Spring Data

Amit Kumar

Spring Data

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.

Features

- Powerful repository and custom object-mapping abstractions
- Dynamic query derivation from repository method names
- Implementation domain base classes providing basic properties
- Support for transparent auditing (created, last changed)
- Possibility to integrate custom repository code
- Easy Spring integration via JavaConfig and custom XML namespaces
- Advanced integration with Spring MVC controllers
- Experimental support for cross-store persistence

Spring Data

Main modules

- Spring Data Commons - Core Spring concepts underpinning every Spring Data module.
- Spring Data JDBC - Spring Data repository support for JDBC.
- Spring Data JDBC Ext - Support for database specific extensions to standard JDBC including support for Oracle RAC fast connection failover, AQ JMS support and support for using advanced data types.
- Spring Data JPA - Spring Data repository support for JPA.
- Spring Data KeyValue - Map based repositories and SPIs to easily build a Spring Data module for key-value stores.
- Spring Data LDAP - Spring Data repository support for Spring LDAP.
- Spring Data MongoDB - Spring based, object-document support and repositories for MongoDB.
- Spring Data Redis - Easy configuration and access to Redis from Spring applications.
- Spring Data REST - Exports Spring Data repositories as hypermedia-driven RESTful resources.
- Spring Data for Apache Cassandra - Easy configuration and access to Apache Cassandra or large scale, highly available, data oriented Spring applications.
- Spring Data for Apache Geode - Easy configuration and access to Apache Geode for highly consistent, low latency, data oriented Spring applications.
- Spring Data for Pivotal GemFire - Easy configuration and access to Pivotal GemFire for your highly consistent, low latency/high through-put, data-oriented Spring applications.

Spring Data

Community modules

- Spring Data Aerospike - Spring Data module for Aerospike.
- Spring Data ArangoDB - Spring Data module for ArangoDB.
- Spring Data Couchbase - Spring Data module for Couchbase.
- Spring Data Azure Cosmos DB - Spring Data module for Microsoft Azure Cosmos DB.
- Spring Data Cloud Datastore - Spring Data module for Google Datastore.
- Spring Data Cloud Spanner - Spring Data module for Google Spanner.
- Spring Data DynamoDB - Spring Data module for DynamoDB.
- Spring Data Elasticsearch - Spring Data module for Elasticsearch.
- Spring Data Hazelcast - Provides Spring Data repository support for Hazelcast.
- Spring Data Jest - Spring Data module for Elasticsearch based on the Jest REST client.
- Spring Data Neo4j - Spring-based, object-graph support and repositories for Neo4j.
- Oracle NoSQL Database SDK for Spring Data - Spring Data module for Oracle NoSQL Database and Oracle NoSQL Cloud Service.
- Spring Data for Apache Solr - Easy configuration and access to Apache Solr for your search-oriented Spring applications.
- Spring Data Vault - Vault repositories built on top of Spring Data KeyValue.
- Spring Data YugabyteDB - Spring Data module for YugabyteDB distributed SQL database.

Spring Data

Release train

Spring Data is an umbrella project consisting of independent projects with, in principle, different release cadences. To manage the portfolio, a BOM (Bill of Materials - see this example) is published with a curated set of dependencies on the individual project. The release trains have names, not versions, to avoid confusion with the sub-projects.

- Spring Data Commons
- Spring Data JPA
- Spring Data KeyValue
- Spring Data LDAP
- Spring Data MongoDB
- Spring Data Redis
- Spring Data REST
- Spring Data for Apache Cassandra
- Spring Data for Apache Geode
- Spring Data for Apache Solr
- Spring Data for Pivotal GemFire
- Spring Data Couchbase (community module)
- Spring Data Elasticsearch (community module)
- Spring Data Neo4j (community module)

Spring JDBC

Spring JdbcTemplate is a powerful mechanism to connect to the database and execute SQL queries. It internally uses JDBC api, but eliminates a lot of problems of JDBC API.

The problems of JDBC API are as follows:

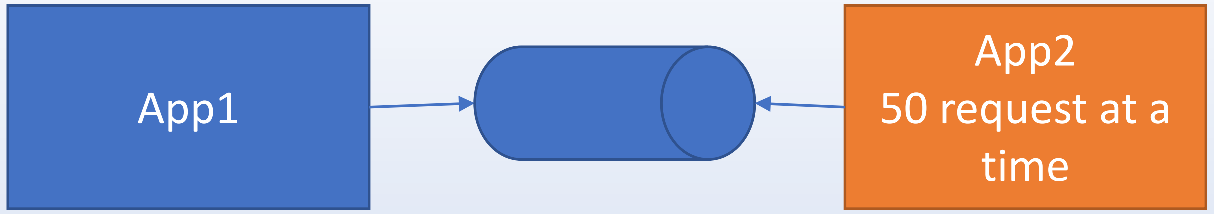
- We need to write a lot of code before and after executing the query, such as creating connection, statement, closing resultset, connection etc.
- We need to perform exception handling code on the database logic.
- We need to handle transaction.
- Repetition of all these codes from one to another database logic is a time consuming task.

