Data Access

- Spring provides support for following:
  - ✓ JDBC
  - ✓ Object Relational Mapping (ORM) - Hibernate and JPA
  - ✓ Data Access Objects (DAO)
  - ✓ NoSQL Databases
- Spring provides a convenient translation from technology-specific exceptions like SQLException to its own exception class hierarchy with the DataAccessException as the root exception.
- In addition to JDBC exceptions, Spring can also wrap Hibernate-specific exceptions, converting them from proprietary exception to a set of focused runtime exceptions (the same is true for JDO and JPA exceptions).

DataSource configuration in application.properties

```
spring.datasource.url=database_connection_url
spring.datasource.username=username
spring.datasource.password=password
spring.datasource.driver-class=driver_class
```

Use spring.datasource.jndi-name property to connect to database through JNDI in application server.

```
spring.datasource.jndi-name=java:jboss/datasources/customers
```

Spring JDBC Classes

- JdbcTemplate
- NamedParameterJdbcTemplate
- SimpleJdbcInsert
- SimpleJdbcCall

Data Access - JDBC

Action	Spring	You
Define connection parameters.		X
Open the connection.	X	
Specify the SQL statement.		X
Declare parameters and provide parameter values		X
Prepare and execute the statement.	X	
Set up the loop to iterate through the results (if any).	X	
Do the work for each iteration.		X
Process any exception.	X	
Handle transactions.	X	
Close the connection, statement and resultset.	X	

JDBC classes

JdbcTemplate

The classic Spring JDBC approach and the most popular. This "lowest level" approach and all others use a JdbcTemplate under the covers.

NamedParameterJdbcTemplate

Wraps a JdbcTemplate to provide named parameters instead of the traditional JDBC "?" placeholders. This approach provides better documentation and ease of use when you have multiple parameters for an SQL statement.

SimpleJdbcInsert and SimpleJdbcCall

Allows you to provide only the name of the table or procedure and provide a map of parameters matching the column names.

DataAccessException

- All data access exception in Spring are subclasses of this exception.
- This is an unchecked exception. So you are not forced to handle data access exceptions in Spring.
- This is data access API agnostic.

JDBC Starters in POM.XML

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

JDBC and Spring Boot

- Spring's JdbcTemplate and NamedParameterJdbcTemplate classes are auto-configured, and you can @Autowire them directly into your own beans.
- You can customize some properties of the template by using the spring.jdbc.template.* properties, as shown in the following example:

```
spring.jdbc.template.max-rows=500
```

- The NamedParameterJdbcTemplate reuses the same JdbcTemplate instance behind the scenes. If more than one JdbcTemplate is defined and no primary candidate exists, the NamedParameterJdbcTemplate is not auto-configured.

JdbcTemplate Methods

void execute(String sql)

Issue a single SQL execute, typically a DDL statement.

<T> List<T>query(String sql, ParameterizedRowMapper<T> rm,Object... args)

Query for a List of Objects of type T using the supplied ParameterizedRowMapper to the query results to the object.

<T> T queryForObject(String sql, Class<T> requiredType, Object... args)

Execute a query for a result object, given static SQL.

List<Map<String,Object>> queryForList(String sql, Object... args)

Execute the supplied query with the (optional) supplied arguments. Parameters are represented by ?.

JdbcTemplate Methods

<T> T queryForObject(String sql, ParameterizedRowMapper<T> rm, Object... args)

Query for an object of type T using the supplied ParameterizedRowMapper to the query results to the object.

int update(String sql, Object... args)

Executes the supplied SQL statement with (optional) supplied arguments.

