Spring Data - 1.0

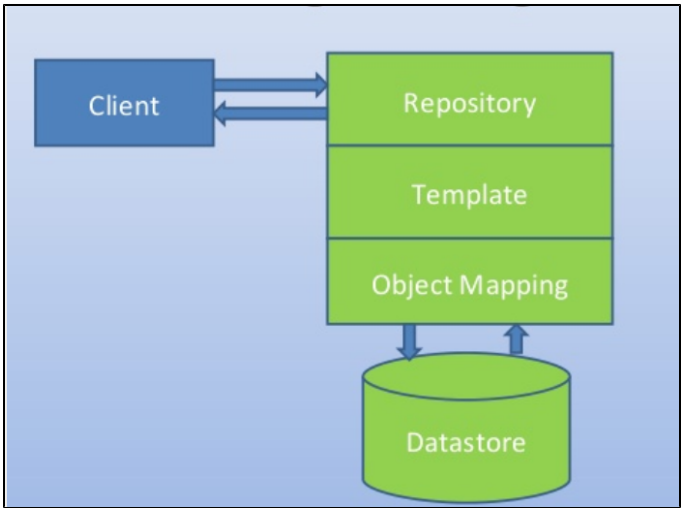
On this page:

Summary: This document provides information about the extension providing **Spring Data** support for JEE.

Description

In what situation should you install this extension?

This extension is specifically targeted at the **Spring Data framework** and should be used in conjunction with the JEE Analyzer extension. **JPA CRUD operations** and **JPA Named Queries** (@NamedQuery and @NamedQueries Annotations) are supported. When client code uses any of these coding mechanisms, the extension will create the links from the **calling method** to the **database table**. This helps form the complete transaction.



Features

Managing Crud method calls

Spring Data Repository abstraction is used to significantly reduce the amount of boilerplate code required to implement data access layers for various persistence stores. A number of crud methods are provided to improve data access. Some of the crud methods are as follows:

- count
- deleteById
- delete
- deleteAll
- deleteAllInBatch

- deleteInBatch
- exists
- existsById
- findAll
- findById
- findOne
- flush
- save
- saveAll
- saveAndFlush

Example code using crud methods:

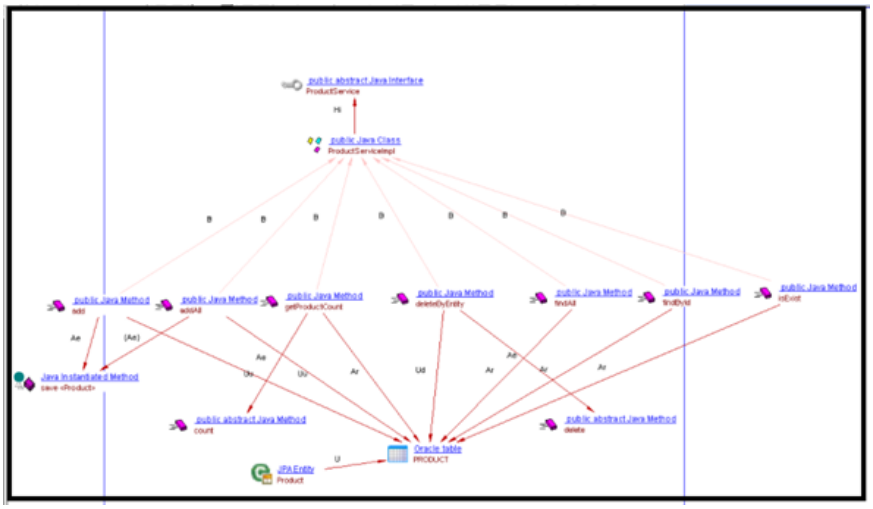
ProductServiceImpl.java

```
@Component
public class ProductServiceImpl implements ProductService{

	@Autowired
	private ProductRepository productRepository;

	@Transactional
	@Override
	public void add(Product product) {
		productRepository.save(product);
	}

	@Transactional(readOnly=true)
	@Override
	public List<Product> findAll() {
		return productRepository.findAll();
	}

	@Override
	public Product findById(long id) {
		return productRepository.findOne(id);
	}
```