Spring JDBC

Spring JDBC are divided into four separate packages:

- ***core*** — the core functionality of JDBC. Some of the important classes under this package include *JdbcTemplate*, *SimpleJdbcInsert*, *SimpleJdbcCall* and *NamedParameterJdbcTemplate*.
- ***datasource*** — utility classes to access a data source. It also has various data source implementations for testing JDBC code outside the Jakarta EE container.
- ***object*** — DB access in an object-oriented manner. It allows running queries and returning the results as a business object. It also maps the query results between the columns and properties of business objects.
- ***support*** — support classes for classes under *core* and *object* packages, e.g., provides the *SQLException* translation functionality

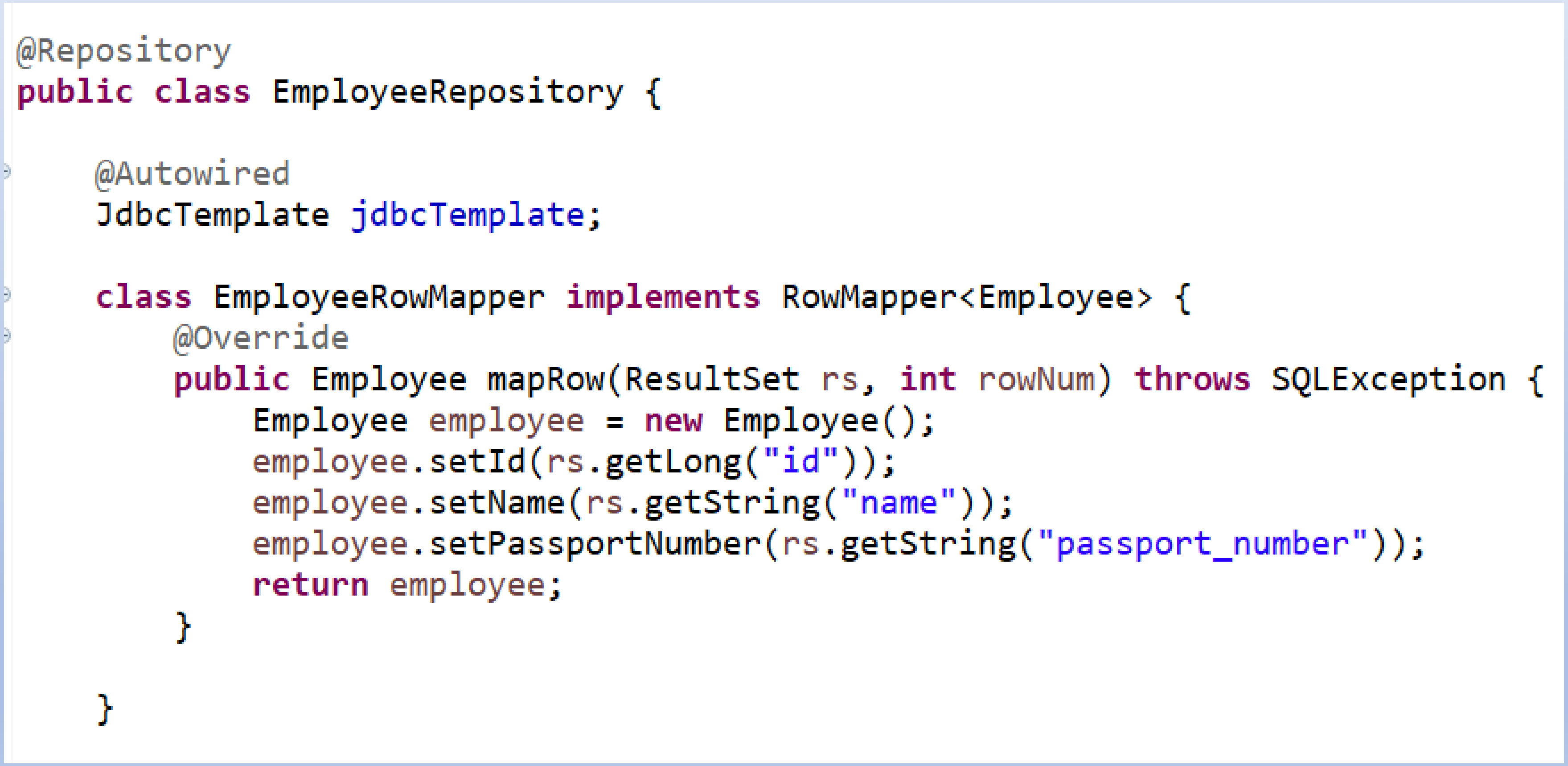
JdbcTemplate



- It is the central class in the Spring JDBC support classes. It takes care of creation and release of resources such as creating and closing of connection object etc. So it will not lead to any problem if you forget to close the connection.
- It handles the exception and provides the informative exception messages by the help of exception classes defined in the **org.springframework.dao** package.
- We can perform all the database operations by the help of JdbcTemplate class such as insertion, updating, deletion and retrieval of the data from the database.

No.	Method	Description
1)	public int update(String query)	is used to insert, update and delete records.
2)	public int update(String query,Object... args)	is used to insert, update and delete records using PreparedStatement using given arguments.
3)	public void execute(String query)	is used to execute DDL query.
4)	public T execute(String sql, PreparedStatementCallback action)	executes the query by using PreparedStatement callback.
5)	public T query(String sql, ResultSetExtractor rse)	is used to fetch records using ResultSetExtractor.
6)	public List query(String sql, RowMapper rse)	is used to fetch records using RowMapper.

