JAKARTA EE

Jakarta Data

1.0.0-b3, July 25, 2023: Draft

Table of Contents

```
Specification: Jakarta Data

Version: 1.0.0-b3

Status: Draft

Release: July 25, 2023
```

Jakarta Data

Chapter 1. Introduction

The Jakarta Data specification provides an API for easier data access. A Java developer can split the persistence from the model with several features, such as the ability to compose custom query methods on a Repository interface where the framework will implement it.

There is no doubt about the importance of data around the application. We often talk about a stateless application, where we delegate the application's state to the database.

Dealing with a database is one of the biggest challenges within a software architecture. In addition to choosing one of several options on the market, it is necessary to consider the persistence integrations. Jakarta Data makes life easier for Java developers.

1.1. Goals

Jakarta Data works in a tight integration between Java and a persistence layer, where it has the following specification goals:

- Be a persistence agnostic API. Therefore, through abstractions, it will connect different types of databases and storage sources.
- Be a pluggable and extensible API. Even when the API won't support a particular behavior of a storage engine, it might provide an extensible API to make it possible.

1.2. Non-Goals

As with any software component, these decisions come with trade-offs and the following non-goals:

- Provide specific features of Jakarta Persistence, Jakarta NoSQL, etc. Those APIs have their own specifications.
- Replace the Jakarta Persistence or Jakarta NoSQL specifications. Indeed, Jakarta Data will work as a complement to these specifications as an agnostic API.

1.3. Conventions

1.4. Jakarta Data Project Team

This specification is being developed as part of Jakarta Data project under the Jakarta EE Specification Process. It is the result of the collaborative work of the project committers and various contributors.

1.4.1. Project Leads

- Nathan Rauh
- Otavio Santana

1.4.2. Committers

- Denis Stepanov
- Dmitry Kornilov
- Emily Jiang
- Graeme Rocher
- James Krueger
- James Stephens
- Michael Redlich
- Nathan Rauh
- Otavio Santana
- Werner Keil

1.4.3. Mentor

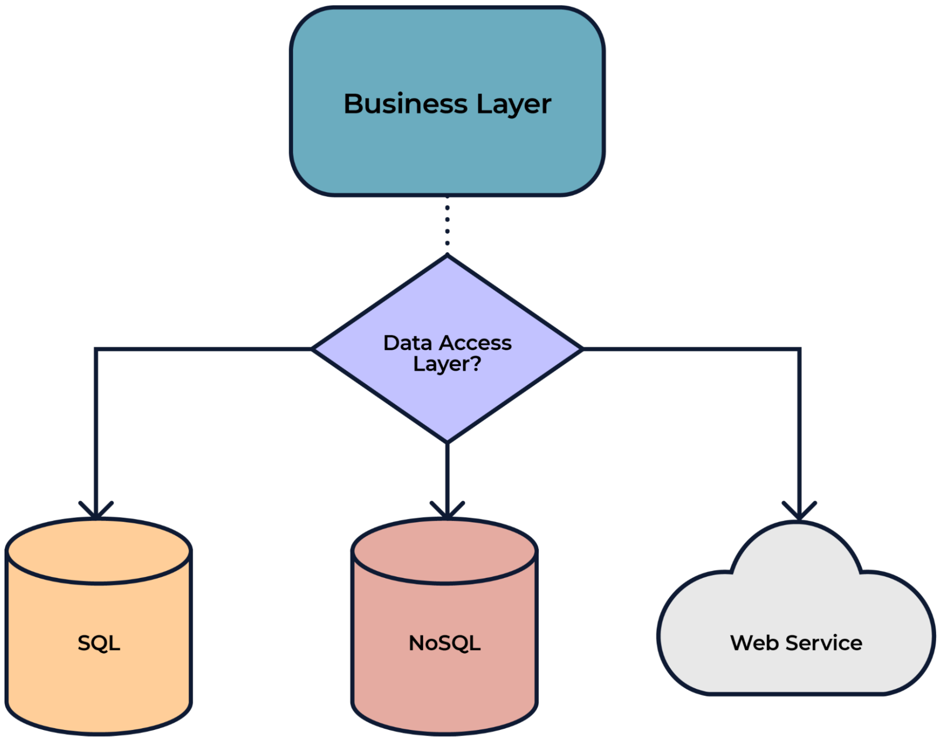
- Dmitry Kornilov

1.4.4. Contributors

The complete list of Jakarta Data contributors may be found here.

