax.validation</groupId>
         <artifactId>validation-api</artifactId>
         <version>1.0.0.GA</version>
      </dependency>


</dependencies>

</project>
```

9.4 Implementation

We start by creating the entity models for our example. Here are the entities:

Account.java

```
package com.jcg.examples.entity;

import java.io.Serializable;

import org.springframework.data.neo4j.annotation.GraphId;
import org.springframework.data.neo4j.annotation.NodeEntity;

@NodeEntity
public class Account implements Serializable
{

    private static final long serialVersionUID = -8860106787025445177L;

    @GraphId
    private Long accountId;

    private String accountType;

    private Double balance;

                public Long getAccountId()
    {
                return accountId;
    }

                public void setAccountId(Long accountId)
    {
                this.accountId = accountId;
    }

                public String getAccountType()
    {
                return accountType;
    }
```

```
            public void setAccountType(String accountType)
{
            this.accountType = accountType;
}

            public Double getBalance()
{
            return balance;
}

            public void setBalance(Double balance)
{
            this.balance = balance;
}

            @Override
public String toString()
{
                return "Account [accountId=" + accountId + ", accountType=" +  ↩
                    accountType + ", balance=" + balance + "]";
}

            @Override
public int hashCode()
{
                final int prime = 31;
                int result = 1;
                result = prime * result + ((accountId == null) ? 0 : accountId.hashCode ↩
                    ());
                result = prime * result + ((accountType == null) ? 0 : accountType. ↩
                    hashCode());
                result = prime * result + ((balance == null) ? 0 : balance.hashCode());
                return result;
}

            @Override
public boolean equals(Object obj)
{
                if (this == obj)
                                return true;
                if (obj == null)
                                return false;
                if (getClass() != obj.getClass())
                                return false;
                Account other = (Account) obj;
                if (accountId == null)
                {
                                if (other.accountId != null)
                                                return false;
                }
                else if (!accountId.equals(other.accountId))
                                return false;
                if (accountType == null)
                {
                                if (other.accountType != null)
                                                return false;
                }
                else if (!accountType.equals(other.accountType))
                                return false;
                if (balance == null)
                {
```

```
                                        if (other.balance != null)
                                                        return false;
                        }
                        else if (!balance.equals(other.balance))
                                        return false;
                        return true;
        }



}
```

Person.java

```
package com.jcg.examples.entity;


import java.io.Serializable;

import org.springframework.data.neo4j.annotation.Fetch;
import org.springframework.data.neo4j.annotation.GraphId;
import org.springframework.data.neo4j.annotation.NodeEntity;
import org.springframework.data.neo4j.annotation.RelatedTo;


@NodeEntity
public class Person implements Serializable
{

                private static final long serialVersionUID = -5378396373373165919L;

                @GraphId
                private Long id;

                private String personName;

                @RelatedTo
                @Fetch
                private Account account;


                public Long getId()
                {
                                return id;
                }

                public void setId(Long id)
                {
                                this.id = id;
                }

                public Account getAccount()
        {
                return account;
        }

                public void setAccount(Account account)
        {
                this.account = account;
        }

                @Override
```

```
    public String toString()
    {
                    return "Person [id=" + id + ", account=" + account + "]";
    }

                @Override
    public int hashCode()
    {
                    final int prime = 31;
                    int result = 1;
                    result = prime * result + ((account == null) ? 0 : account.hashCode());
                    result = prime * result + ((id == null) ? 0 : id.hashCode());
                    return result;
    }

                @Override
    public boolean equals(Object obj)
    {
                    if (this == obj)
                                    return true;
                    if (obj == null)
                                    return false;
                    if (getClass() != obj.getClass())
                                    return false;
                    Person other = (Person) obj;
                    if (account == null)
                    {
                                    if (other.account != null)
                                                    return false;
                    }
                    else if (!account.equals(other.account))
                                    return false;
                    if (id == null)
                    {
                                    if (other.id != null)
                                                    return false;
                    }
                    else if (!id.equals(other.id))
                                    return false;
                    return true;
    }

                public String getPersonName()
                {
                                return personName;
                }

                public void setPersonName(String personName)
                {
                                this.personName = personName;
                }

}
```

- `@NodeEntity` This annotation is used to mark the PoJo as Neo4J Entity.
- `@GraphId` This marks the annotated field as the Node ID. It must be of the type `java.lang.Long`. If the field name is `id`, it need not be annotated.
- `@RelatedTo` annotation is used to relate to other entities.
- `@Fetch` If this tag is present on a relationship property, it eagerly fetches that entity.

Now that the entities are configured, we can create the DAO layer by configuring the basic repositories:

AccountRepo.java

```
package com.jcg.examples.repo;

import org.springframework.data.neo4j.repository.GraphRepository;
import org.springframework.stereotype.Repository;

import com.jcg.examples.entity.Account;


@Repository
public interface AccountRepo extends GraphRepository
{
}
```

PersonRepo.java

```
package com.jcg.examples.repo;

import org.springframework.data.neo4j.repository.GraphRepository;
import org.springframework.stereotype.Repository;

import com.jcg.examples.entity.Person;

@Repository
public interface PersonRepo extends GraphRepository
{

}
```

Spring Data provides a number of inbuilt method for manipulating the Data. We need not write the queries for basic data manipulation and reading. It is achieved by extending the `GraphRepository` and declaring the proper Generics as per the PoJo, which in our case is the `Person` and `Account`.

In case the Developer is not satisfied with the existing method, he can create his own method by specifying the Query using the `@Query` annotation. The Spring IoC Container creates an instance of this Repository and makes it available to be used as a Bean since we have annotated it with the stereotype annotation `@Repository` and enabled `component-scan` in the spring configuration xml.

Here's a trivial implementation of the service layer for `Person`

PersonService.java

```
package com.jcg.examples.service;

import java.util.Collection;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import com.jcg.examples.entity.Person;
import com.jcg.examples.repo.PersonRepo;

@Service
public class PersonService
{

                @Autowired
                private PersonRepo personRepo;
```

```
                @Transactional
                public Person save(Person person)
                {
                                return personRepo.save(person);
                }

                @Transactional
                public void delete(Long personId)
                {
                                personRepo.delete(personId);
                }

                @Transactional
                public Person get(Long personId)
                {
                                return personRepo.findOne(personId);
                }

                @SuppressWarnings("unchecked")
                public Collection findAll()
                {
                                return personRepo.findAll().as(Collection.class);
                }

                public PersonRepo getPersonRepo()
                {
                                return personRepo;
                }

                public void setPersonRepo(PersonRepo personRepo)
                {
                                this.personRepo = personRepo;
                }
}
```

We have annotated the service methods with `@Transactional` to wrap around the operations within a transaction boundary. The last and the most important part is to configure the Spring Container using the `spring-configuration.xml` :

spring-configuration.xml

```
<beans xmlns="https://www.springframework.org/schema/beans"
       xmlns:context="https://www.springframework.org/schema/context"
       xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
       xmlns:neo4j="https://www.springframework.org/schema/data/neo4j"
       xmlns:tx="https://www.springframework.org/schema/tx"
       xsi:schemaLocation="https://www.springframework.org/schema/beans https://www. ↩
           springframework.org/schema/beans/spring-beans-3.0.xsd
        https://www.springframework.org/schema/context https://www.springframework.org/ ↩
            schema/context/spring-context-3.0.xsd
        https://www.springframework.org/schema/tx https://www.springframework.org/schema/tx ↩
            /spring-tx.xsd
        https://www.springframework.org/schema/data/neo4j https://www.springframework.org/ ↩
            schema/data/neo4j/spring-neo4j.xsd">

    <context:component-scan base-package="com.jcg.examples" />

    <neo4j:config storeDirectory="C:\\Users\\chandansingh\\Documents\\Neo4j" base-package=" ↩
        com.jcg.examples.entity"/>

    <neo4j:repositories base-package="com.jcg.examples.repo"/>

    <bean id="applicationTest" class="com.jcg.examples.main.ApplicationTest" />
```

```
</beans>
```

Line-13 : Declares the Neo4J store location and the location of the Neo4J entities.

Line-15 : Scan and initiate the Neo4J repositories.

Now that all is set, let's run the application and test out the code! Here's Application class that loads the XML file to instantiate the Spring Container and execute a few queries.

ApplicationTest.java

```
package com.jcg.examples.main;


import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.core.io.ClassPathResource;

import com.jcg.examples.entity.Account;
import com.jcg.examples.entity.Person;
import com.jcg.examples.service.AccountService;
import com.jcg.examples.service.PersonService;


public class ApplicationTest
{
                @Autowired
                private PersonService personService;

                @Autowired
                private AccountService accountService;

                public static void main(String[] args)
                {
                                ApplicationContext context = new  ↩
                                    ClassPathXmlApplicationContext(new ClassPathResource(" ↩
                                    spring-configuration.xml").getPath());
                                ApplicationTest applicationTest = context.getBean( ↩
                                    ApplicationTest.class);
                                System.out.println("Starting with the Tests..");
                                Long personId = applicationTest.savePerson();
                                applicationTest.printPerson(personId);
                                System.out.println("Finished!");