Repository: ProductRepository.java

```
@Repository
public interface ProductRepository extends JpaRepository<Product, Long> {

	/** No need to define findAll() here, because
	 *	inherited from JpaRepository with many other basic JPA operations.**/
	public List<Product> findAll();


	/** spring-jpa-data understands this method name,
	 *  because it supports the resolution of specific keywords inside method names. **/
	public List<Product> findByNameContainingIgnoreCase(String searchString);


    /** You can define a JPA query.**/
        @Query("select p from Product p where p.name = :name")
    public List<Product> findByNameIs(@Param("name") String name);

        //Page<Product> finadAll(Pageable pageable);

        //List<Product> findByProductSku(SKU sku);

        /** This method will get query from Product class @NamedQuery Annotation  **/
        public List<Product> findByName(String name);

}
```

The code listed above will produce the following links and objects when the Spring Data extension is installed:



Use of JPA NamedQueries

The **@NamedQuery** annotations can be used individually or can coexist in the class definition for an entity. The annotations define the name of the query, as well as the query text. In a real application, you will probably need multiple named queries defined on an entity class. For this, you will need to place multiple **@NamedQuery** annotations inside a **@NamedQueries** annotation.

Example @NamedQueries code:

Post.java

```
@Entity
@Table(name="POST")
@NamedQueries(
            {
                @NamedQuery( name = "@findCustomer", query = "from Customer" )
            }
        )
@NamedQuery(name = "Post.fetchByTitle",
query = "SELECT p.title FROM Post p")
public class Post {
        @Id
```

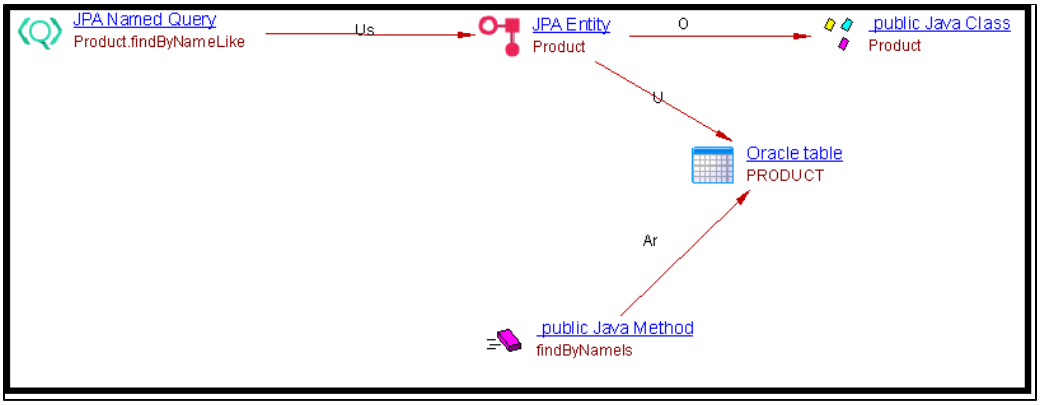
Source code using @NamedQuery: Product.java

```
@Entity
@Table(name= "product")
@NamedQuery(name = "Product.findByName",query = "select p from Product p where p.name = ?1")
public class Product {

    @Id
   /* @GeneratedValue(strategy = GenerationType.IDENTITY)*/
    @GeneratedValue(strategy = GenerationType.SEQUENCE,generator = "id_Sequence")
    @Column(name = "id", updatable = false, nullable = false)
    @SequenceGenerator(name = "id_Sequence", sequenceName = "ID_SEQ")
        public Long id;
    public String name;
```