Chapter 2. Repository

In Domain-Driven Design (DDD) the repository pattern encapsulates the logic required to access data sources. The repository pattern consolidates data access functionality, providing better maintainability and decoupling the infrastructure or technology used to access databases from the domain model layer.



This pattern focuses on the closest proximity of entities and hides where the data comes from.

The Repository pattern is a well-documented way of working with a data source. In the book Patterns of Enterprise Application Architecture, Martin Fowler describes a repository as follows:

> A repository performs the tasks of an intermediary between the domain model layers and data mapping, acting in a similar way to a set of domain objects in memory. Client objects declaratively build queries and send them to the repositories for answers. Conceptually, a repository encapsulates a set of objects stored in the database and operations that can be performed on them, providing a way that is closer to the persistence layer. Repositories also support the purpose of separating, clearly and in one direction, the dependency between the work domain and the data allocation or mapping.

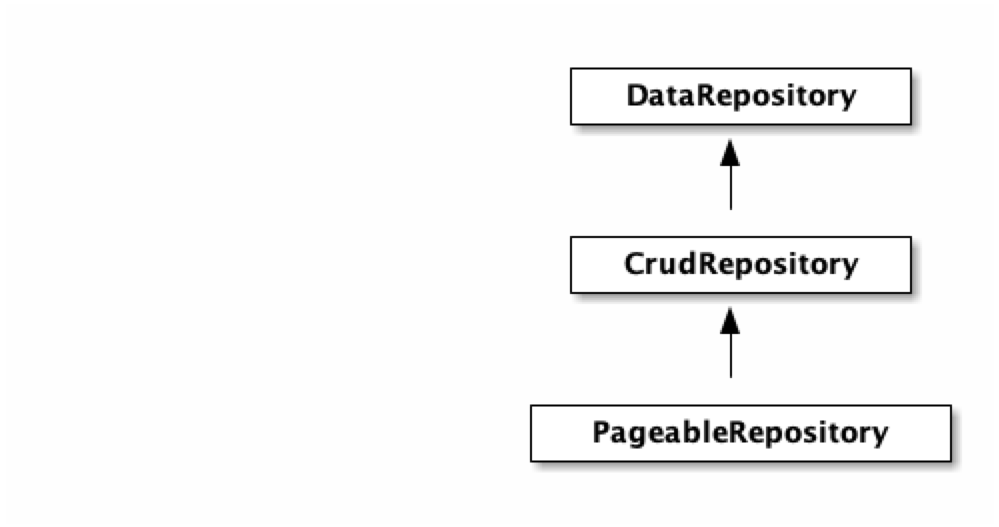
It also becomes very famous in Domain-Driven Design: Tackling Complexity in the Heart of Software by Eric Evans.

2.1. Repositories on Jakarta Data

A repository abstraction aims to significantly reduce the boilerplate code required to implement data access layers for various persistence stores.

The parent interface in Jakarta Data repository abstraction is DataRepository.

By default, Jakarta Data has support for two interfaces. However, the core is extensible. Therefore, a provider might extend one or more interfaces to a specific data target.



- Interface with generic CRUD operations on a repository for a specific type. This one we can see more often on several Java implementations.
- Interface with generic CRUD operations using the pagination feature.

From the Java developer perspective, create an interface that is annotated with the `@Repository` annotation and optionally extends one of the built-in repository interfaces.