                }

                private Long savePerson()
                {
                                Person person = new Person();
                                person.setPersonName("Chandan Singh");
                                Account account = new Account();
                                account.setBalance(212.21);
                                account.setAccountType("Savings");
                                person.setAccount(account);
                                person = personService.save(person);
                                System.out.println("Saved Person sucessfully!");
                                return person.getId();
                }

                private void printPerson(Long personId)
                {
```

```
                        System.out.println(personService.get(personId));
                }


                public PersonService getPersonService()
                {
                        return personService;
                }

                public void setPersonService(PersonService personService)
                {
                        this.personService = personService;
                }

                public AccountService getAccountService()
                {
                        return accountService;
                }

                public void setAccountService(AccountService accountService)
                {
                        this.accountService = accountService;
                }



}
```

Here's the sample output of the program :

```
Starting with the Tests..
Saved Person sucessfully!
Person [id=6, account=Account [accountId=7, accountType=Savings, balance=212.21]]
Finished!
```

9.5 Download the Source Code

In this example, we studied how we can integrate Neo4J with Spring Data.

Download You can download the full source code of this example here: **SpringDataNeo4JExample.zip**

Chapter 10

Elasticsearch Example

Elasticsearch is a highly scalable open-source which can be used for data store, text search and analytics engine. Every instance of ElasticSearch is called a node and several nodes can be grouped together in a cluster.

In this article, we will see how we can use spring-data-elasticsearch module which integrates spring-data and elasticsearch.

10.1 Dependencies

Include `<spring-core>`, `<spring-context>` and `<spring-data-elasticsearch>` in your `pom.xml`.

pom.xml:

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↩
    XMLSchema-instance"
        xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd ↩
            /maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <groupId>com.javacodegeeks.data.elasticsearch</groupId>
        <artifactId>springDataElasticsearchExample</artifactId>
        <version>1.0-SNAPSHOT</version>
        <packaging>jar</packaging>
        <name>SpringElasticsearchExample</name>
        <description>Example of spring elasticsearch</description>

        <properties>
                <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
                <java.version>1.8</java.version>
        </properties>

        <dependencies>
                <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-core</artifactId>
                        <version>4.1.5.RELEASE</version>
                </dependency>
                <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-context</artifactId>
                        <version>4.1.5.RELEASE</version>
                </dependency>
                <dependency>
                        <groupId>org.springframework.data</groupId>
                        <artifactId>spring-data-elasticsearch</artifactId>
                        <version>1.3.2.RELEASE</version>
```

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

10.2 Start Elasticsearch Sever

Download the latest Elasticsearch and unzip the file. Go to elasticsearch working folder/bin, in our case `elasticsearch-2.1.1bin` and run `elasticsearch` command. `elasticsearch.yml` is the main configuration file for ElasticSearch. We can set here the node name and cluster name.

```
C:\elasticsearch-2.1.1\bin<elasticsearch
[2016-01-17 20:20:22,821][WARN ][bootstrap                ] unable to install sy
scall filter: syscall filtering not supported for OS: 'Windows 7'
[2016-01-17 20:20:23,043][INFO ][node                     ] [Wilbur Day] version
[2.1.1], pid[9784], build[40e2c53/2015-12-15T13:05:55Z]
[2016-01-17 20:20:23,043][INFO ][node                     ] [Wilbur Day] initial
izing ...
[2016-01-17 20:20:23,123][INFO ][plugins                  ] [Wilbur Day] loaded
[], sites []
[2016-01-17 20:20:23,149][INFO ][env                      ] [Wilbur Day] using [
1] data paths, mounts [[OSDisk (C:)]], net usable_space [24.2gb], net total_spac
e [476gb], spins? [unknown], types [NTFS]
[2016-01-17 20:20:25,551][INFO ][node                     ] [Wilbur Day] initial
ized
[2016-01-17 20:20:25,552][INFO ][node                     ] [Wilbur Day] startin
g ...
[2016-01-17 20:20:25,903][INFO ][transport                ] [Wilbur Day] publish
_address {127.0.0.1:9300}, bound_addresses {127.0.0.1:9300}, {[::1]:9300}
[2016-01-17 20:20:25,912][INFO ][discovery                ] [Wilbur Day] elastic
search/d5McLMFpTNGpnYEZDacPvg
[2016-01-17 20:20:29,945][INFO ][cluster.service          ] [Wilbur Day] new_mas
ter {Wilbur Day}{d5McLMFpTNGpnYEZDacPvg}{127.0.0.1}{127.0.0.1:9300}, reason: zen
-disco-join(elected_as_master, [0] joins received)
[2016-01-17 20:20:30,002][INFO ][gateway                  ] [Wilbur Day] recover
ed [0] indices into cluster_state
[2016-01-17 20:20:30,160][INFO ][http                     ] [Wilbur Day] publish
_address {127.0.0.1:9200}, bound_addresses {127.0.0.1:9200}, {[::1]:9200}
[2016-01-17 20:20:30,160][INFO ][node                     ] [Wilbur Day] started
```

10.3 Elasticsearch Storage Structure

Before we start our spring data elasticsearch example, its important to understand the Elasticsearch storage structure.

- **Index**- This is the main data container it is analogous to database in SQL
- **Mappings**- Data is organized as data types called mappings. The equivalent structure in SQL is table.
- **Field**- A mapping contains records which in rurn are composed of fields.
- **Object**- This is the format of a record which is in form of JSON object

10.4 Store and retrieve Data

In order to have a working system, all we need to do is define domain entities and a repository class for the support of CRUD machinery. In order to mark a POJO class as domain entity, we just need to add `org.springframework.data.elasticsearch.annotations.Document` to our index object. indexing your objects to Elasticsearch is to add the @Document annotation to them and create a Repository interface extending ElasticsearchRepository.

Let's first define index and mapping.

Employee:

```
package com.javacodegeeks.spring.elasticsearch;

import java.util.List;

import org.springframework.data.annotation.Id;
import org.springframework.data.elasticsearch.annotations.Document;
import org.springframework.data.elasticsearch.annotations.Field;
import org.springframework.data.elasticsearch.annotations.FieldType;

@Document(indexName = "resource", type = "employees")
public class Employee {
        @Id
        private String id;
        private String name;
        private Integer age;

    @Field( type = FieldType.Nested)
        private List<Skill> skills;

        public Employee(){}

        public Employee(String id, String name, int age) {
                this.id = id;
                this.name = name;
                this.age = age;
        }

        public String getId() {
                return id;
        }

        public void setId(String id) {
                this.id = id;
        }

        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }

        public Integer getAge() {
                return age;
        }

        public void setAge(Integer age) {
                this.age = age;
        }
```

```
        public List<Skill> getSkills() {
                return skills;
        }

        public void setSkills(List<Skill> skills) {
                this.skills = skills;
        }

        public String toString() {
                return "Employee [(" + getId() + ", " + getName() + ", " + age + "), skills ↩
                    : " + getSkills() + "]";
        }
}
```

It depends on POJO skill which is our embedded object so its type is defined as `FieldType.NESTED`.

Skill:

```
package com.javacodegeeks.spring.elasticsearch;

public class Skill {
        private String name;
        private int experience;

        public Skill() {
        }

        public Skill(String name, int experience) {
                this.name = name;
                this.experience = experience;
        }

        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }

        public int getExperience() {
                return experience;
        }

        public void setExperience(int experience) {
                this.experience = experience;
        }

        public String toString() {
                return "Skill(" + name + ", " + experience + ")";
        }
}
```

10.5 Employee Repository

`EmployeeRepository` extends spring data provided `ElasticsearchRepository` which is the base repository class for elasticsearch based domain classes. Since it extends Spring based repository classes, we get the benefit of avoiding boilerplate code required to implement data access layers for various persistence stores.

`Repository` is the central markup interface in Spring Data repository. It takes the domain class to manage as well as the id type of the domain class as type arguments. Its main purpose is to make the repository typed. The next main interface is `CrudRepository` which provides sophisticated CRUD functionality for the entity class that is being managed. On top of the `CrudRepository` there is a `PagingAndSortingRepository` abstraction that adds additional methods to ease paginated access to entities.

Declare query methods on the interface. Since we are using Spring JPA repository we don't need to write implementation for it.

EmployeeRepository:

```
package com.javacodegeeks.spring.elasticsearch;

import java.util.List;

import org.springframework.data.elasticsearch.repository.ElasticsearchRepository;


public interface EmployeeRepository extends ElasticsearchRepository {
    List findEmployeesByAge(int age);
    List findEmployeesByName(String name);
    List findEmployeesBySkillsIn(List skills);
}
```

10.6 Spring Configuration

The Spring Data Elasticsearch module contains a custom namespace allowing definition of repository beans as well as elements for instantiating a ElasticsearchServer.

```
xmlns:elasticsearch="https://www.springframework.org/schema/data/elasticsearch
```

Spring is instructed to scan `com.javacodegeeks.spring.elasticsearch` and all its sub-packages for interfaces extending Repository or one of its sub-interfaces.