ProductServiceImpl.java

```
@Component
public class ProductServiceImpl implements ProductService{

@Autowired
private ProductRepository productRepository;

@Transactional(readOnly=true)
@Override
public List<Product> findByNameIs(String name) {
    productRepository.findByName(name);
    return productRepository.findByNameIs(name);
}
```

The code listed above will produce the following links and objects when the Spring Data extension is installed:



Using JPA NamedQueries via XML

NamedQuery works with annotations as well as with XML files. The application's web.xml file contains the **param-value** which indicates the XML file that contains the named query. Using the Spring Data extension, proper links can be created from the methods which call these queries to the data base table.

web.xml

```
<web-app id="WebApp_ID" version="2.4"
        xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

        <display-name>Spring-data Application</display-name>

        <context-param>
                <param-name>contextConfigLocation</param-name>
                <param-value>
                        /WEB-INF/spring-servlet.xml,/WEB-INF/orm.xml
                </param-value>
        </context-param>
```

orm.xml

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">

  <!--  <persistence-unit name="myUnit" transaction-type="RESOURCE_LOCAL">
    <mapping-file>META-INF/orm.xml</mapping-file>
    <exclude-unlisted-classes/>
  </persistence-unit> -->

<!-- Named Query using XML Configuration -->
        <named-query name="Post.fetchByTitle">
                <query>SELECT p FROM Product p WHERE p.name = ?1</query>
        </named-query>
</persistence>
```

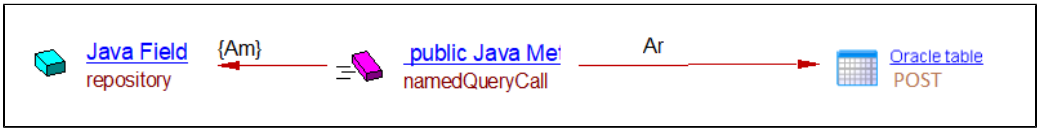
PostService.java

```
@Component
public class PostService {
        @Autowired
        PostRepository repository;


        @Transactional
        public void add(Post post) {
                repository.save(post);
                repository.count();
        }

        @Transactional
        public void namedQueryCall(){
                List<Post> ret = repository.fetchByTitle();
        }
        @Transactional
        public void check()
        {
                repository.delete();
        }

        @Transactional
        public void countByEmailAddress()
        {
                repository.countByEmailAddressAndLastname();
        }
```

PostRepository.java

```
public interface PostRepository extends JpaRepository<Post, Integer> {

        List<Post> countByEmailAddressAndLastname(String emailAddress, String lastname);