JdbcTemplate Query Examples

```
int rowCount = this.jdbcTemplate.queryForObject
    ("select count(*) from  employees“, Integer.class);
```

```
String lastName = this.jdbcTemplate.queryForObject
("select last_name from employees where employee_id = ?",
   new Object[]{111}, String.class);
```

```
List<Actor> actors = this.jdbcTemplate.query
("select first_name, last_name  from employees",
   new RowMapper<Employee>() {
    public Employee mapRow(ResultSet rs, int rowNum)
                           throws SQLException {
     Employee e = new Employee();
     e.setFirstName(rs.getString("first_name"));
     e.setLastName(rs.getString("last_name"));
     return e;
    }
});
```

JdbcTemplate Examples

```
jdbcTemplate.update
("insert into jobs (job_id, job_title) values (?, ?)",
"IT_DBA", "Oracle DBA");
```

```
jdbcTemplate.update
     ("delete from jobs where job_id = ?", "IT_DBA"));
```

```
this.jdbcTemplate.execute
   ("create table mytable (id integer, name varchar(100))");
```

NamedParameterJdbcTemplate Example

```
@SpringBootApplication
public class ListEmployees implements CommandLineRunner {
        @Autowired
        private NamedParameterJdbcTemplate jdbcTemplate;
        public static void main(String[] args) {
         SpringApplication.run(ListEmployees.class, args);
        }
        @Override
        public void run(String... args) {
          List<String> names = jdbcTemplate.queryForList(
          "select first_name from employees where salary between
                     :low and :high",
          new MapSqlParameterSource().addValue("low", 1000)
                                     .addValue("high", 5000),
          String.class);

          for (String name : names)
              System.out.println(name);
      }
}
```

SimpleJdbcInsert Example

```
@Component
public class AddJob{
	private SimpleJdbcInsert sji;

	public AddJob(DataSource datasource) {
	    this.sji = new SimpleJdbcInsert(datasource)
	                    .withTableName(“jobs”);
	}

	public void add() {
		HashMap<String,Object> job = new HashMap<>();
		job.put("job_id","SD");
		job.put("job_title","Spring Developer");
		int count = sji.execute(job);
		System.out.println("Added Job Successfully");
	}
}
```

Spring Transaction Management

- Provides a consistent programming model across different transaction APIs such as JTA, JDBC, Hibernate, JPA, and JDO.
- Supports declarative transaction management.
- Provides a simpler API for programmatic transaction management than a number of complex transaction APIs such as JTA.
- Integrates very well with Spring's various data access abstractions.
- You write your code once, and it can benefit from different transaction management strategies in different environments

Global vs. Local Transactions

- Global transactions enable you to work with multiple transactional resources, typically relational databases and message queues.
- The application server manages global transactions through the JTA
- Usage of global transaction confines application to Application server as only Application server provides JTA
- A JDBC transaction is called local transaction
- They are easy to use but cannot work with multiple resources

Transaction Managers

- Spring provides PlatformTransactionManager interface
- Its methods throw unchecked exception TransactionException

```
public interface PlatformTransactionManager {
  TransactionStatus getTransaction
        (TransactionDefinition definition)
                throws TransactionException;
  void commit(TransactionStatus status)
                throws TransactionException;
  void rollback(TransactionStatus status)
                throws TransactionException;
}
```