Next, we are using a Node Client element to register an instance of Elasticsearch Server in the context.

```
<elasticsearch:node-client id="client" local="true"/>
```

If you want to create `NodeClient` programatically, you can do it using node builder.

```
private static NodeClient getNodeClient() {
                return (NodeClient) nodeBuilder().clusterName(UUID.randomUUID().toString()) ↩
                    .local(true).node()
                                .client();
        }
```

applicationContext.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="https://www.springframework.org/schema/beans"
       xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="https://www.springframework.org/schema/context"
       xmlns:elasticsearch="https://www.springframework.org/schema/data/elasticsearch"
       xsi:schemaLocation="https://www.springframework.org/schema/data/elasticsearch https: ↩
           //www.springframework.org/schema/data/elasticsearch/spring-elasticsearch-1.0.xsd
                https://www.springframework.org/schema/beans https://www.springframework. ↩
                    org/schema/beans/spring-beans-3.2.xsd
                https://www.springframework.org/schema/context https://www.springframework. ↩
                    org/schema/context/spring-context.xsd">

        <context:annotation-config/>
```

```
        <bean name="mainBean" class="com.javacodegeeks.spring.elasticsearch. ↩
            SpringElasticsearchExample"/>

    <elasticsearch:repositories base-package="com.javacodegeeks.spring.elasticsearch"/>
    <elasticsearch:node-client id="client" local="true"/>

    <bean name="elasticsearchTemplate" class="org.springframework.data.elasticsearch.core. ↩
        ElasticsearchTemplate">
        <constructor-arg name="client" ref="client"/>
    </bean>
</beans>
```

10.7 Configuration using Annotation

The Spring Data Elasticsearch repositories scan also be activated using annotation `@EnableElasticsearchRepositories`.

SpringElasticsearchExampleUsingAnnotation:

```
@Configuration("mainBean")
@EnableElasticsearchRepositories(basePackages = "com.javacodegeeks.spring.elasticsearch")
public class SpringElasticsearchExampleUsingAnnotation {
        @Autowired
        private EmployeeRepository repository;

        @Autowired
        private ElasticsearchTemplate template;


        @Bean
        public ElasticsearchTemplate elasticsearchTemplate() {
                return new ElasticsearchTemplate(getNodeClient());
        }
...
}
```

10.8 Run the Example

`SpringElasticsearchExample` loads the spring context. It next gets the `SpringElasticsearchExample` bean and adds few employees. We then execute several finder methods to list the employees.

The the repository instance `EmployeeRepository` is injected into it using `@Autowired`.

We also inject bean `ElasticsearchTemplate` which is the central class that spring provides using which we save our domain entities.

SpringElasticsearchExample:

```
package com.javacodegeeks.spring.elasticsearch;

import java.net.URISyntaxException;
import java.util.Arrays;
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.data.elasticsearch.core.ElasticsearchTemplate;
```

```
import org.springframework.data.elasticsearch.core.query.IndexQuery;

@Configuration
public class SpringElasticsearchExample {
        @Autowired
        private EmployeeRepository repository;

        @Autowired
        private ElasticsearchTemplate template;

        public static void main(String[] args) throws URISyntaxException, Exception {
                ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext(
                                "applicationContext.xml");
                try {
                        System.out.println("Load context");
                        SpringElasticsearchExample s = (SpringElasticsearchExample) ctx
                                        .getBean("mainBean");
                        System.out.println("Add employees");
                        s.addEmployees();
                        System.out.println("Find all employees");
                        s.findAllEmployees();
                        System.out.println("Find employee by name 'Joe'");
                        s.findEmployee("Joe");
                        System.out.println("Find employee by name 'John'");
                        s.findEmployee("John");
                        System.out.println("Find employees by age");
                        s.findEmployeesByAge(32);
                } finally {
                        ctx.close();
                }
        }

        public void addEmployees() {
                Employee joe = new Employee("01", "Joe", 32);
                Skill javaSkill = new Skill("Java", 10);
                Skill db = new Skill("Oracle", 5);
                joe.setSkills(Arrays.asList(javaSkill, db));
                Employee johnS = new Employee("02", "John S", 32);
                Employee johnP = new Employee("03", "John P", 42);
                Employee sam = new Employee("04", "Sam", 30);

                template.putMapping(Employee.class);
                IndexQuery indexQuery = new IndexQuery();
                indexQuery.setId(joe.getId());
                indexQuery.setObject(joe);
                template.index(indexQuery);
                template.refresh(Employee.class, true);
                repository.save(johnS);
                repository.save(johnP);
                repository.save(sam);
        }

        public void findAllEmployees() {
                repository.findAll().forEach(System.out::println);
        }

        public void findEmployee(String name) {
                List empList = repository.findEmployeesByName(name);
                System.out.println("Employee list: " + empList);
        }

        public void findEmployeesByAge(int age) {
```

```
                List empList = repository.findEmployeesByAge(age);
                System.out.println("Employee list: " + empList);
        }
}
```

Output:

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See https://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Load context
Add employees
Find all employees
Employee [(04, Sam, 30), skills: null]
Employee [(01, Joe, 32), skills: [Skill(Java, 10), Skill(Oracle, 5)]]
Employee [(02, John S, 32), skills: null]
Employee [(03, John P, 42), skills: null]
Find employee by name 'Joe'
Employee list: [Employee [(01, Joe, 32), skills: [Skill(Java, 10), Skill(Oracle, 5)]]]
Find employee by name 'John'
Employee list: [Employee [(02, John S, 32), skills: null], Employee [(03, John P, 42),  ↩
    skills: null]]
Find employees by age
Employee list: [Employee [(01, Joe, 32), skills: [Skill(Java, 10), Skill(Oracle, 5)]],  ↩
    Employee [(02, John S, 32), skills: null]]
```

10.9 Annotation based main class

We can recreate the above example using just the annotations. If you notice, we have created `ElasticsearchTemplate`. programatically.</code>

```
@Bean
public ElasticsearchTemplate elasticsearchTemplate() {
        return new ElasticsearchTemplate(getNodeClient());
}
```

annotationApplicationContext.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="https://www.springframework.org/schema/beans"
       xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="https://www.springframework.org/schema/context"
       xmlns:elasticsearch="https://www.springframework.org/schema/data/elasticsearch"
       xsi:schemaLocation="https://www.springframework.org/schema/data/elasticsearch https: ↩
           //www.springframework.org/schema/data/elasticsearch/spring-elasticsearch-1.0.xsd
                https://www.springframework.org/schema/beans https://www.springframework. ↩
                    org/schema/beans/spring-beans-3.2.xsd
                https://www.springframework.org/schema/context https://www.springframework. ↩
                    org/schema/context/spring-context.xsd">

        <context:annotation-config/>

    <elasticsearch:node-client id="client" local="true"/>

    <bean name="elasticsearchTemplate" class="org.springframework.data.elasticsearch.core. ↩
        ElasticsearchTemplate">
        <constructor-arg name="client" ref="client"/>
    </bean>
</beans>
```

SpringElasticsearchExampleUsingAnnotation:

```
package com.javacodegeeks.spring.elasticsearch;

import static org.elasticsearch.node.NodeBuilder.nodeBuilder;

import java.net.URISyntaxException;
import java.util.Arrays;
import java.util.List;
import java.util.UUID;

import org.elasticsearch.client.node.NodeClient;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.elasticsearch.core.ElasticsearchTemplate;
import org.springframework.data.elasticsearch.core.query.IndexQuery;
import org.springframework.data.elasticsearch.repository.config. ↩
    EnableElasticsearchRepositories;

@Configuration("mainBean")
@EnableElasticsearchRepositories(basePackages = "com.javacodegeeks.spring.elasticsearch")
public class SpringElasticsearchExampleUsingAnnotation {
        @Autowired
        private EmployeeRepository repository;

        @Autowired
        private ElasticsearchTemplate template;


        @Bean
        public ElasticsearchTemplate elasticsearchTemplate() {
                return new ElasticsearchTemplate(getNodeClient());
        }

        public static void main(String[] args) throws URISyntaxException, Exception {
                AnnotationConfigApplicationContext ctx = new  ↩
                    AnnotationConfigApplicationContext();
                try {
                        ctx.register(SpringElasticsearchExampleUsingAnnotation.class);
                        ctx.refresh();
                        System.out.println("Load context");
                        SpringElasticsearchExampleUsingAnnotation s = ( ↩
                            SpringElasticsearchExampleUsingAnnotation) ctx
                                        .getBean("mainBean");
                        System.out.println("Add employees");
                        s.addEmployees();
                        System.out.println("Find all employees");
                        s.findAllEmployees();
                        System.out.println("Find employee by name 'Joe'");
                        s.findEmployee("Joe");
                        System.out.println("Find employee by name 'John'");
                        s.findEmployee("John");
                        System.out.println("Find employees by age");
                        s.findEmployeesByAge(32);
                } finally {
                        ctx.close();
                }
        }

        public void addEmployees() {
```

```
                Employee joe = new Employee("01", "Joe", 32);
                Skill javaSkill = new Skill("Java", 10);
                Skill db = new Skill("Oracle", 5);
                joe.setSkills(Arrays.asList(javaSkill, db));
                Employee johnS = new Employee("02", "John S", 32);
                Employee johnP = new Employee("03", "John P", 42);
                Employee sam = new Employee("04", "Sam", 30);

                template.putMapping(Employee.class);
                IndexQuery indexQuery = new IndexQuery();
                indexQuery.setId(joe.getId());
                indexQuery.setObject(joe);
                template.index(indexQuery);
                template.refresh(Employee.class, true);
                repository.save(johnS);
                repository.save(johnP);
                repository.save(sam);
        }

        public void findAllEmployees() {
                repository.findAll().forEach(System.out::println);
        }

        public void findEmployee(String name) {
                List<Employee> empList = repository.findEmployeesByName(name);
                System.out.println("Employee list: " + empList);
        }

        public void findEmployeesByAge(int age) {
                List<Employee> empList = repository.findEmployeesByAge(age);
                System.out.println("Employee list: " + empList);
        }

        private static NodeClient getNodeClient() {
                return (NodeClient) nodeBuilder().clusterName(UUID.randomUUID().toString()) ↩
                    .local(true).node()
                                .client();
        }
}
```

10.10 Elasticsearch as RESTFul Server

ElasticSearch can also be used as a RESTful server, the main protocol is the HTTP, listening on port number 9200 (default).

To view the index type: Enter `https://localhost:9200/resource/employees/_search` in your browser.