JdbcTemplate



```
@Repository
public class EmployeeRepository {

    @Autowired
    JdbcTemplate jdbcTemplate;

    class EmployeeRowMapper implements RowMapper<Employee> {
        @Override
        public Employee mapRow(ResultSet rs, int rowNum) throws SQLException {
            Employee employee = new Employee();
            employee.setId(rs.getLong("id"));
            employee.setName(rs.getString("name"));
            employee.setPassportNumber(rs.getString("passport_number"));
            return employee;
        }

    }
```

Stored Procedures With SimpleJdbcCall

```
SimpleJdbcCall simpleJdbcCall = new SimpleJdbcCall(dataSource)
.withProcedureName(“GET_EMPLOYEE”);

public Employee getEmployeeUsingSimpleJdbcCall(int id) {

    SqlParameterSource in = new MapSqlParameterSource().addValue("in_id",
id);

    Map<String, Object> out = simpleJdbcCall.execute(in);

    Employee emp = new Employee();

    emp.setFirstName((String) out.get("FIRST_NAME"));

    emp.setLastName((String) out.get("LAST_NAME"));

    return emp;

}
```

Spring JDBC

Spring JDBC Configuration (MYSQL):

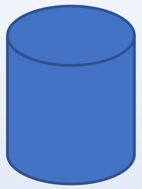
```
@Configuration
@ComponentScan("com.amit.jdbc")
public class SpringJdbcConfig {
    @Bean
    public DataSource mysqlDataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/sample");
        dataSource.setUsername(“root");
        dataSource.setPassword(“root");
        return dataSource;
    }
}
```

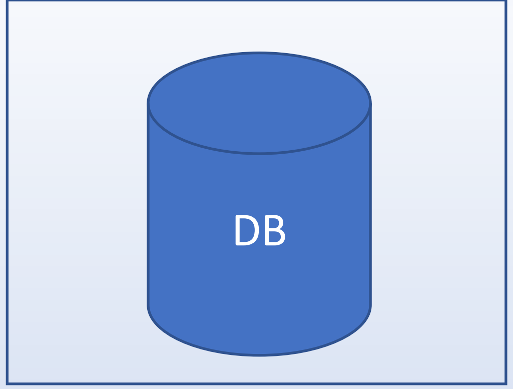
Spring JDBC with Spring boot

Spring boot JDBC Configuration (MYSQL):

```
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=root
```

Spring boot H2 DB





The **H2** in-memory database is volatile, and data will be lost when we restart the application. H2 is an open source database and is written in Java. It is very fast and of very small size. It is primarily used as an in-memory database which means it stores the data in memory and will not persist data on disk. Although if we need to persist the data, it supports that as well.

The H2 database is not recommended for production environments and is ideal for a quick POC kind of project where there is a need for a simple database. We can change that behavior by using file-based storage. To do this we need to update the spring.datasource.url:

```
spring.datasource.url=jdbc:h2:file:/data/sample
```

Spring boot Derby

Apache Derby, an Apache DB subproject, is an open source relational database implemented entirely in Java and available under the Apache License, Version 2.0. Some key advantages include:

- Derby has a small footprint -- about 3.5 megabytes for the base engine and embedded JDBC driver.
- Derby is based on the Java, JDBC, and SQL standards.
- Derby provides an embedded JDBC driver that lets you embed Derby in any Java-based solution.
- Derby also supports the more familiar client/server mode with the Derby Network Client JDBC driver and Derby Network Server.
- Derby is easy to install, deploy, and use.

Official website - https://db.apache.org/derby/

Spring Boot has very good integration for Derby. It's very easy to use Derby database in spring boot applications by just adding the Derby dependency.

Spring boot HSQL

HSQLDB (HyperSQL DataBase) is the leading SQL relational database software written in Java. It offers a small, fast multithreaded and transactional database engine with in-memory and disk-based tables and supports embedded and server modes. It includes a powerful command line SQL tool and simple GUI query tools.

Official website - http://hsqldb.org/

Spring Boot has very good integration for HSQLDB. It's very easy to use HSQLDB database in spring boot applications by just adding the HSQLDB dependency

Spring boot MySQL

MySQL is the most popular Open Source Relational SQL Database Management System. MySQL is one of the best RDBMS being used for developing various web-based software applications.

- MySQL is released under an open-source license. So you have nothing to pay to use it.
- MySQL is a very powerful program in its own right. It handles a large subset of the functionality of the most expensive and powerful database packages.
- MySQL uses a standard form of the well-known SQL data language.
- MySQL works on many operating systems and with many languages including PHP, PERL, C, C++, JAVA, etc.
- MySQL supports large databases, up to 50 million rows or more in a table. The default file size limit for a table is 4GB, but you can increase this (if your operating system can handle it) to a theoretical limit of 8 million terabytes (TB).