So, given a `Product` entity where the ID is a `long` type, the repository would be:

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

}
```

There is no nomenclature restriction to make mandatory the `Repository` suffix. Such as, you might represent the repository of the Car's entity as a `Garage` instead of `CarRepository`.

```java
@Repository
public interface Garage extends CrudRepository<Car, String> {

}
```

2.2. Entity Classes

Entity classes are simple Java objects with fields or accessor methods designating each entity property.

You may use `jakarta.persistence.Entity` and the corresponding entity-related annotations of the Jakarta Persistence specification in the same package (such as `jakarta.persistence.Id` and `jakarta.persistence.Column`) to define and customize entities for relational databases.

You may use `jakarta.nosql.Entity` and the corresponding entity-related annotations of the Jakarta NoSQL specification in the same package (such as `jakarta.nosql.Id` and `jakarta.nosql.Column`) to define and customize entities for NoSQL databases.

Applications are recommended not to mix Entity annotations from different models for the sake of clarity and to allow for the Entity annotation to identify which provider is desired in cases where multiple types of Jakarta Data providers are available.

Repository implementations will search for the Entity annotation(s) they support and ignore other annotations.

2.3. Query Methods

In Jakarta Data, besides finding by an ID, custom queries can be written in two ways:

- `@Query` annotation: Defines a query string in the annotation.
- Query by method name: Defines a query based on naming convention used in the method name.

Due to the variety of data sources, those resources might not work; it varies based on the Jakarta Data implementation and the database engine, which can provide queries on more than a Key or ID or not, such as a Key-value database.

2.3.1. Using the Query Annotation

The `@Query` annotation supports providing a search expression as a String. The specification does not define the query syntax, which may vary between vendors and data sources, such as SQL, JPQL, Cypher, CQL, etc.

```
@Repository
public interface ProductRepository extends CrudRepository<Product, Long> {
  @Query("SELECT p FROM Products p WHERE p.name=?1")  // example in JPQL
  Optional<Product> findByName(String name);
}
```

Jakarta Data also includes the `@Param` annotation to define a binder annotation, where as with the query expression, each vendor will express the syntax freely such as `?`, `@`, etc..

```
@Repository
public interface ProductRepository extends CrudRepository<Product, Long> {
  @Query("SELECT p FROM Products p WHERE p.name=:name")  // example in JPQL
  Optional<Product> findByName(@Param("name") String name);
}
```

2.3.2. Query by Method

The Query by method mechanism allows for creating query commands by naming convention.

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

  List<Product> findByName(String name);

  @OrderBy("price")
  List<Product> findByNameLike(String namePattern);

  @OrderBy(value = "price", descending = true)
  List<Product> findByNameLikeAndPriceLessThan(String namePattern, float priceBelow);

}
```

The parsing query method name has two parts: the subject and the property.

The first part defines the query's subject or condition, and the second the condition value; both forms the predicate.

A predicate can refer only to a direct property of the managed entity. We also have the option to handle entities with another class on them.

2.3.3. Entity Property Names

Within an entity, property names must be unique ignoring case. For simple entity properties, the field or accessor method name serves as the entity property name. In the case of embedded classes, entity property names are computed by concatenating the field or accessor method names at each level.

Assume an Order entity has an Address with a ZipCode. In that case, the access is `order.address.zipCode`. This form is used within annotations, such as `@Query`.

```
@Repository
public interface OrderRepository extends CrudRepository<Order, Long> {

  @Query("SELECT order FROM Order order WHERE order.address.zipCode=?1")
  List<Order> withZipCode(ZipCode zipCode);

}
```

For queries by method name, the resolution algorithm starts by interpreting the whole part (AddressZipCode) as the property and checks the domain class for a property with that name (uncapitalized). If the algorithm succeeds, it uses that property.

```
@Repository
public interface OrderRepository extends CrudRepository<Order, Long> {

  Stream<Order> findByAddressZipCode(ZipCode zipCode);

}
```

Although this should work for most cases, to resolve this ambiguity, you can use `_` inside your method name to manually define traversal points.

```
@Repository
public interface OrderRepository extends CrudRepository<Order, Long> {

  Stream<Order> findByAddress_ZipCode(ZipCode zipCode);

}
```

Define as a priority following standard Java naming conventions, camel case, using underscore as the last resort.