```
{"resource":{"aliases":{},"mappings":{"employees":{"properties":{"age":{"type":"long"},"id" ↩
    :{"type":"string"},"name":{"type":"string"},"skills":{"type":"nested","properties":{" ↩
    experience":{"type":"long"},"name":{"type":"string"}}}}}},"settings":{"index":{" ↩
    refresh_interval":"1s","number_of_shards":"5","creation_date":"1453094779722","store":{" ↩
    type":"fs"},"uuid":"7YWl_3VBTq-eluY74GU4sQ","version":{"created":"1050299"}," ↩
    number_of_replicas":"1"}},"warmers":{}}}
```

To find all employees, enter `https://localhost:9200/resource/employees/_search`. Here's the result JSON object.

```
{"took":6,"timed_out":false,"_shards":{"total":5,"successful":5,"failed":0},"hits":{"total" ↩
    :4,"max_score":1.0,"hits":[{"_index":"resource","_type":"employees","_id":"04","_score" ↩
    :1.0,"_source":{"id":"04","name":"Sam","age":30,"skills":null}},{"_index":"resource"," ↩
    _type":"employees","_id":"01","_score":1.0,"_source":{"id":"01","name":"Joe","age":32," ↩
```

```
    skills":[{"name":"Java","experience":10},{"name":"Oracle","experience":5}]}},{"_index":" ↩
    resource","_type":"employees","_id":"02","_score":1.0,"_source":{"id":"02","name":"John  ↩
    S","age":32,"skills":null}},{"_index":"resource","_type":"employees","_id":"03","_score" ↩
    :1.0,"_source":{"id":"03","name":"John P","age":42,"skills":null}}]}}
```


You can also restrict the employees per page. For example, enter `https://localhost:9200/resource/employees/_search?page=1&size=2` to fetch two employees in the first page.


```
{"took":2,"timed_out":false,"_shards":{"total":5,"successful":5,"failed":0},"hits":{"total" ↩
    :4,"max_score":1.0,"hits":[{"_index":"resource","_type":"employees","_id":"04","_score" ↩
    :1.0,"_source":{"id":"04","name":"Sam","age":30,"skills":null}},{"_index":"resource"," ↩
    _type":"employees","_id":"01","_score":1.0,"_source":{"id":"01","name":"Joe","age":32," ↩
    skills":[{"name":"Java","experience":10},{"name":"Oracle","experience":5}]}}]}}
```


You can also fetch the employee by ID. For example, enter `https://localhost:9200/resource/employees/01`


```
{"_index":"resource","_type":"employees","_id":"01","_version":3,"found":true,"_source":{" ↩
    id":"01","name":"Joe","age":32,"skills":[{"name":"Java","experience":10},{"name":"Oracle ↩
    ","experience":5}]}}
```


10.11 Download the Eclipse Project

This was an example about spring data Elasticsearch.

Download You can download the full source code of this example here: **springDataElasticsearch.zip**

Chapter 11

JPA Example

The goal of Spring Data repository abstraction is to significantly reduce the amount of boilerplate code required to implement data access layers for various persistence stores. If we try to implement a data access layer of an application on our own, we will end up with lots of boilerplate code.

Spring provides JPA module which not only eliminates boilerplate code but also takes care of the CRUD operations, provides inbuilt implementation for simple queries, performs pagination and auditing. Using Spring Data JPA we can write our own repository interfaces, including custom finder methods, and Spring will provide the implementation automatically. In this article we will see some examples of Spring Data JPA.

11.1 Dependencies

Since we are using Spring Data Jpa, we certainly need to add spring-data-jpa. We also need a database. In this example, we will use HSQL database in embedded mode. We will use hibernate as the JPA vendor so you need to add hibernate-jpa-2.0-api and hibernate-entitymanager. Other than that we also need the spring-core and spring-context.

pom.xml:

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↩
    XMLSchema-instance"
        xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd ↩
            /maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <groupId>com.javacodegeeks.camel</groupId>
        <artifactId>springQuartzScheduler</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <dependencies>
                <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-core</artifactId>
                        <version>4.1.5.RELEASE</version>
                </dependency>
                <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-context</artifactId>
                        <version>4.1.5.RELEASE</version>
                </dependency>
                <dependency>
                        <groupId>org.springframework.data</groupId>
                        <artifactId>spring-data-jpa</artifactId>
                        <version>1.8.0.RELEASE</version>
                </dependency>
                <dependency>
```

```
                        <groupId>org.hsqldb</groupId>
                        <artifactId>hsqldb</artifactId>
                        <version>2.2.9</version>
                        <scope>runtime</scope>
                </dependency>
                <dependency>
                        <groupId>org.hibernate.javax.persistence</groupId>
                        <artifactId>hibernate-jpa-2.0-api</artifactId>
                        <version>1.0.1.Final</version>
                </dependency>
                <dependency>
                        <groupId>org.hibernate</groupId>
                        <artifactId>hibernate-entitymanager</artifactId>
                        <version>4.3.10.Final</version>
                        <scope>runtime</scope>
                </dependency>

        </dependencies>
</project>
```

11.2 Entity Bean

Let's first define an entity bean. It is an employee class, marked with @Entity. It has two columns, age and name. Other than these two,there is an auto-generated ID column.

Employee:

```
package com.javacodegeeks.spring.jpa;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Employee {
        @Id
        @GeneratedValue
        private Long id;
        private String name;
        private Integer age;

        public Employee(){}

        public Employee(String name, int age) {
                this.name = name;
                this.age = age;
        }

        public Long getId() {
                return id;
        }

        public void setId(Long id) {
                this.id = id;
        }

        public String getName() {
                return name;
        }
```

```
        public void setName(String name) {
                this.name = name;
        }

        public Integer getAge() {
                return age;
        }

        public void setAge(Integer age) {
                this.age = age;
        }

        public String toString() {
                return "Employee (" + getId() + ", " + getName() + ", " + age + ")";
        }

        @Override
        public boolean equals(Object obj) {

                if (this == obj) {
                        return true;
                }

                if (this.id == null || obj == null || !(this.getClass().equals(obj.getClass ↩
                    ())) {
                        return false;
                }

                Employee o = (Employee) obj;

                return this.id.equals(o.getId());
        }

        @Override
        public int hashCode() {
                return id == null ? 0 : id.hashCode();
        }
}
```

11.3 Initialization of Database

We also want to initialize the database with some sample data as the application starts. Here is some sample employee data.

data.sql:

```
insert into Employee (id, name, age) values (1, 'Joe', 32);
insert into Employee (id, name, age) values (2, 'Sam', 28);
insert into Employee (id, name, age) values (3, 'John', 43);
```

11.4 Configure Entity Manager Factory and Transaction Manager

Let's begin configuring the entity manager factory and transaction manager.

EntityManagerFactory - An entity manager factory provides entity manager instances, all instances are configured to connect to the same database, to use the same default settings as defined by the particular implementation, etc. You can prepare several entity manager factories to access several data stores. This interface is similar to the `SessionFactory` in native Hibernate.

EntityManager - This is used to access a database in a particular unit of work. It is used to create and remove persistent entity instances, to find entities by their primary key identity, and to query over all entities. This interface is similar to the `Session` in Hibernate.

We will use HSQL as the database in embedded mode. To specify HSQL explicitly, set the `type` attribute of the `embedded-database` tag to `HSQL`. If you are using the builder API, call the `setType(EmbeddedDatabaseType)` method with `EmbeddedDatabaseType.HSQL`.

`@EnableTransactionManagement` Enables Spring's annotation-driven transaction management capability.

Using the below `JpaConfig`, we provide DataSource bean, EntityManager factory, transaction manager and a database populator bean.