TransactionStatus

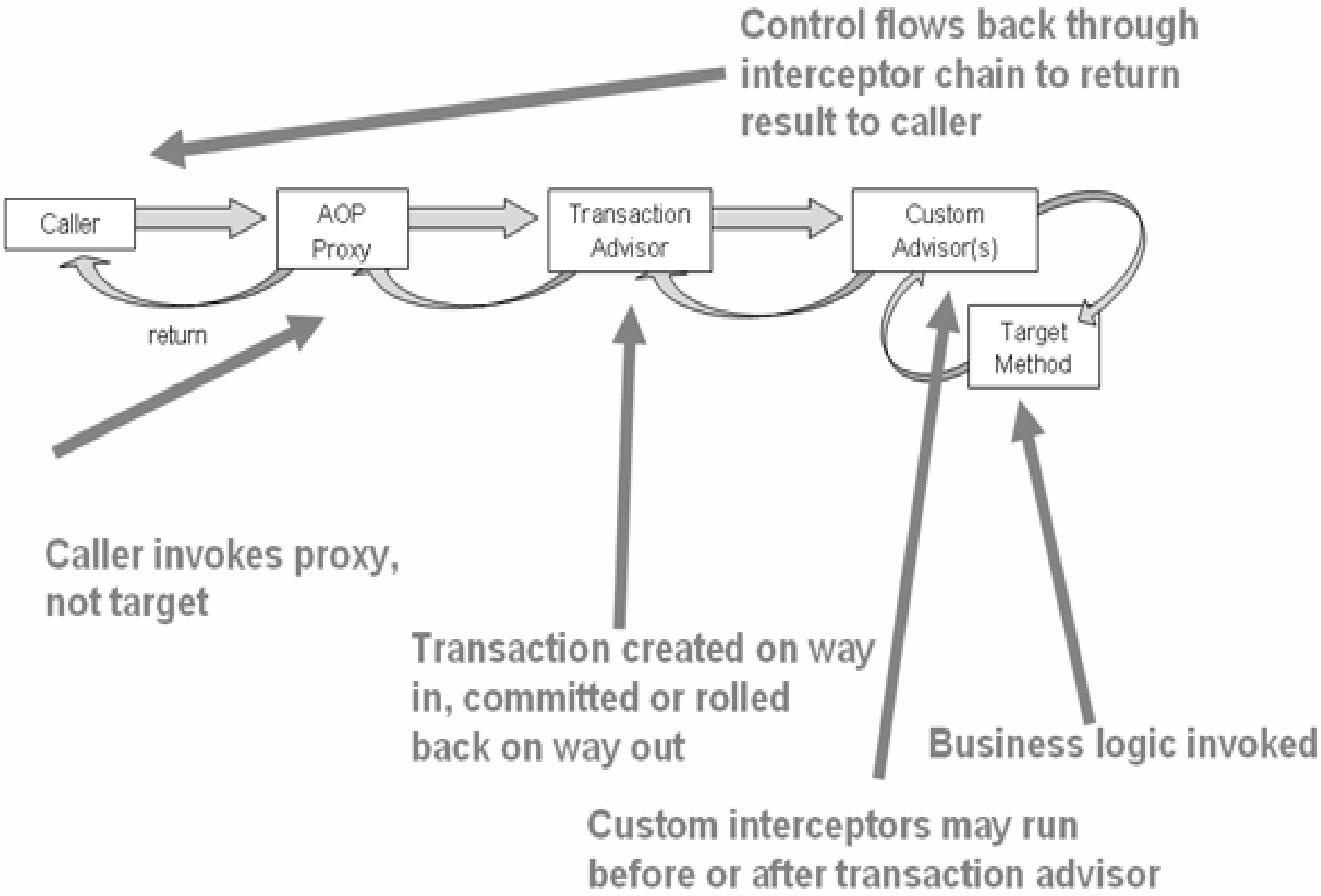
- TransactionStatus interface provides methods to control transaction and query transaction status

```
public interface TransactionStatus extends SavepointManager {
    boolean isNewTransaction();
    boolean hasSavepoint();
    void setRollbackOnly();
    boolean isRollbackOnly();
    void flush();
    boolean isCompleted();
}
```

Available Transaction Managers

- DataSourceTransactionManager
- JtaTransactionManager
- HibernateTransactionManager

Transaction Management



When a transaction is rolled back?

- To rollback work is to throw an Exception from code that is currently executing in the context of a transaction.
- In its default configuration, the Spring Framework's transaction infrastructure code *only marks a* transaction for rollback in the case of runtime, unchecked exceptions
- Checked exceptions that are thrown from a transactional method do *not result in rollback in the* default configuration.
- You can configure exactly which Exception types mark a transaction for rollback, including checked exceptions using configuration.

Transaction Example

```
@Component
public class EmployeeManager {
 @Autowired private JdbcTemplate jdbcTemplate;
 @Transactional(propagation = Propagation.REQUIRED)
 public void changeSalaries(int first, int second) {
   TransactionStatus status =
                TransactionAspectSupport
                          .currentTransactionStatus();
   try {
    int count=jdbcTemplate.update
     ("update employees set salary=10000 where employee_id="
                 + first);
    if (count == 0)
      throw new UpdateException
                 ("Update of " + first + " failed!");
    count=jdbcTemplate.update
     ("update employees set salary=10000 where employee_id=“
                  + second);
```

Transaction Example

```
   if (count == 0)
       throw new UpdateException
            ("Update of " + second + "    failed!");
     System.out.println("Commiting transaction ...");
  }
  catch (Exception ex) {
      System.out.println("Rolling back transaction ....");
      status.setRollbackOnly();
  } // catch
 } // changeSalaries
} // EmployeeManager
```