In queries by method name, `Id` is an alias for the entity property that is designated as the id. Entity property names that are used in queries by method name must not contain reserved words.

2.3.3.1. Query Methods Keywords

The following table lists the subject keywords generally supported by Jakarta Data.

Keyword	Description
findBy	General query method returning the repository type.
deleteBy	Delete query method returning either no result (void) or the delete count.
countBy	Count projection returning a numeric result.
existsBy	Exists projection, returning typically a `boolean` result.

Jakarta Data implementations support the following list of predicate keywords to the extent that the database is capable of the behavior. A repository method will raise `jakarta.data.exceptions.DataException` or a more specific subclass of the exception if the database does not provide the requested functionality.

Keyword	Description	Method signature Sample
And	The `and` operator.	findByNameAndYear

Keyword	Description	Method signature Sample
Or	The `or` operator.	findByNameOrYear
Between	Find results where the property is between the given values	findByDateBetween
Empty	Find results where the property is an empty collection or has a null value.	deleteByPendingTasksEmpty
LessThan	Find results where the property is less than the given value	findByAgeLessThan
GreaterThan	Find results where the property is greater than the given value	findByAgeGreaterThan
LessThanEqual	Find results where the property is less than or equal to the given value	findByAgeLessThanEqual
GreaterThanEqual	Find results where the property is greater than or equal to the given value	findByAgeGreaterThanEqual
Like	Finds string values "like" the given expression	findByTitleLike
IgnoreCase	Requests that string values be compared independent of case for query conditions and ordering.	findByStreetNameIgnoreCaseLike
In	Find results where the property is one of the values that are contained within the given list	findByIdIn
Null	Finds results where the property has a null value.	findByYearRetiredNull
True	Finds results where the property has a boolean value of true.	findBySalariedTrue
False	Finds results where the property has a boolean value of false.	findByCompletedFalse
OrderBy	Specify a static sorting order followed by the property path and direction of ascending.	findByNameOrderByAge
OrderBy____Desc	Specify a static sorting order followed by the property path and direction of descending.	findByNameOrderByAgeDesc

Keyword	Description	Method signature Sample
OrderBy____Asc	Specify a static sorting order followed by the property path and direction of ascending.	findByNameOrderByAgeAsc
OrderBy____(Asc|Desc)*(Asc|Desc)	Specify several static sorting orders	findByNameOrderByAgeAscNameDescYearAsc

Logical Operator Precedence

For relational databases, the logical operator `And` takes precedence over `Or`, meaning that `And` is evaluated on conditions before `Or` when both are specified on the same method. For other database types, the precedence is limited to the capabilities of the database. For example, some graph databases are limited to precedence in traversal order.

2.4. Special Parameter Handling

Jakarta Data also supports particular parameters to define pagination and sorting.

Jakarta Data recognizes, when specified on a repository method after the query parameters, specific types, like `Limit`, `Pageable`, and `Sort`, to dynamically apply limits, pagination, and sorting to queries. The following example demonstrates these features:

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

  List<Product> findByName(String name, Pageable pageable);

  List<Product> findByNameLike(String pattern, Limit max, Sort... sorts);

}
```

You can define simple sorting expressions by using property names.

```
Sort name = Sort.asc("name");
```

You can combine sorting with a starting page and maximum page size by using property names.