Spring boot Postgress

- PostgreSQL, originally called Postgres, was created at UCB by a computer science professor named Michael Stonebraker. Stonebraker started Postgres in 1986 as a follow-up project to its predecessor, Ingres, now owned by Computer Associates.
- **1977-1985** − A project called INGRES was developed.
  - Proof-of-concept for relational databases
  - Established the company Ingres in 1980
  - Bought by Computer Associates in 1994
- **1986-1994** − POSTGRES
  - Development of the concepts in INGRES with a focus on object orientation and the query language - Quel
  - The code base of INGRES was not used as a basis for POSTGRES
  - Commercialized as Illustra (bought by Informix, bought by IBM)
- **1994-1995** − Postgres95
  - Support for SQL was added in 1994
  - Released as Postgres95 in 1995
  - Re-released as PostgreSQL 6.0 in 1996
  - Establishment of the PostgreSQL Global Development Team

Spring boot Postgress

Key Features of PostgreSQL:

- PostgreSQL runs on all major operating systems, including Linux, UNIX (AIX, BSD, HP-UX, SGI IRIX, Mac OS X, Solaris, Tru64), and Windows. It supports text, images, sounds, and video, and includes programming interfaces for C / C++, Java, Perl, Python, Ruby, Tcl and Open Database Connectivity (ODBC).
- PostgreSQL supports a large part of the SQL standard and offers many modern features including the following
- Complex SQL queries
- SQL Sub-selects
- Foreign keys
- Trigger
- Views
- Transactions
- Multiversion concurrency control (MVCC)

Spring boot Postgress

Key Features of PostgreSQL:

- Streaming Replication (as of 9.0)
- Hot Standby (as of 9.0)
- You can check official documentation of PostgreSQL to understand the above-mentioned features. PostgreSQL can be extended by the user in many ways. For example by adding new −
- Data types
- Functions
- Operators
- Aggregate functions
- Index methods
- Procedural Languages Support
- PostgreSQL supports four standard procedural languages, which allows the users to write their own code in any of the languages and it can be executed by PostgreSQL database server. These procedural languages are - PL/pgSQL, PL/Tcl, PL/Perl and PL/Python. Besides, other non-standard procedural languages like PL/PHP, PL/V8, PL/Ruby, PL/Java, etc., are also supported.

Spring data JPA

Spring Data JPA, part of the larger Spring Data family, makes it easy to easily implement JPA based repositories. This module deals with enhanced support for JPA based data access layers. It makes it easier to build Spring-powered applications that use data access technologies.

Implementing a data access layer of an application has been cumbersome for quite a while. Too much boilerplate code has to be written to execute simple queries as well as perform pagination, and auditing. Spring Data JPA aims to significantly improve the implementation of data access layers by reducing the effort to the amount that’s actually needed. As a developer you write your repository interfaces, including custom finder methods, and Spring will provide the implementation automatically.

Spring data JPA

Features:

- Sophisticated support to build repositories based on Spring and JPA
- Support for Querydsl predicates and thus type-safe JPA queries
- Transparent auditing of domain class
- Pagination support, dynamic query execution, ability to integrate custom data access code
- Validation of @Query annotated queries at bootstrap time
- Support for XML based entity mapping
- JavaConfig based repository configuration by introducing @EnableJpaRepositories.

Spring data Repositories

- ***CrudRepository*** provides CRUD functions
- ***PagingAndSortingRepository*** provides methods to do pagination and sort records
- ***JpaRepository*** provides JPA related methods such as flushing the persistence context and delete records in a batch

- *JpaRepository* – which extends *PagingAndSortingRepository* and, in turn, the *CrudRepository*.

Spring data Repositories

- **CrudRepository**: This interface looks quite generic and simple, but actually, it provides all basic query abstractions needed in an application.
- save(…) – save an Iterable of entities. Here, we can pass multiple objects to save them in a batch
- findOne(…) – get a single entity based on passed primary key value
- findAll() – get an Iterable of all available entities in database
- count() – return the count of total entities in a table
- delete(…) – delete an entity based on the passed object
- exists(…) – verify if an entity exists based on the passed primary key value

Spring data Repositories

- **CrudRepository:** This interface looks quite generic and simple, but actually, it provides all basic query abstractions needed in an application.

```
public interface CrudRepository<T, ID extends Serializable>
  extends Repository<T, ID> {
    <S extends T> S save(S entity);
    T findOne(ID primaryKey);
    Iterable<T> findAll();
    Long count();
    void delete(T entity);
    boolean exists(ID primaryKey);
}
```

Spring data Repositories

- **PagingAndSortingRepository:** This interface provides a method findAll(Pageable pageable), which is the key to implementing Pagination.

When using Pageable, we create a Pageable object with certain properties and we've to specify at least:

1. Page size
2. Current page number
3. Sorting

Spring data Repositories

- **PagingAndSortingRepository**:

So, let's assume that we want to show the first page of a result set sorted by lastName, ascending, having no more than five records each. This is how we can achieve this using a PageRequest and a Sort definition:

```
Sort sort = new Sort(new Sort.Order(Direction.ASC, "lastName"));
Pageable pageable = new PageRequest(0, 5, sort);
```

Passing the pageable object to the Spring data query will return the results in question (the first parameter of PageRequest is zero-based).