JpaConfig:

```
package com.javacodegeeks.spring.jpa;

import java.sql.Connection;
import java.sql.SQLException;

import javax.sql.DataSource;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Lazy;
import org.springframework.core.io.ClassPathResource;
import org.springframework.jdbc.datasource.DataSourceUtils;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;
import org.springframework.jdbc.datasource.init.ResourceDatabasePopulator;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.Database;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;

@Configuration
@EnableTransactionManagement
public class JpaConfig {

        /**
         * Bootstraps an in-memory HSQL database.
         */
        @Bean
        public DataSource dataSource() {
                EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
                return builder.setType(EmbeddedDatabaseType.HSQL).build();
        }

        /**
         * Picks up entities from the project's base package.
         */
        @Bean
        public LocalContainerEntityManagerFactoryBean entityManagerFactory() {

                HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
                vendorAdapter.setDatabase(Database.HSQL);
                vendorAdapter.setGenerateDdl(true);

                LocalContainerEntityManagerFactoryBean factory = new  ↩
                    LocalContainerEntityManagerFactoryBean();
                factory.setJpaVendorAdapter(vendorAdapter);
                factory.setPackagesToScan(getClass().getPackage().getName());
```

```
                factory.setDataSource(dataSource());

                return factory;
        }

        @Bean
        public PlatformTransactionManager transactionManager() {

                JpaTransactionManager txManager = new JpaTransactionManager();
                txManager.setEntityManagerFactory(entityManagerFactory().getObject());
                return txManager;
        }

        @Bean
        @Lazy(false)
        public ResourceDatabasePopulator populateDatabase() throws SQLException {

                ResourceDatabasePopulator populator = new ResourceDatabasePopulator();
                populator.addScript(new ClassPathResource("data.sql"));

                Connection connection = null;

                try {
                        connection = DataSourceUtils.getConnection(dataSource());
                        populator.populate(connection);
                } finally {
                        if (connection != null) {
                                DataSourceUtils.releaseConnection(connection, dataSource()) ↩
                                    ;
                        }
                }
                return populator;
        }
}
```

11.5 Configure in Spring XML Context

The same Spring Configuration with XML:

applicationContext.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="https://www.springframework.org/schema/beans"
        xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance" xmlns:jdbc="https://www. ↩
            springframework.org/schema/jdbc"
        xmlns:tx="https://www.springframework.org/schema/tx" xmlns:jpa="https://www. ↩
            springframework.org/schema/data/jpa"
        xmlns:context="https://www.springframework.org/schema/context"
        xsi:schemaLocation="https://www.springframework.org/schema/jdbc https://www. ↩
            springframework.org/schema/jdbc/spring-jdbc.xsd
                https://www.springframework.org/schema/beans https://www.springframework. ↩
                    org/schema/beans/spring-beans.xsd
                https://www.springframework.org/schema/data/jpa https://www.springframework ↩
                    .org/schema/data/jpa/spring-jpa.xsd
                https://www.springframework.org/schema/tx https://www.springframework.org/ ↩
                    schema/tx/spring-tx.xsd
                https://www.springframework.org/schema/context https://www.springframework. ↩
                    org/schema/context/spring-context.xsd">
```

```
        <bean id="transactionManager" class="org.springframework.orm.jpa. ↩
            JpaTransactionManager">
                <property name="entityManagerFactory" ref="entityManagerFactory" />
        </bean>

        <bean id="entityManagerFactory"
                class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
                <property name="dataSource" ref="dataSource" />
                <property name="packagesToScan" value="com.javacodegeeks.spring.jpa" />
                <property name="jpaVendorAdapter">
                        <bean class="org.springframework.orm.jpa.vendor. ↩
                            HibernateJpaVendorAdapter">
                                <property name="database" value="HSQL" />
                                <property name="generateDdl" value="true" />
                        </bean>
                </property>
        </bean>

        <jdbc:initialize-database data-source="dataSource">
                <jdbc:script location="classpath:data.sql" />
        </jdbc:initialize-database>

        <jdbc:embedded-database id="dataSource" type="HSQL" />

</beans>
```

11.6 CRUD Repository

The central interface in Spring Data repository abstraction is Repository interface. It takes the domain class to manage as well as the id type of the domain class as type arguments. This interface acts primarily as a marker interface to capture the types to work with and to help you to discover interfaces that extend this one.

The CrudRepository provides sophisticated CRUD functionality for the entity class that is being managed.

CrudRepository:

```
public interface CrudRepository<T, ID extends Serializable> extends Repository<T, ID> {

        <S extends T> S save(S entity);

...
        T findOne(ID id);

        Iterable<T> findAll();

        void delete(ID id);

...
}
```

The repository interface provides the CRUD operations

- Saves the given entity.
- Returns the entity identified by the given id.
- Returns all entities.
- Returns the number of entities.

- Deletes the given entity.
- Indicates whether an entity with the given id exists.

Spring also provides JpaRepository which is JPA specific, provides persistence technology-specific abstractions.

11.7 Employee Repository

We will now define employee specific repository. We come up with an employee repository extending Repository or one of its sub-interfaces.

If you want to write your own query method, you can use a named query using annotation the Spring Data JPA @Query annotation. Using named queries, we can declare queries for entities.

If you want paging, you either extend PagingAndSortingRepository that provides additional methods to ease paginated access to entities or write one of your own,

EmployeeRepository:

```
package com.javacodegeeks.spring.repositories;

import java.util.List;

import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.CrudRepository;

import com.javacodegeeks.spring.jpa.Employee;


public interface EmployeeRepository extends CrudRepository<Employee,String> {
    List<Employee> findEmployeesByAge(int age);
    List<Employee> findEmployeesByName(String name);
    @Query("select emp from Employee emp where emp.age >= ?1 and emp.age <= ?2")
    List<Employee> findEmployeesBetweenAge(int from, int to);
    Page<Employee> findEmployeesByAgeGreaterThan(int age, Pageable pageable);
}
```

11.8 CRUD Operations using Annotation Based Configuration

In our main class, we need to somehow scan the employee repository so that we have access to the CRUD operations. We will use @EnableJpaRepositories annotation to enable JPA repositories. We will have to provide the base package locations for scanning the package of the annotated configuration class for Spring Data repositories by default.

Using the employee repository we:

- Add employees
- Query the employees
- Accessing employees page wise

SpringDataJpaExampleUsingAnnotation:

```
package com.javacodegeeks.spring.jpa;

import java.net.URISyntaxException;
import java.sql.Connection;
import java.sql.SQLException;
import java.util.List;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import org.springframework.core.io.ClassPathResource;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort.Direction;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.jdbc.datasource.DataSourceUtils;
import org.springframework.jdbc.datasource.init.ResourceDatabasePopulator;
import org.springframework.transaction.annotation.Transactional;

import com.javacodegeeks.spring.repositories.EmployeeRepository;

@Configuration("mainBean")
@EnableJpaRepositories(basePackages = "com.javacodegeeks.spring.repositories")
@Import(JpaConfig.class)
@Transactional
public class SpringDataJpaExampleUsingAnnotation {
        @Autowired
        private EmployeeRepository repository;

        @Autowired
        private DataSource dataSource;

        public static void main(String[] args) throws URISyntaxException, Exception {
                AnnotationConfigApplicationContext ctx = new  ↩
                    AnnotationConfigApplicationContext();
                try {
                        ctx.register(SpringDataJpaExampleUsingAnnotation.class);
                        ctx.refresh();
                        System.out.println("Load context");
                        SpringDataJpaExampleUsingAnnotation s = ( ↩
                            SpringDataJpaExampleUsingAnnotation) ctx.getBean("mainBean");

                        System.out.println("Add employees");
                        s.addEmployees();
                        System.out.println("Find all employees");
                        s.findAllEmployees();
                        System.out.println("Find employee by name 'Joe'");
                        s.findEmployee("Joe");
                        System.out.println("Find employee by name 'John'");
                        s.findEmployee("John");
                        System.out.println("Find employees by age");
                        s.findEmployeesByAge(32);
                        System.out.println("Find employees between 30 and 45");
                        s.findEmployeesBetweenAge(30, 45);
                        System.out.println("Find employees greater than 20");
                        s.findEmployeesGreaterThanAgePageWise(20, 1, 0);
                        s.findEmployeesGreaterThanAgePageWise(20, 1, 1);
```