}
```

The code listed above will produce the following links and objects when the Spring Data extension is installed. A link will be created from the method **name dQueryCall** to the table **POST**:



Creating query by inference

The query builder mechanism of Spring Data is useful for building queries over entities of the repository. The mechanism is to create the query for patterns such as **find..By**, **read..By**, **query..By**, **count..By**, and **get..By**. Spring Data parses this string as it may contain further expressions, such as a **Distinct** to set a distinct flag on the query to be created. However, the first **By** acts as delimiter to indicate the start of the actual criteria. If such a scenario is used and the methods declared in the repository are used in some other method then using the Spring Data extension the transaction link can be drawn from the method to the database table.

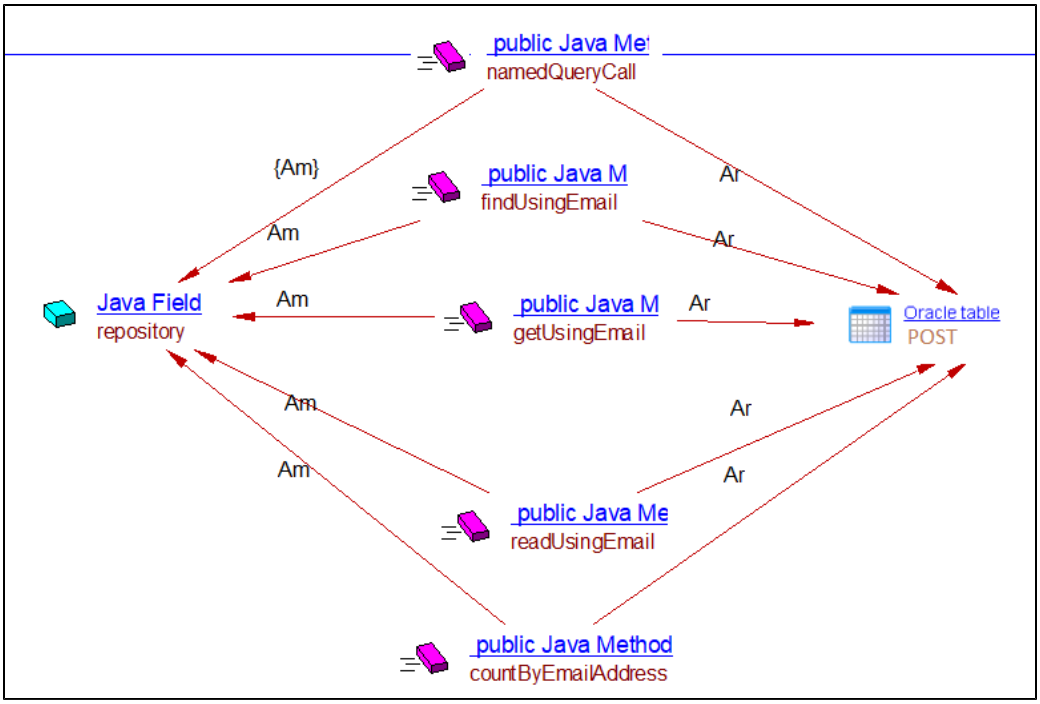
Postservice.java

```
@Component
public class PostService {
	@Autowired
	PostRepository repository;


	@Transactional
	public void add(Post post) {
		repository.save(post);
		repository.count();
	}

	@Transactional
	public void namedQueryCall(){
		List<Post> ret = repository.fetchByTitle();
	}
	@Transactional
	public void check()
	{
		repository.delete();
	}

	@Transactional
	public void countByEmailAddress()
	{
		repository.countByEmailAddressAndLastname();
	}