Transaction Example

```
public class UpdateException extends RuntimeException {
    public UpdateException(String msg) {
            super(msg);
    }
}
```

Propagation Options

MANDATORY	Support a current transaction, throw an exception if none exists.
NESTED	Execute within a nested transaction if a current transaction exists, behave like PROPAGATION_REQUIRED else.
NEVER	Execute non-transactionally, throw an exception if a transaction exists.
NOT_SUPPORTED	Execute non-transactionally, suspend the current transaction if one exists.
REQUIRED	Support a current transaction, create a new one if none exists.
REQUIRES_NEW	Create a new transaction, and suspend the current transaction if one exists.
SUPPORTS	Support a current transaction, execute non-transactionally if none exists.

JPA Starter in POM.XML

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

When Spring Boot finds JPA starter, it does the following:

- Hibernate : One of the most popular JPA implementations.
- Spring Data JPA : Makes it easy to implement JPA-based repositories.
- Spring ORMs : Core ORM support from the Spring Framework.

Entity class

All classes associated with @Entity, @Embeddable or @MappedSuperclass are considered.

Repository Interface

- The central interface in the Spring Data repository abstraction is Repository.
- It takes the domain class to manage as well as the ID type of the domain class as type arguments.
- The CrudRepository provides sophisticated CRUD functionality for the entity class that is being managed.

CrudRepository Interface

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

CrudRepository Interface Methods

```
long count()
Returns the number of entities available.

void delete(T entity)
Deletes a given entity.

void deleteAll()
Deletes all entities managed by the repository.

void deleteAll(Iterable<? extends T> entities)
Deletes the given entities.

void deleteById(ID id)
Deletes the entity with the given id.

boolean existsById(ID id)
Returns whether an entity with the given id exists.
```

CrudRepository Interface

```
Iterable<T> findAll()
Returns all instances of the type.

Iterable<T> findAllById(Iterable<ID> ids)
Returns all instances of the type with the given IDs.

Optional<T> findById(ID id)
Retrieves an entity by its id.

<S extends T> S save(S entity)
Saves a given entity.

<S extends T> Iterable<S> saveAll(Iterable<S> entities)
Saves all given entities.
```

PagingAndSortingRepository Interface

- Repository fragment to provide methods to retrieve entities using the pagination and sorting abstraction.

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

Returns a Page of entities meeting the paging restriction provided in the Pageable object.

```
Iterable<T> findAll(Sort sort)
```

Returns all entities sorted by the given options.

Pagable Interface

Abstract interface for pagination information.

```
static Pageable ofSize(int pageSize)
```
Creates a new Pageable for the first page (page number 0) given pageSize .

```
Pageable next()
```
Returns the Pageable requesting the next Page.

```
Pageable first()
```
Returns the Pageable requesting the first page.

```
Pageable withPage(int pageNumber)
```
Creates a new Pageable with pageNumber applied.

```
static Pageable ofSize(int pageSize)
```
Creates a new Pageable for the first page (page number 0) given pageSize .

PageRequest Interface

- **Basic Java Bean implementation of Pageable.**

static PageRequest of(int page, int size)
Creates a new unsorted PageRequest.
static PageRequest of(int page, int size, Sort sort)
Creates a new PageRequest with sort parameters applied.
static PageRequest of(int page, int size, Sort.Direction direction, String... properties)
Creates a new PageRequest with sort direction and properties applied.
static PageRequest ofSize(int pageSize)
Creates a new PageRequest for the first page (page number 0) given pageSize .

```
PageRequest.of(1,20)  // second page with size 20
PageRequest.of(0,5, Sort.by("amount").descending()))
```

Sort Class

- **Sort option for queries.**
- **You have to provide at least a list of properties to sort for that must not include null or empty strings.**
- **The direction defaults to DEFAULT_DIRECTION.**

static Sort by(String... properties)

Creates a new Sort for the given properties.

Sort and(Sort sort)

Returns a new Sort consisting of the Sort.Orders of the current Sort combined with the given ones.