Spring data Repositories

- **PagingAndSortingRepository**:

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

    Iterable<T> findAll(Sort sort);

    Page<T> findAll(Pageable pageable);
}
```

Spring data Repositories

- **JPARepository:** contains the full API of CrudRepository and PagingAndSortingRepository
- *findAll() – get a List of all available entities in database*
- *findAll(…) – get a List of all available entities and sort them using the provided condition*
- *save(…) – save an Iterable of entities. Here, we can pass multiple objects to save them in a batch*
- *flush() – flush all pending task to the database*
- *saveAndFlush(…) – save the entity and flush changes immediately*
- *deleteInBatch(…) – delete an Iterable of entities. Here, we can pass multiple objects to delete them in a batch*

Clearly, above interface extends PagingAndSortingRepository which means it has all methods present in the CrudRepository as well.

Spring data Repositories

- **JPARepository:** contains the full API of CrudRepository and PagingAndSortingRepository

```
public interface JpaRepository<T, ID extends Serializable> extends
  PagingAndSortingRepository<T, ID> {
    List<T> findAll();
    List<T> findAll(Sort sort);
    List<T> save(Iterable<? extends T> entities);
    void flush();
    T saveAndFlush(T entity);
    void deleteInBatch(Iterable<T> entities);
}
```

Using Spring Data Repositories

```
@Repository
public interface EmployeesRepository extends CrudRepository<Employee, Long> {
    Employee findDistinctFirstByAge(int age);
}
```

Spring knows the Employee is the entity and Long is the type of primary key.

EmployeesRepository inherits all the methods from CrudRepository.

Additionally, it defines a custom query method findDistinctFirstByAge.

@Repository annotates it as a Repository.

Spring provides proxy implementation of EmployeesRepository and all of its methods including findDistinctFirstByAge.

JPA Query Concepts

- Spring derives SQL queries from the method names and also see how to write more complex queries in the form of query methods. To parse the query method names into the actual queries, Spring uses certain strategy. If we follow these strategies we can easily write the query methods.

JPA Query Concepts

- **Retrieve Entities:** query methods will be all about retrieving or finding certain entities from the table. Spring lets us starting the query method names by keywords like find..By, get...By, read..By, count..By, and query..By.
- All of the patterns except count..By are aliases of each other. All of the methods below will behave similarly:
- Employee findById(Long id);
- Employee readById(Long id);
- Employee getById(Long id);
- Employee queryById(Long id);
- **Integer countByName(String name);**

JPA Query Concepts

- **Find By Multiple Fields:** combination of more than one field or condition
- All of the patterns except count..By are aliases of each other. All of the methods below will behave similarly:
- `List<Employee> findByAgeAndHeight(Integer age, double height);`
- `List<Employee> findByAgeAndNameAndDept(Integer age, String name, String dept);`
- `List<Employee> findByNameOrAge(String name, Integer age);`
- `List<Employee> findByNameIgnoreCaseAndAge(String name, String age);`

JPA Query Concepts

- **Limiting Results:** limit the number of records
- All of the patterns except count..By are aliases of each other. All of the methods below will behave similarly:
- Employee findFirstByName(String name);
- Employee findTopByName(String name);
- List<Employee> findTop10ByAge(String Age);
- Employee findTopByOrderByBirthDateDesc();

JPA Named Queries

```
@Entity

@NamedQuery(name = "User.findByEmailAddress",

  query = "select u from User u where u.emailAddress = ?1")

public class User {

}
```

@Query

- Queries annotated to the query method take precedence over queries defined using @Query

```
public interface UserRepository extends JpaRepository<User, Long> {

  @Query("select u from User u where u.emailAddress = ?1")

  User findByEmailAddress(String emailAddress);

}
```

@Async

- if we want that our query method is executed asynchronously, we have to annotate it with the @Async annotation and return a Future<T> object. Here are some examples of query methods that are executed asynchronously:

```
interface EmployeeRepository extends Repository<Employee, Long> {
    @Async
    @Query("SELECT t.name FROM Employee t where t.id = :id")
    Future<String> findNameById(@Param("id") Long id);
     @Async
    Future<Employee> findById(Long id);
}
```

Spring Data MongoDB

Spring Data provides additional projects that help you access a variety of NoSQL technologies including MongoDB, Neo4J, Elasticsearch, Solr, Redis, Gemfire, Couchbase and Cassandra. Spring Boot provides auto-configuration for Redis, MongoDB, Neo4j, Elasticsearch, Solr and Cassandra; you can make use of the other projects, but you will need to configure them yourself. Refer to the appropriate reference documentation at projects.spring.io/spring-data.

Spring Data for MongoDB is part of the umbrella Spring Data project which aims to provide a familiar and consistent Spring-based programming model for new datastores while retaining store-specific features and capabilities.

The Spring Data MongoDB project provides integration with the MongoDB document database. Key functional areas of Spring Data MongoDB are a POJO centric model for interacting with a MongoDB DBCollection and easily writing a Repository style data access layer.

Spring Data MongoDB

MongoDB is an open-source NoSQL document database that uses a JSON-like schema instead of traditional table-based relational data. Spring Boot offers several conveniences for working with MongoDB, including the spring-boot-starter-data-mongodb 'Starter'. Spring Data includes repository support for MongoDB.

Spring Boot offers auto-configuration for Embedded Mongo. To use it in your Spring Boot application add a dependency on de.flapdoodle.embed:de.flapdoodle.embed.mongo. The port that Mongo will listen on can be configured using the spring.data.mongodb.port property. To use a randomly allocated free port use a value of zero. The MongoClient created by MongoAutoConfiguration will be automatically configured to use the randomly allocated port.

- spring.data.mongodb.host=mongoserver
- spring.data.mongodb.port=27017

Spring Data Transaction Management

Spring Boot and Spring Data JPA make the handling of transactions extremely simple. They enable you to declare your preferred transaction handling and provide seamless integration with Hibernate and JPA.

The only thing you need to do is to annotate one of your methods with @Transactional. But what does that actually do? Which method(s) should you annotate with @Transactional? And how can you set different propagation levels?

What is a transaction?

Transactions manage the changes that you perform in one or more systems. These can be databases, message brokers, or any other kind of software system. The main goal of a transaction is to provide ACID characteristics to ensure the consistency and validity of your data.

ACID

ACID is an acronym that stands for atomicity, consistency, isolation, and durability:

- **Atomicity** describes an all or nothing principle. Either all operations performed within the transaction get executed or none of them. That means if you commit the transaction successfully, you can be sure that all operations got performed. It also enables you to abort a transaction and roll back all operations if an error occurs.
- The **consistency** characteristic ensures that your transaction takes a system from one consistent state to another consistent state. That means that either all operations were rolled back and the data was set back to the state you started with or the changed data passed all consistency checks. In a relational database, that means that the modified data needs to pass all constraint checks, like foreign key or unique constraints, defined in your database.
- **Isolation** means that changes that you perform within a transaction are not visible to any other transactions until you commit them successfully.
- **Durability** ensures that your committed changes get persisted.

ACID

- As you can see, a transaction that ensures these characteristics makes it very easy to keep your data valid and consistent.
- Relational databases support ACID transactions, and the JDBC specification enables you to control them. Spring provides annotations and different transaction managers to integrate transaction management into their platform and to make it easier to use. But in the end, it all boils down to the features provided by these lower-level APIs.

ACID

Using transactions with JDBC

There are 3 main operations you can do via the java.sql.Connection interface to control an ACID transaction on your database.