```
                        s.findEmployeesGreaterThanAgePageWise(20, 2, 0);
                        s.findEmployeesGreaterThanAgePageWise(20, 2, 1);
                } finally {
                        ctx.close();
                }
        }

        public void addEmployees() {
                Employee emp1 = new Employee("Richard", 32);
                Employee emp2 = new Employee("Satish", 30);
                Employee emp3 = new Employee("Priya", 16);
                Employee emp4 = new Employee("Rimi", 30);

                repository.save(emp1);
                repository.save(emp2);
                repository.save(emp3);
                repository.save(emp4);
        }

        public void findAllEmployees() {
                repository.findAll().forEach(System.out::println);
        }

        public void findEmployee(String name) {
                List<Employee> empList = repository.findEmployeesByName(name);
                System.out.println("Employee list: " + empList);
        }

        public void findEmployeesByAge(int age) {
                List<Employee> empList = repository.findEmployeesByAge(age);
                System.out.println("Employee list: " + empList);
        }

        public void findEmployeesBetweenAge(int from, int to) {
                List<Employee> empList = repository.findEmployeesBetweenAge(from, to);
                System.out.println("Employee list: " + empList);
        }

        public void findEmployeesGreaterThanAgePageWise(int age, int pageSize, int pageNbr) ↩
             {
                System.out.println("Page size: " + pageSize + ", page " + pageNbr);
                Pageable pageable = new PageRequest(pageNbr, pageSize, Direction.DESC, " ↩
                    name", "age");
                Page<Employee> page = repository.findEmployeesByAgeGreaterThan(age,  ↩
                    pageable);
                System.out.println(page.getContent());
        }
}
```

Output:

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See https://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Jan 27, 2016 9:22:12 AM org.hibernate.jpa.internal.util.LogHelper  ↩
    logPersistenceUnitInformation
INFO: HHH000204: Processing PersistenceUnitInfo [
        name: default
        ...]
Jan 27, 2016 9:22:12 AM org.hibernate.Version logVersion
INFO: HHH000412: Hibernate Core {4.3.10.Final}
Jan 27, 2016 9:22:12 AM org.hibernate.cfg.Environment
```

```
INFO: HHH000206: hibernate.properties not found
Jan 27, 2016 9:22:12 AM org.hibernate.cfg.Environment buildBytecodeProvider
INFO: HHH000021: Bytecode provider name : javassist
Jan 27, 2016 9:22:12 AM org.hibernate.annotations.common.reflection.java. ↩
    JavaReflectionManager
INFO: HCANN000001: Hibernate Commons Annotations {4.0.5.Final}
Jan 27, 2016 9:22:13 AM org.hibernate.dialect.Dialect
INFO: HHH000400: Using dialect: org.hibernate.dialect.HSQLDialect
Jan 27, 2016 9:22:13 AM org.hibernate.hql.internal.ast.ASTQueryTranslatorFactory
INFO: HHH000397: Using ASTQueryTranslatorFactory
Jan 27, 2016 9:22:13 AM org.hibernate.tool.hbm2ddl.SchemaUpdate execute
INFO: HHH000228: Running hbm2ddl schema update
Jan 27, 2016 9:22:13 AM org.hibernate.tool.hbm2ddl.SchemaUpdate execute
INFO: HHH000102: Fetching database metadata
Jan 27, 2016 9:22:13 AM org.hibernate.tool.hbm2ddl.SchemaUpdate execute
INFO: HHH000396: Updating schema
Jan 27, 2016 9:22:13 AM org.hibernate.tool.hbm2ddl.DatabaseMetadata getTableMetadata
INFO: HHH000262: Table not found: Employee
Jan 27, 2016 9:22:13 AM org.hibernate.tool.hbm2ddl.DatabaseMetadata getTableMetadata
INFO: HHH000262: Table not found: Employee
Jan 27, 2016 9:22:13 AM org.hibernate.tool.hbm2ddl.DatabaseMetadata getTableMetadata
INFO: HHH000262: Table not found: Employee
Jan 27, 2016 9:22:13 AM org.hibernate.tool.hbm2ddl.SchemaUpdate execute
INFO: HHH000232: Schema update complete
Load context
Add employees
Find all employees
Employee (1, Joe, 32)
Employee (2, Sam, 28)
Employee (3, John, 43)
Employee (4, Richard, 32)
Employee (5, Satish, 30)
Employee (6, Priya, 16)
Employee (7, Rimi, 30)
Find employee by name 'Joe'
Employee list: [Employee (1, Joe, 32)]
Find employee by name 'John'
Employee list: [Employee (3, John, 43)]
Find employees by age
Employee list: [Employee (1, Joe, 32), Employee (4, Richard, 32)]
Find employees between 30 and 45
Employee list: [Employee (1, Joe, 32), Employee (3, John, 43), Employee (4, Richard, 32),  ↩
    Employee (5, Satish, 30), Employee (7, Rimi, 30)]
Find employees greater than 20
Page size: 1, page 0
[Employee (5, Satish, 30)]
Page size: 1, page 1
[Employee (2, Sam, 28)]
Page size: 2, page 0
[Employee (5, Satish, 30), Employee (2, Sam, 28)]
Page size: 2, page 1
[Employee (7, Rimi, 30), Employee (4, Richard, 32)]
```

11.9 CRUD Operations using XML Configured Context

We will modify the context so that the repositories can be scanned. We take care of below points in the XML context.

- If you want to initialize a database and you can provide a reference to a `DataSource` bean, use the `initialize-database` tag in the `spring-jdbc` namespace.

- Repositories are scanned using <jpa:repositories>.
- In order to use <jpa:repositories> element, we need to add JPA namespace to the XML. We will use <context:annotation-config /> to point to the main configuration class.
- Define entityManagerFactory
- Define transactionManager

applicationContext.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="https://www.springframework.org/schema/beans"
        xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance" xmlns:jdbc="https://www. ↩
            springframework.org/schema/jdbc"
        xmlns:tx="https://www.springframework.org/schema/tx" xmlns:jpa="https://www. ↩
            springframework.org/schema/data/jpa"
        xmlns:context="https://www.springframework.org/schema/context"
        xsi:schemaLocation="https://www.springframework.org/schema/jdbc https://www. ↩
            springframework.org/schema/jdbc/spring-jdbc.xsd
                https://www.springframework.org/schema/beans https://www.springframework. ↩
                    org/schema/beans/spring-beans.xsd
                https://www.springframework.org/schema/data/jpa https://www.springframework ↩
                    .org/schema/data/jpa/spring-jpa.xsd
                https://www.springframework.org/schema/tx https://www.springframework.org/ ↩
                    schema/tx/spring-tx.xsd
                https://www.springframework.org/schema/context https://www.springframework. ↩
                    org/schema/context/spring-context.xsd">

        <jpa:repositories base-package="com.javacodegeeks.spring.repositories" />

        <bean id="transactionManager" class="org.springframework.orm.jpa. ↩
            JpaTransactionManager">
                <property name="entityManagerFactory" ref="entityManagerFactory" />
        </bean>

        <bean id="entityManagerFactory"
                class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
                <property name="dataSource" ref="dataSource" />
                <property name="packagesToScan" value="com.javacodegeeks.spring.jpa" />
                <property name="jpaVendorAdapter">
                        <bean class="org.springframework.orm.jpa.vendor. ↩
                            HibernateJpaVendorAdapter">
                                <property name="database" value="HSQL" />
                                <property name="generateDdl" value="true" />
                        </bean>
                </property>
        </bean>

        <jdbc:initialize-database data-source="dataSource">
                <jdbc:script location="classpath:data.sql" />
        </jdbc:initialize-database>

        <jdbc:embedded-database id="dataSource" type="HSQL" />

        <context:annotation-config />
        <bean id="mainBean"
                class="com.javacodegeeks.spring.jpa.SpringDataJpaExampleUsingXML" />

</beans>
```

Create the XML context using `ClassPathXmlApplicationContext`. Remaining code related to CRUD operations remain same.

```
ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("applicationContext ↩
    .xml");
```

SpringDataJpaExampleUsingXML:

```
package com.javacodegeeks.spring.jpa;

import java.net.URISyntaxException;
import java.sql.Connection;
import java.sql.SQLException;
import java.util.List;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.core.io.ClassPathResource;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort.Direction;
import org.springframework.jdbc.datasource.DataSourceUtils;
import org.springframework.jdbc.datasource.init.ResourceDatabasePopulator;
import org.springframework.stereotype.Repository;

import com.javacodegeeks.spring.repositories.EmployeeRepository;


@Configuration
@Repository
public class SpringDataJpaExampleUsingXML {
        @Autowired
        private EmployeeRepository repository;

        @Autowired
        private DataSource dataSource;

        public static void main(String[] args) throws URISyntaxException, Exception {
                ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext(" ↩
                    applicationContext.xml");
                try {
                        SpringDataJpaExampleUsingXML s = (SpringDataJpaExampleUsingXML) ctx ↩
                            .getBean("mainBean");
                        System.out.println("Add employees");
                        s.addEmployees();
                        System.out.println("Find all employees");
                        s.findAllEmployees();
                        System.out.println("Find employee by name 'Joe'");
                        s.findEmployee("Joe");
                        System.out.println("Find employee by name 'John'");
                        s.findEmployee("John");
                        System.out.println("Find employees by age");
                        s.findEmployeesByAge(32);
                        System.out.println("Find employees between 30 and 45");
                        s.findEmployeesBetweenAge(30, 45);
                        System.out.println("Find employees greater than 20");
                        s.findEmployeesGreaterThanAgePageWise(20, 1, 0);
                        s.findEmployeesGreaterThanAgePageWise(20, 1, 1);
                        s.findEmployeesGreaterThanAgePageWise(20, 2, 0);
                        s.findEmployeesGreaterThanAgePageWise(20, 2, 1);
```

```
                } finally {
                        ctx.close();
                }
        }

        public void addEmployees() {
                Employee emp1 = new Employee("Richard", 32);
                Employee emp2 = new Employee("Satish", 30);
                Employee emp3 = new Employee("Priya", 16);
                Employee emp4 = new Employee("Rimi", 30);

                repository.save(emp1);
                repository.save(emp2);
                repository.save(emp3);
                repository.save(emp4);
        }

        public void findAllEmployees() {
                repository.findAll().forEach(System.out::println);
        }

        public void findEmployee(String name) {
                List<Employee> empList = repository.findEmployeesByName(name);
                System.out.println("Employee list: " + empList);
        }

        public void findEmployeesByAge(int age) {
                List<Employee> empList = repository.findEmployeesByAge(age);
                System.out.println("Employee list: " + empList);
        }

        public void findEmployeesBetweenAge(int from, int to) {
                List<Employee> empList = repository.findEmployeesBetweenAge(from, to);
                System.out.println("Employee list: " + empList);
        }

        public void populateDatabase() throws SQLException {

                ResourceDatabasePopulator populator = new ResourceDatabasePopulator();
                populator.addScript(new ClassPathResource("data.sql"));