	@Transactional
	public void findUsingEmail()
	{
		repository.findByEmail();
	}

	@Transactional
	public void readUsingEmail()
	{
		repository.readByEmail();
	}

	@Transactional
	public void getUsingEmail()
	{
		repository.getByEmail();
	}

}
```

PostRepository.java

```
public interface PostRepository extends JpaRepository<Post, Integer> {

	List<Post> findByEmail(String email);
	List<Post> readByEmail(String email);
	List<Post> queryByEmail(String email);
	List<Post> getByEmail(String email);
}
```

The code listed above will produce the following links and objects when the Spring Data extension is installed:



Database table not found

For all of the above features, when the database table is not found by the extension, it creates one **unknown table object** and forms the links associated with it. If multiple classes use the same table then only one table is created.

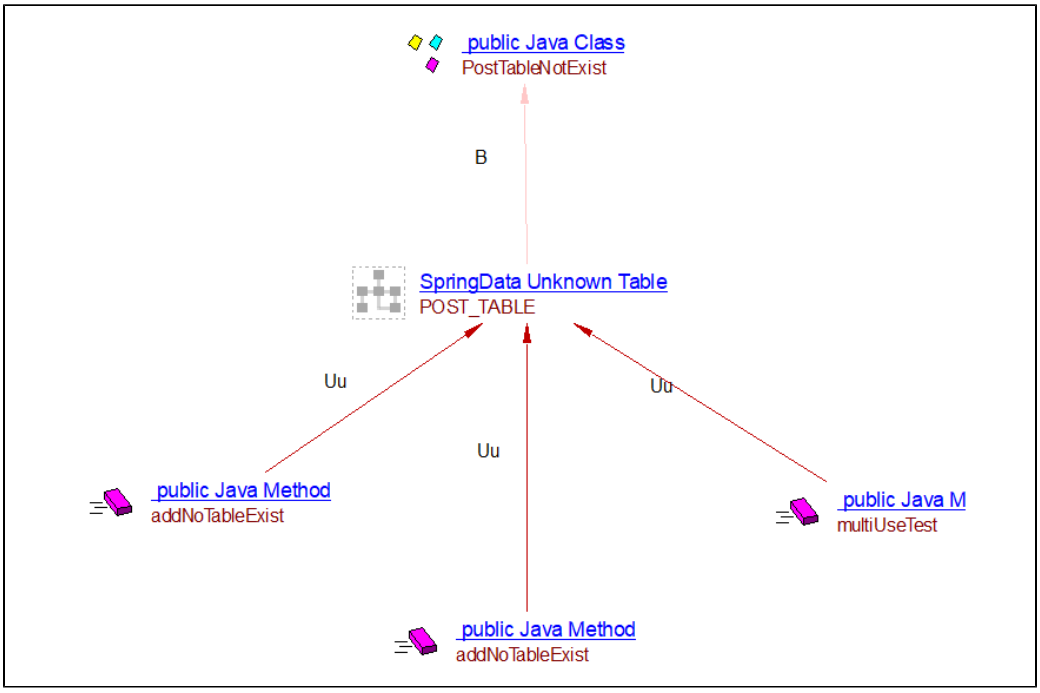
Example: PostTableNotExist.java

```
@Entity
@Table(name="POST_TABLE")
public class PostTableNotExist {
        @Id
        @GeneratedValue(strategy=GenerationType.AUTO)
        @Column(name="POSTID")
        Integer postId;
        @Column(name="TITLE")
        String title;
        public Integer getPostId() {
                return postId;
        }
        public void setPostId(Integer postId) {
                this.postId = postId;
        }
        public String getTitle() {
                return title;
        }
```

PostService.java

```
@Component
public class PostService {


	@Autowired
	PostRepository repository;
	@Autowired
	PostRepository2 repo;
	@Autowired
	PostRepository3 multiUse;

	@Transactional
	public void add(Post post) {
		repository.save(post);
		repository.count();
	}

	@Transactional
	public void addNoTableExist(PostTableNotExist notExisting) {
		repo.save(notExisting);

	}
	@Transactional
	public void multiUseTest(TableNotExistMultyUse multy) {
		multiUse.save(multy);

	}
```

The code listed above will produce the following links and objects when the Spring Data extension is installed:



Handling Query Dsl

Querydsl is a framework which enables the construction of statically typed SQL-like queries, instead of writing queries as inline strings. Querydsl for JPA is an alternative to both JPQL and Criteria queries. Querydsl for JPA/Hibernate is an alternative to both JPQL and JPA 2 Criteria queries. It combines the dynamic nature of Criteria queries with the expressiveness of JPQL and all that in a fully typesafe manner. Using the Spring Data extension the link between the function which uses the query dsl to the JPA enity related to the entity used in query dsl can be identified.

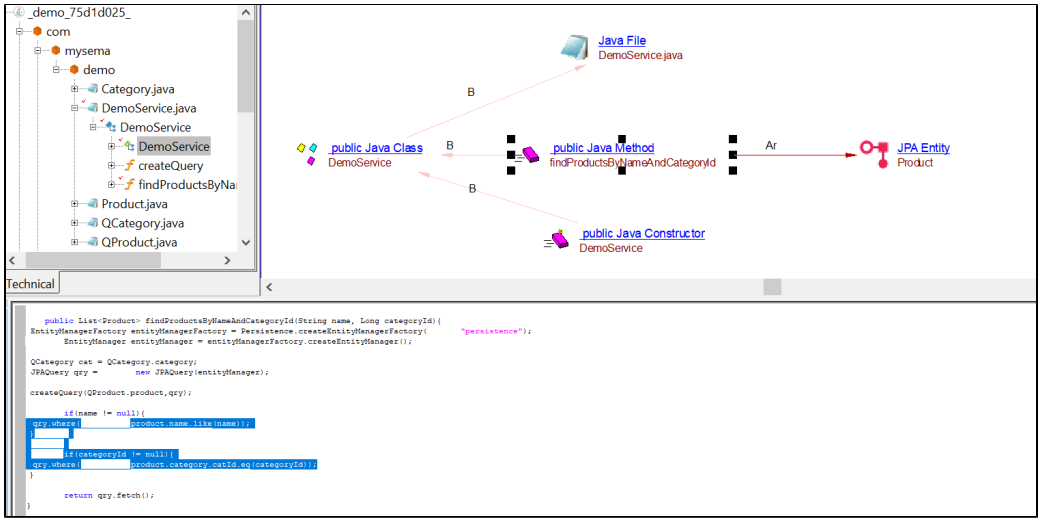
Example: Product.java