```
try (Connection con = dataSource.getConnection()) {
    con.setAutoCommit(false);

    // do something ...

    con.commit();
} catch (SQLException e) {
    con.rollback();
}
```

ACID

- Start a transaction by getting a Connection and deactivating auto-commit. This gives you control over the database transaction. Otherwise, you would automatically execute each SQL statement within a separate transaction.
- Commit a transaction by calling the commit() method on the Connection interface. This tells your database to perform all required consistency checks and persist the changes permanently.
- Rollback all operations performed during the transaction by calling the rollback() method on the Connection interface. You usually perform this operation if an SQL statement failed or if you detected an error in your business logic.

- As you can see, conceptually, controlling a database transaction isn't too complex. But implementing these operations consistently in a huge application, it's a lot harder than it might seem. That's where Spring's transaction management comes into play.

Managing Transactions with Spring

- Spring provides all the boilerplate code that's required to start, commit, or rollback a transaction. It also integrates with Hibernate's and JPA's transaction handling. If you're using Spring Boot, this reduces your effort to a @Transactional annotation on each interface, method, or class that shall be executed within a transactional context.

```
@Service
public class AuthorService {
    private AuthorRepository authorRepository;
@Transactional
    public void updateAuthorNameTransaction() {
        Author author = authorRepository.findById(1L).get();
} }
```

Managing Transactions with Spring

- The @**Transactional** annotation tells Spring that a transaction is required to execute this method. When you inject the AuthorService somewhere, Spring generates a proxy object that wraps the AuthorService object and provides the required code to manage the transaction.
- By default, that proxy starts a transaction before your request enters the first method that's annotated with @**Transactional**. After that method got executed, the proxy either commits the transaction or rolls it back if a RuntimeException or Error occurred. Everything that happens in between, including all method calls, gets executed within the context of that transaction.
- The @**Transactional** annotation supports a set of attributes that you can use to customize the behavior. The most important ones are propagation, readOnly, rollbackFor, and noRollbackFor. Let's take a closer look at each of them

Managing Transactions with Spring

The annotation supports further configuration as well:

- the Propagation Type of the transaction
- the Isolation Level of the transaction
- a Timeout for the operation wrapped by the transaction
- a readOnly flag – a hint for the persistence provider that the transaction should be read only
- the Rollback rules for the transaction

Note that by default, rollback happens for runtime, unchecked exceptions only. The checked exception does not trigger a rollback of the transaction. We can, of course, configure this behavior with the rollbackFor and noRollbackFor annotation parameters.

Managing Transactions with Spring

Transactions and Proxies:

At a high level, Spring creates proxies for all the classes annotated with @Transactional, either on the class or on any of the methods. The proxy allows the framework to inject transactional logic before and after the running method, mainly for starting and committing the transaction.

What's important to keep in mind is that, if the transactional bean is implementing an interface, by default the proxy will be a Java Dynamic Proxy. This means that only external method calls that come in through the proxy will be intercepted. Any self-invocation calls will not start any transaction, even if the method has the @Transactional annotation.

Another caveat of using proxies is that only public methods should be annotated with @Transactional. Methods of any other visibilities will simply ignore the annotation silently as these are not proxied.

Transaction Propogation

They enable you to control the handling of existing and creation of new transactions. You can choose between:

- REQUIRED to tell Spring to either join an active transaction or to start a new one if the method gets called without a transaction. This is the default behavior.
- SUPPORTS to join an activate transaction if one exists. If the method gets called without an active transaction, this method will be executed without a transactional context.
- MANDATORY to join an activate transaction if one exists or to throw an Exception if the method gets called without an active transaction.

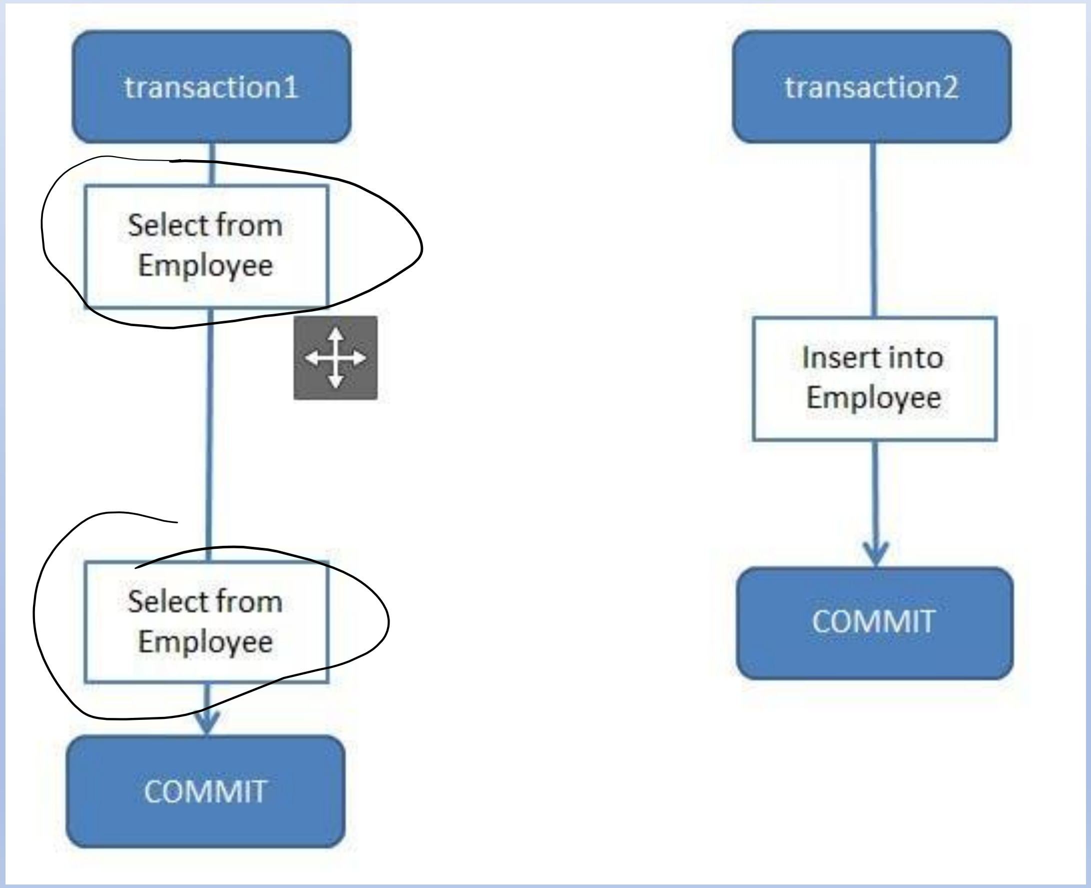
Transaction Propogation

- NEVER to throw an Exception if the method gets called in the context of an active transaction.
- NOT_SUPPORTED to suspend an active transaction and to execute the method without any transactional context.
- REQUIRES_NEW to always start a new transaction for this method. If the method gets called with an active transaction, that transaction gets suspended until this method got executed.
- NESTED to start a new transaction if the method gets called without an active transaction. If it gets called with an active transaction, Spring sets a savepoint and rolls back to that savepoint if an Exception occurs.

Transaction Isolation