                Connection connection = null;

                try {
                        connection = DataSourceUtils.getConnection(dataSource);
                        populator.populate(connection);
                } finally {
                        if (connection != null) {
                                DataSourceUtils.releaseConnection(connection, dataSource);
                        }
                }
        }

        public void findEmployeesGreaterThanAgePageWise(int age, int pageSize, int pageNbr) ↩
             {
                System.out.println("Page size: " + pageSize + ", page " + pageNbr);
                Pageable pageable = new PageRequest(pageNbr, pageSize, Direction.DESC, " ↩
                    name", "age");
                Page<Employee> page = repository.findEmployeesByAgeGreaterThan(age,  ↩
                    pageable);
                System.out.println(page.getContent());
        }
```

```
}
```

Output:

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See https://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Jan 27, 2016 9:44:21 AM org.hibernate.jpa.internal.util.LogHelper ↩
    logPersistenceUnitInformation
INFO: HHH000204: Processing PersistenceUnitInfo [
        name: default
        ...]
Jan 27, 2016 9:44:21 AM org.hibernate.Version logVersion
INFO: HHH000412: Hibernate Core {4.3.10.Final}
Jan 27, 2016 9:44:21 AM org.hibernate.cfg.Environment
INFO: HHH000206: hibernate.properties not found
Jan 27, 2016 9:44:21 AM org.hibernate.cfg.Environment buildBytecodeProvider
INFO: HHH000021: Bytecode provider name : javassist
Jan 27, 2016 9:44:21 AM org.hibernate.annotations.common.reflection.java. ↩
    JavaReflectionManager
INFO: HCANN000001: Hibernate Commons Annotations {4.0.5.Final}
Jan 27, 2016 9:44:21 AM org.hibernate.dialect.Dialect
INFO: HHH000400: Using dialect: org.hibernate.dialect.HSQLDialect
Jan 27, 2016 9:44:22 AM org.hibernate.hql.internal.ast.ASTQueryTranslatorFactory
INFO: HHH000397: Using ASTQueryTranslatorFactory
Jan 27, 2016 9:44:22 AM org.hibernate.tool.hbm2ddl.SchemaUpdate execute
INFO: HHH000228: Running hbm2ddl schema update
Jan 27, 2016 9:44:22 AM org.hibernate.tool.hbm2ddl.SchemaUpdate execute
INFO: HHH000102: Fetching database metadata
Jan 27, 2016 9:44:22 AM org.hibernate.tool.hbm2ddl.SchemaUpdate execute
INFO: HHH000396: Updating schema
Jan 27, 2016 9:44:22 AM org.hibernate.tool.hbm2ddl.DatabaseMetadata getTableMetadata
INFO: HHH000262: Table not found: Employee
Jan 27, 2016 9:44:22 AM org.hibernate.tool.hbm2ddl.DatabaseMetadata getTableMetadata
INFO: HHH000262: Table not found: Employee
Jan 27, 2016 9:44:22 AM org.hibernate.tool.hbm2ddl.DatabaseMetadata getTableMetadata
INFO: HHH000262: Table not found: Employee
Jan 27, 2016 9:44:22 AM org.hibernate.tool.hbm2ddl.SchemaUpdate execute
INFO: HHH000232: Schema update complete
Add employees
Find all employees
Employee (1, Joe, 32)
Employee (2, Sam, 28)
Employee (3, John, 43)
Employee (4, Richard, 32)
Employee (5, Satish, 30)
Employee (6, Priya, 16)
Employee (7, Rimi, 30)
Find employee by name 'Joe'
Employee list: [Employee (1, Joe, 32)]
Find employee by name 'John'
Employee list: [Employee (3, John, 43)]
Find employees by age
Employee list: [Employee (1, Joe, 32), Employee (4, Richard, 32)]
Find employees between 30 and 45
Employee list: [Employee (1, Joe, 32), Employee (3, John, 43), Employee (4, Richard, 32), ↩
    Employee (5, Satish, 30), Employee (7, Rimi, 30)]
Find employees greater than 20
Page size: 1, page 0
[Employee (5, Satish, 30)]
Page size: 1, page 1
[Employee (2, Sam, 28)]
```

```
Page size: 2, page 0
[Employee (5, Satish, 30), Employee (2, Sam, 28)]
Page size: 2, page 1
[Employee (7, Rimi, 30), Employee (4, Richard, 32)]
```

11.10 Download the Eclipse Project

This was an example about Spring Data Jpa.

Download You can download the full source code of this example here: **springDataJpaExample.zip**

Chapter 12

Couchbase Example

In this example we shall demonstrate how we can connect Spring Data with Couchbase.

12.1 What is CouchBase?

Couchbase is a highly scalable, Document based NoSQL Database. Document based NoSQL databases work on map-like concept of `KEY-VALUE` pairs. The key being uniquely identifiable property like a String, path etc and the value being the Document that is to be saved. Another example of Document based NoSQL is MongoDB. In one of our previous examples, we have already demonstrated how we can connect and manage Spring Data with MongoDB.

Spring Data CouchBase is the Spring module which helps us in integrating with the CouchBase Database Server.As with the other modules demonstrated in this series, this module too provides supports both for derived queries(based on the method names) and the annotated query.

12.2 Project Set-Up

So, let's start the project setup by installing CouchBase Database Server from here. We have used the Community Edition for this example.

Upon successful installation the user will be directed to this page : https://localhost:8091/index.html

Here is how the page looks like:

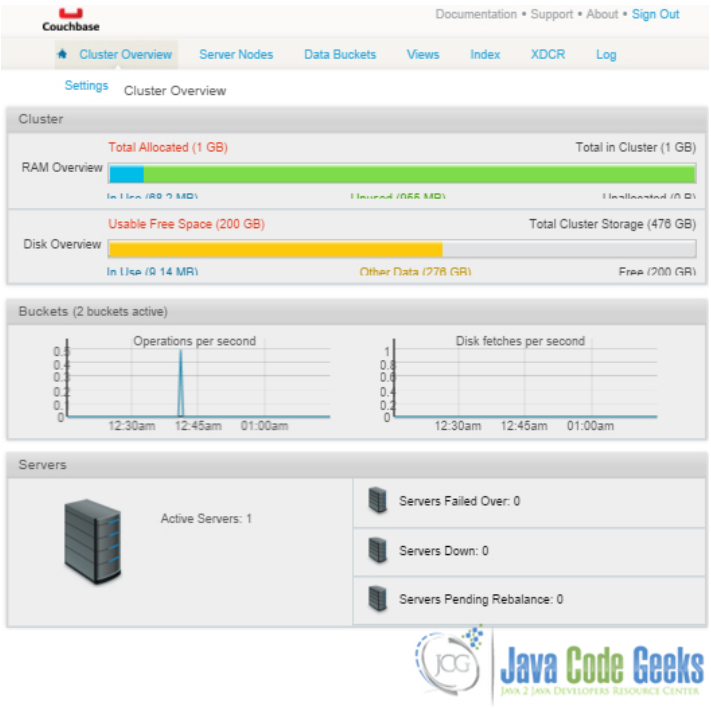


Figure 12.1: Successful Couchbase Setup

Next step is to create a new `Data-Bucket`. `Data-Bucket` is like analogous to a `Table` in `RDBMS` or to a `Collection` in a `MongoDb` Database. This can done by clicking on the `Create New Data Bucket` button present on the `Data Buckets` tab. We are naming it `JavaCodeGeeks` for the sake of this example. We shall be adding our documents to this data bucket.