Sort ascending()

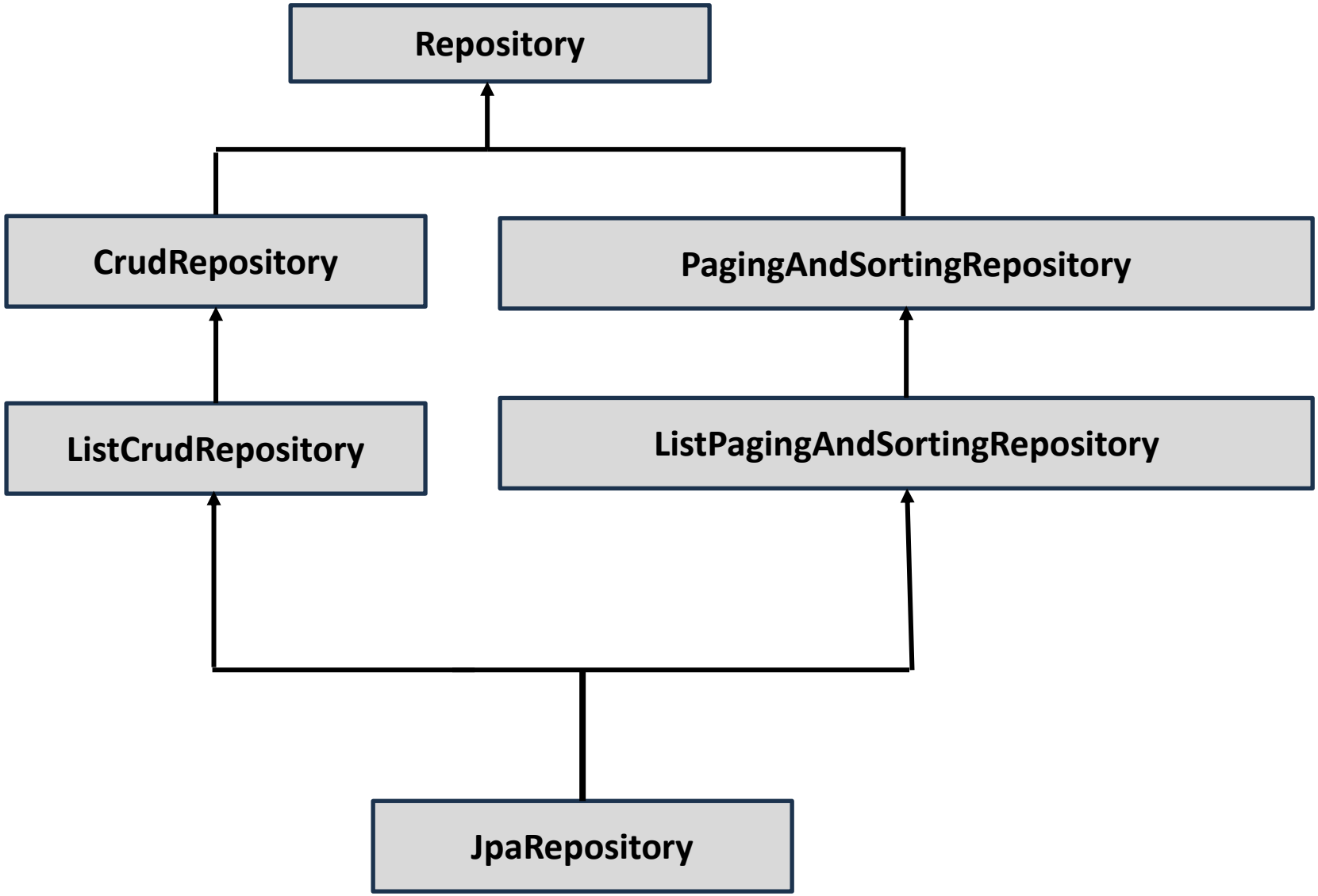
Returns a new Sort with the current setup but ascending order direction.