Transaction Isolation defines the database state when two transactions concurrently act on the same database entity. It involves locking of database records. So it describes the **behaviour or state of the database when one transaction is working on database entity and then some other concurrent transaction tries to simultaneously access/edit the same database entity.**
The ANSI/ISO standard defines four isolation levels. **Isolation is one of the ACID (Atomicity, Consistency, Isolation, Durability) properties.** So transaction isolation level is not something specific to Spring Framework. Using Spring we can change the isolation level to suit our business logic.

Transaction Isolation



Transaction Isolation

Types of Transaction Isolation Levels:

SERIALIZABLE
If two transactions are executing concurrently then it is as if the transactions get executed serially i.e the first transaction gets committed only then the second transaction gets executed. This is **total isolation**. So a running transaction is never affected by other transactions. However this may cause issues as **performance will be low and deadlock might occur**.

REPEATABLE_READ
If two transactions are executing concurrently - **till the first transaction is committed the existing records cannot be changed by second transaction but new records can be added.** After the second transaction is committed, the new added records get reflected in first transaction which is still not committed. For MySQL the default isolation level is REPEATABLE_READ.
However the REPEATABLE READ isolation level behaves differently when using mysql. When using MYSQL we are not able to see the newly added records that are committed by the second transaction.

Transaction Isolation

Types of Transaction Isolation Levels:

READ_COMMITTED
If two transactions are executing concurrently - **before the first transaction is committed the existing records can be changed as well as new records can be changed by second transaction.** After the second transaction is committed, the newly added and also updated records get reflected in first transaction which is still not committed.

READ_UNCOMMITTED
If two transactions are executing concurrently - before the first transaction is committed the existing records can be changed as well as new records can be changed by second transaction. **Even if the second transaction is not committed the newly added and also updated records get reflected** in first transaction which is still not committed.

Transaction Isolation

- **Dirty Reads -** Suppose two transactions - Transaction A and Transaction B are running concurrently. If Transaction A modifies a record but not commits it. Transaction B reads this record but then Transaction A again rollbacks the changes for the record and commits it. So Transaction B has a wrong value.
- **Non-Repeatable Reads -** Suppose two transactions - Transaction A and Transaction B are running concurrently. If Transaction A reads some records. Transaction B modifies these records before transaction A has been committed. So if Transaction A again reads these records they will be different. So same select statements result in different existing records.
- **Phantom Reads -** Suppose two transactions - Transaction A and Transaction B are running concurrently. If Transaction A reads some records. Transaction B adds more such records before transaction A has been committed. So if Transaction A again reads there will be more records than the previous select statement. So same select statements result in different number records to be displayed as new records also get added.

Transaction Isolation Summary

Isolation Level	**Dirty Reads**	**Non-Repeatable Reads**	**Phantom Reads**
SERIALIZABLE	This scenario is not possible as the second transaction cannot start execution until the first is committed. They never execute parallelly but only sequentially	This scenario is not possible as the second transaction cannot start execution until the first is committed. They never execute parallelly but only sequentially	This scenario is not possible as the second transaction cannot start execution until the first is committed. They never execute parallelly but only sequentially
REPEATABLE_READ	This scenario is not possible as any existing record change gets reflected only if the transaction is committed. So other transaction will never read wrong value.	This scenario is not possible since any record can be changed only after a transaction has been committed. So multiple select statements before transaction commit will always return same existing records.	This scenario is possible as other transactions can insert new records even if first transaction commit has not taken place.
READ_COMMITTED	This scenario is not possible as any existing record change gets reflected only if the transaction is committed. So other transaction will never read wrong value.	This scenario is possible as other transactions can modify existing records even if first transaction commit has not taken place.	This scenario is possible as other transactions can insert new records even if first transaction commit has not taken place.
READ_UNCOMMITTED	This scenario is possible as any record can be read by other transactions even if the first transaction is not committed. So if first transaction rollbacks the record changes then other transactions will have wrong values	This scenario is possible since any record can be changed even if a transaction is not committed.	This scenario is possible as any record can be inserted even if a transaction is not committed.

Read-Only Transactions

- If you want to implement a read-only operation, I recommend using a DTO projection. It enables you to only read the data you actually need for your business code and provides a much better performance.