Now that the CouchBase server is up and running, we will setup the application environment. Create a simple Maven Project in `Eclipse IDE` by selecting the `Skip Archetype Selection` checkbox from the New Maven Project Pop-up. We are using the below `pom.xml` to manage the dependencies for CouchBase from Spring Data.

pom.xml

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↩
    XMLSchema-instance"
        xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd ↩
            /maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <groupId>com.jcg.examples.springDataCouchbaseExample</groupId>
        <artifactId>SpringDataCouchbaseExample</artifactId>
        <version>0.0.1-SNAPSHOT</version>

        <dependencies>
                <dependency>
                        <groupId>org.springframework.data</groupId>
                        <artifactId>spring-data-couchbase</artifactId>
                        <version>2.0.0.RELEASE</version>
                </dependency>

        </dependencies>

</project>
```

Eclipse will download the required JAR files and add the dependencies in the project classpath. Now that the project is setup and dependencies imported, we can begin writing the actual code.

12.3 Implementation

The implementation will consist of three major configuration files. The first one is the actual domain object which will be persisted in the CouchBase Database as the `Document`.

Book.java

```
package com.jcg.examples.entity;

import org.springframework.data.couchbase.core.mapping.Document;

import com.couchbase.client.java.repository.annotation.Field;
import com.couchbase.client.java.repository.annotation.Id;

@Document(expiry=0)
public class Book
{

                @Id
                private long bookId;

                @Field
                private String name;

                @Field
                private long isbnNumber;

                public long getBookId()
                {
                                return bookId;
                }

                public void setBookId(long bookId)
                {
                                this.bookId = bookId;
                }

                public String getName()
                {
                                return name;
                }

                public void setName(String name)
                {
                                this.name = name;
                }

                public long getIsbnNumber()
                {
                                return isbnNumber;
                }

                public void setIsbnNumber(long isbnNumber)
                {
                                this.isbnNumber = isbnNumber;
                }

                @Override
                public String toString()
                {
                                return "Book [bookId=" + bookId + ", name=" + name + ",  ↩
                                    isbnNumber=" + isbnNumber + "]";
```

```
                }

}
```

The `@Document` annotation is used to mark the PoJo as the `Couchbase Document`. It has an `expiry` attribute which is the TTL of the document.

`@Id` marks the corresponding instance variable as document Id in the database.We shall retrieve the document later based on this Id.

Next is the basic Repository Class:

BookRepo.java

```
package com.jcg.examples.repo;

import org.springframework.data.couchbase.repository.CouchbaseRepository;
import org.springframework.stereotype.Repository;

import com.jcg.examples.entity.Book;

@Repository
public interface BookRepo extends CouchbaseRepository<Book, Long>
{
}
```

The repo simply extends from the `org.springframework.data.couchbase.repository.CouchbaseRepository` interface and the implementation is provided by the Spring container at the run-time. We need to provide declare the proper Generics as per the Entity, which in our case is the . Spring Data Module provides us with a number of inbuilt method for manipulating the Data. We need not write the queries for basic data manipulation and reading.

For executing custom written queries, the developer can create his own method by specifying the Query using the `org.springframework.data.couchbase.core.query.Query` annotation.

Lastly, and most important is the `XML` based configuration to initiate the Couchbase datasource so that we can communicate and execute our queries against the Server.

spring-couchbase-integration.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns:beans="https://www.springframework.org/schema/beans"
  xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
  xmlns:couchbase="https://www.springframework.org/schema/data/couchbase"
  xsi:schemaLocation="https://www.springframework.org/schema/beans
    https://www.springframework.org/schema/beans/spring-beans-4.1.xsd
    https://www.springframework.org/schema/data/couchbase
    https://www.springframework.org/schema/data/couchbase/spring-couchbase.xsd">

    <couchbase:cluster>
      <couchbase:node>127.0.0.1</couchbase:node>
    </couchbase:cluster>

    <couchbase:clusterInfo  login="Administrator" password="Administrator"/>

        <beans:bean id="couchbaseEnv" class="com.couchbase.client.java.env. ↩
            DefaultCouchbaseEnvironment" factory-method="create"/>
        <beans:bean id="myCustomTranslationService" class="org.springframework.data. ↩
            couchbase.core.convert.translation.JacksonTranslationService"/>

        <couchbase:indexManager/>

        <couchbase:repositories base-package="com.jcg.examples.repo" />

         <couchbase:template translation-service-ref="myCustomTranslationService"/>
```

```
    <couchbase:bucket bucketName="JavaCodeGeeks" bucketPassword="password.1"/>
</beans:beans>
```

That is all from the setup point of view. Now that all is set, let's run the application and test out the code! Here's `ApplicationTest` class that loads the `XML` file to instantiate the Spring Container and execute a few queries.

ApplicationTest.java

```
package com.jcg.examples;

import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.core.io.ClassPathResource;

import com.jcg.examples.entity.Book;
import com.jcg.examples.repo.BookRepo;

public class ApplicationTest
{


                public static void main(String[] args)
                {
                                ClassPathXmlApplicationContext context = new  ↩
                                    ClassPathXmlApplicationContext(new ClassPathResource(" ↩
                                    spring-couchbase-integration.xml").getPath());
                                BookRepo bookRepo = context.getBean(BookRepo.class);

                                Book book = new Book();
                                book.setBookId(123);;
                                book.setIsbnNumber(2133443554);
                                book.setName("Kane And Abel by XYZ");
                                bookRepo.save(book);

                                Book anotherBook = new Book();
                                anotherBook.setBookId(456);;
                                anotherBook.setIsbnNumber(2133443554);
                                anotherBook.setName("The Prodigal Daughter");
                                bookRepo.save(anotherBook);

                                Book retreivedBook = bookRepo.findOne(123l);

                                System.out.println(retreivedBook);

                                bookRepo.delete(456l);

                                context.close();
                }

}
```

We have created two `Book` objects and persisted them as documents in the Database. Then we will try to retrieve them by their Id and delete one of them by passing their ID.

Here's the sample output of the program:

```
INFO: Connected to Node 127.0.0.1
Feb 28, 2016 12:41:27 AM com.couchbase.client.core.config.DefaultConfigurationProvider$8  ↩
    call
INFO: Opened bucket JavaCodeGeeks
Book [bookId=123, name=Kane And Abel by XYZ, isbnNumber=2133443554]
```

In the UI console, the user can see the save Documents under the Data-Buckets>Documents Tab. Here's what it looks like :

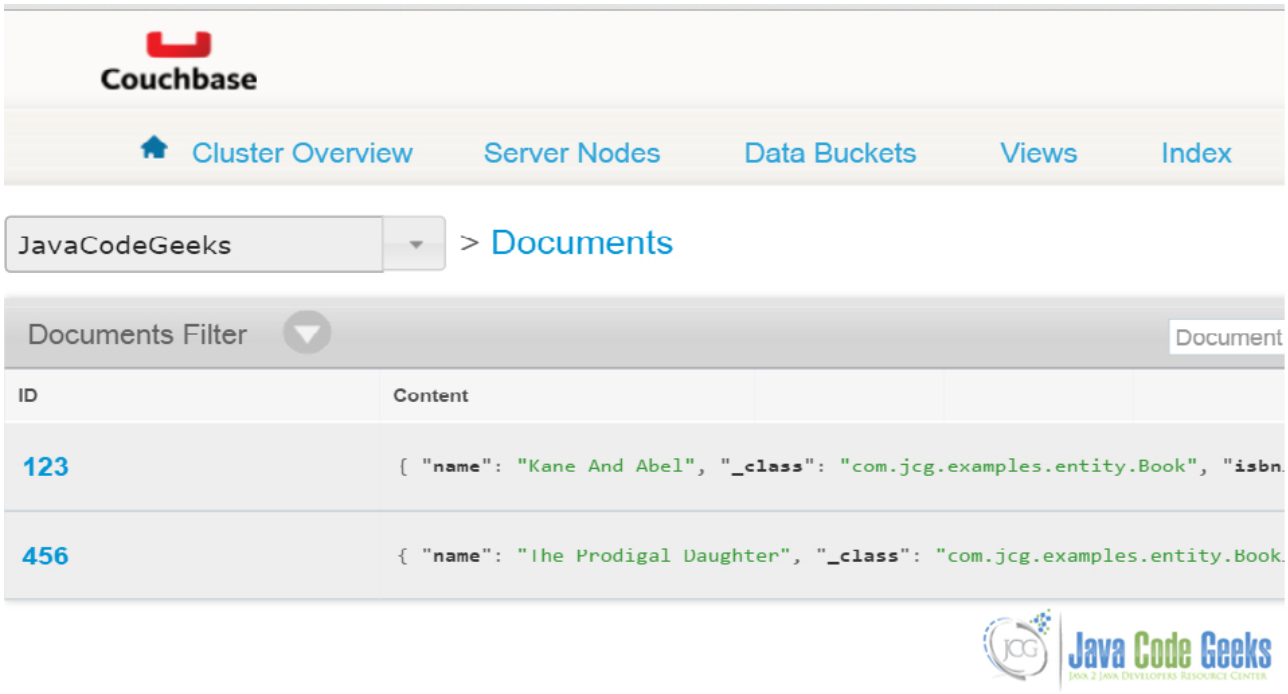


Figure 12.2: CouchBase Save Example

And the detailed Document View:

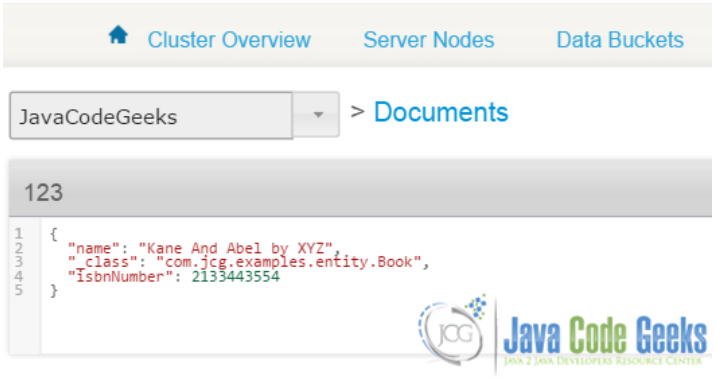


Figure 12.3: Saved Document details

12.4 Download the Eclipse Project

Here we demonstrated how to configure and manage a Couchbase Database Server using Spring Data.

Download You can download the source code of this example here: **SpringDataCouchbaseExample.zip**