Sort descending()

Returns a new Sort with the current setup but descending order direction.

```
Sort sort = Sort.by("name").ascending().and
          (Sort.by("description").descending());
```

Repositories Hierarchy



JPARepository Interface

- JPA specific extension of Repository.
- Extends ListCrudRepository and PagingAndSortingRepository

void deleteAllByIdInBatch (Iterable<ID> ids)	Deletes the entities identified by the given ids using a single query.
void deleteAllInBatch()	Deletes all entities in a batch call.
void deleteAllInBatch (Iterable<T> entities)	Deletes the given entities in a batch which means it will create a single query.
<S extends T> List<S> findAll (Example<S> example)	Returns all entities matching the given Example.
void flush()	Flushes all pending changes to the database.
T getReferenceById(ID id)	Returns a reference to the entity with the given identifier.
<S extends T> List<S> saveAllAndFlush (Iterable<S> entities)	Saves all entities and flushes changes instantly.
<S extends T> S saveAndFlush(S entity)	Saves an entity and flushes changes instantly.

Query Derivation

- When query is derived from method name, it is called query derivation, and methods are called query methods.
- Query derivation for both count and delete queries is available.
- The mechanism strips the prefixes find…By, read…By, query…By, count…By, and get…By from the method and starts parsing the rest of it.
- The introducing clause can contain further expressions, such as a Distinct to set a distinct flag on the query to be created.
- And and Or can be used to combine multiple property expressions. You can also use Between, LessThan, and GreaterThan
- Apply IsStartingWith, StartingWith, StartsWith, IsEndingWith, EndingWith, EndsWith, IsContaining, Containing, Like, IsNotContaining, NotContaining, and NotContains
- Apply IgnoreCase against a single field, or if you want to apply it to all properties, use AllIgnoreCase at the end
- Apply OrderBy with Asc or Desc against a field when you know the ordering in advance
- You can limit the results of query methods by using the *First* or *Top* keywords followed by an optional numeric to specify the maximum result size to be returned.

Query Derivation Examples

```
List<User> findByNameStartingWith(String prefix)
List<User> findByNameEndingWith(String suffix)
List<User> findByNameContaining(String infix)
List<User> findByNameLike(String likePattern)
List<User> findByAgeLessThan(Integer age)
List<User> findByAgeBetween(Integer startAge,Integer endAge)
List<User> findByAgeIn(Collection<Integer> ages)
List<User> findByNameOrderByName(String name)
List<User> findByNameOrderByNameDesc(String name)
User findFirstByOrderByLastnameAsc()
User findTopByOrderByAgeDesc()
List<User> findByAgeIsNull()
List<User> findByFirstNameAndLastName
                (String firstName, String lastName)
long countByLastname(String lastname);
List<User> findTop3ByLastname(String lastname)
List<User> findFirst10ByLastname(String lastname, Sort sort);
```

@Query

- We can bind a query with a method directly by using @Query annotation.
- The given query is transformed into a valid JPQL query.
- Upon query execution, the parameter passed to the method call replace parameters (?1) used in query.
- Allows for running native queries by setting the nativeQuery flag to true.

```
@Query(value="SELECT * FROM USERS WHERE EMAIL_ADDRESS = ?1",
   nativeQuery = true)
User findByEmailAddress(String emailAddress);
```

EmployeeRepository Interface

```
import java.util.List;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.CrudRepository;

public interface EmployeeRepository extends
                        CrudRepository<Employee,Integer>{

  List<Employee> findEmployeesByDept(int dept);
  List<Employee> findEmployeesByJob(String job);
  int  countByJob(String job);

  @Query("from Employee e where e.salary > ?1")
  List<Employee> findCostlyEmployees(int salary);
}
```

ListEmployees.java

```
@Autowired
private EmployeeRepository empRepository;

public void test() {
   System.out.printf("No. of Programmers :  %d\n",
                     empRepository.countByJob("IT_PROG"));

   for(Employee emp:
            empRepository.findCostlyEmployees(20000))
      System.out.println(emp.getFirstName());

   for(Employee emp : empRepository.findAll())
      System.out.println(emp.getFirstName());
}
```

UpdateSalary.java

```
@Autowired
private EmployeeRepository empRepository;

public void changeSalary(int empid, int newSalary) {
        Optional<Employee> emp =
                    empRepository.findById(empid);
        if(emp.isPresent()) {
           Employee employee = emp.get();
           employee.setSalary(newSalary);
           empRepository.save(employee);
           System.out.println("Updated Salary!");
        }
        else
           System.out.println("Sorry! Employee not found!");
}
```