```
@Service
public class AuthorService {
    private AuthorRepository authorRepository;
     @Transactional(readOnly = true)
    public Author getAuthor() {
        return authorRepository.findById(1L).get();
    }
}
```

Handling Exceptions

- I explained earlier, that the Spring proxy automatically rolls back your transaction if a RuntimeException or Error occurred. You can customize that behavior using the rollbackFor and noRollbackFor attributes of the @Transactional annotation.

- As you might guess from its name, the rollbackFor attribute enables you to provide an array of Exception classes for which the transaction shall be rolled back. And the noRollbackFor attribute accepts an array of Exception classes that shall not cause a rollback of the transaction.

- In the following example, I want to roll back the transaction for all subclasses of the Exception class except the EntityNotFoundException.

Handling Exceptions

```
@Service
public class AuthorService {
     private AuthorRepository authorRepository;
     @Transactional
        (rollbackFor = Exception.class,  noRollbackFor = EntityNotFoundException.class)
    public void updateAuthorName() {
        Author author = authorRepository.findById(1L).get();
        author.setName("new name");
```

Transaction Logging

A helpful method to understand transactional related issues is fine-tuning logging in the transactional packages. The relevant package in Spring is “org.springframework.transaction”, which should be configured with a logging level of TRACE.

Transaction Rollback

The @Transactional annotation is the metadata that specifies the semantics of the transactions on a method. We have two ways to rollback a transaction: declarative and programmatic.

In the **declarative approach**, we annotate the methods with the @Transactional annotation. The @Transactional annotation makes use of the attributes rollbackFor or rollbackForClassName to rollback the transactions, and the attributes noRollbackFor or noRollbackForClassName to avoid rollback on listed exceptions.

Transaction Rollback

The default rollback behavior in the declarative approach will rollback on runtime exceptions.

Let's see a simple example using the declarative approach to rollback a transaction for runtime exceptions or errors:

```
@Transactional
public void createCourseDeclarativeWithRuntimeException(Course course) {
    courseDao.create(course);
    throw new DataIntegrityViolationException("Throwing exception for demoing
Rollback!!!");
}
```

Transaction Rollback

The declarative approach to rollback a transaction for the listed checked exceptions. The rollback in our example is on SQLException:

```
@Transactional(rollbackFor = { SQLException.class })
public void createCourseDeclarativeWithCheckedException(Course course) throws SQLException {
    courseDao.create(course);
    throw new SQLException("Throwing exception for demoing rollback");
}
```

Transaction Rollback

A simple use of attribute noRollbackFor in the declarative approach to prevent rollback of the transaction for the listed exception:

```
@Transactional(noRollbackFor = { SQLException.class })
public void createCourseDeclarativeWithNoRollBack(Course course) throws SQLException {
    courseDao.create(course);
    throw new SQLException("Throwing exception for demoing rollback");
}
```

Transaction Rollback

In the programmatic approach, we rollback the transactions using TransactionAspectSupport:

```
public void createCourseDefaultRatingProgramatic(Course course) {
    try {
       courseDao.create(course);
    } catch (Exception e) {
       TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();
    }
}
```

The declarative rollback strategy should be favored over the programmatic rollback strategy.