```
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

@Entity
@Table(name="PRODUCT")
public class Product {

	@Id
	private Long id;

	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 double getPrice() {
		return price;
	}

	public void setPrice(double price) {
		this.price = price;
	}

	public Category getCategory() {
		return category;
	}

	public void setCategory(Category category) {
		this.category = category;
	}

	private String name;

	private double price;

	@ManyToOne
	private Category category;
}
```

DemoService.java

```
import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import static com.mysema.demo.QProduct.product;

import com.querydsl.jpa.impl.JPAQuery;

public class DemoService {

	public List<Product> findProductsByNameAndCategoryId(String name, Long categoryId){
		EntityManagerFactory entityManagerFactory = Persistence.createEntityManagerFactory
("persistence");
	EntityManager entityManager = entityManagerFactory.createEntityManager();

		QCategory cat = QCategory.category;
		JPAQuery qry = new JPAQuery(entityManager);

		createQuery(QProduct.product,qry);

		if(name != null){
			qry.where(product.name.like(name));
		}

		if(categoryId != null){
			qry.where(product.category.catId.eq(categoryId));
		}

		return qry.fetch();
	}

	private JPAQuery createQuery(QProduct product,JPAQuery qr) {
		return (JPAQuery) qr.from(product);
	}

}
```

The following links are created with above code when the Spring Data extension is used:



Function Point, Quality and Sizing support

This extension provides the following support:

Function Points (transactions)	Quality and Sizing
✅	❌

CAST AIP compatibility

This extension is compatible with:

CAST AIP release	Supported
8.3.x	✓
8.2.x	✓

Supported DBMS servers

This extension is compatible with the following DBMS servers:

CAST AIP release	CSS	Oracle	Microsoft
All supported releases (see above)	✓	✓	✗

Prerequisites

✓	An installation of any compatible release of CAST AIP (see table above)

Dependencies with other extensions

Some CAST extensions require the presence of other CAST extensions in order to function correctly. The SPRING-DATA extension requires that the following other CAST extensions are also installed:

- **com.castsoftware.internal.platform** (internal technical extension).

Note that when using the **CAST Extension Downloader** to download the extension and the **Manage Extensions** interface in **CAST Server Manager** to install the extension, any dependent extensions are **automatically** downloaded and installed for you. You do not need to do anything.

Download and installation instructions

Please see:

- Download an extension
- Install an extension

The latest release status of this extension can be seen when downloading it from the CAST Extend server.

Packaging, delivering and analyzing your source code

Once the extension is installed, no further configuration changes are required before you can package your source code and run an analysis.

You can refer to the existing official CAST AIP documentation for more information about the CAST Delivery Manager Tool packaging and delivery process - see: Delivery.

What results can you expect?

This extension will create the links between objects that are created by the analyzer itself. Also it will create the unknown table object if the db table object is not found.

Objects

The following objects are displayed in CAST Enlighten:

Icon	Metamodel description
	**SpringData Unknown Table**