```
Pageable pageable = Pageable.ofSize(20).page(1).sortBy(Sort.desc("price"));
first20 = products.findByNameLike(name, pageable);
```

2.5. Precedence of Sort Criteria

The specification defines different ways of providing sort criteria on queries. This section discusses how these different mechanisms relate to each other.

2.5.1. Sort Criteria within Query Language

Sort criteria can be hard-coded directly within query language by making use of the @Query annotation. A repository method that is annotated with @Query with a value that contains an ORDER BY clause (or query language equivalent) must not provide sort criteria via the other mechanisms.

A repository method that is annotated with @Query with a value that does not contain an ORDER BY clause and ends with a WHERE clause (or query language equivalents to these) can use other mechanisms that are defined by this specification for providing sort criteria.

2.5.2. Static Mechanisms for Sort Criteria

Sort criteria is provided statically for a repository method by using the OrderBy keyword or by annotating the method with one or more @OrderBy annotations. The OrderBy keyword cannot be intermixed with the @OrderBy annotation or the @Query annotation. Static sort criteria takes precedence over dynamic sort criteria in that static sort criteria is evaluated first. When static sort criteria sorts entities to the same position, dynamic sort criteria is applied to further order those entities.

2.5.3. Dynamic Mechanisms for Sort Criteria

Sort criteria is provided dynamically to repository methods either via Sort parameters or via a Pageable parameter that has one or more Sort values. Sort and Pageable containing Sort must not both be provided to the same method.

2.5.4. Examples of Sort Criteria Precedence

The following examples work through scenarios where static and dynamic sort criteria are provided to the same method.

```java
// Sorts first by type. When type is the same, applies the Pageable's sort criteria
Page<User> findByNameStartsWithOrderByType(String namePrefix, Pageable pagination);

// Sorts first by type. When type is the same, applies the criteria in the Sorts
List<User> findByNameStartsWithOrderByType(String namePrefix, Sort... sorts);

// Sorts first by age. When age is the same, applies the Pageable's sort criteria
@OrderBy("age")
Page<User> findByNameStartsWith(String namePrefix, Pageable pagination);

// Sorts first by age. When age is the same, applies the criteria in the Sorts
@OrderBy("age")
List<User> findByNameStartsWith(String namePrefix, Sort... sorts);

// Sorts first by name. When name is the same, applies the Pageable's sort criteria
@Query("SELECT u FROM User u WHERE (u.age > ?1)")
@OrderBy("name")
KeysetAwarePage<User> olderThan(int age, Pageable pagination);
```

2.6. Keyset Pagination

Keyset pagination aims to reduce missed and duplicate results across pages by querying relative to the observed values of entity properties that constitute the sorting criteria. Keyset pagination can also offer an improvement in performance because it avoids fetching and ordering results from prior pages by causing those results to be non-matching. A Jakarta Data provider appends additional conditions to the query and tracks keyset values automatically when `KeysetAwareSlice` or `KeysetAwarePage` are used as the repository method return type. The application invokes `nextPageable` or `previousPageable` on the keyset aware slice or page to obtain a `Pageable` which keeps track of the keyset values.

For example,

```
@Repository
public interface CustomerRepository extends CrudRepository<Customer, Long> {
  KeysetAwareSlice<Customer> findByZipcodeOrderByLastNameAscFirstNameAscIdAsc(
                                  int zipcode, Pageable pageable);
}
```

You can obtain the next page with,

```
for (Pageable p = Pageable.ofSize(50); p != null; ) {
  page = customers.findByZipcodeOrderByLastNameAscFirstNameAscIdAsc(55901, p);
  ...
  p = page.nextPageable();
}
```

Or you can obtain the next (or previous) page relative to a known entity,

```
Customer c = ...
Pageable p = Pageable.ofSize(50).afterKeyset(c.lastName, c.firstName, c.id);
page = customers.findByZipcodeOrderByLastNameAscFirstNameAscIdAsc(55902, p);
```

The sort criteria for a repository method that performs keyset pagination must uniquely identify each entity and must be provided by:

- `OrderBy` name pattern of the repository method (as in the examples above) or `@OrderBy` annotation(s) on the repository method.
- `Sort` parameters of the `Pageable` that is supplied to the repository method.

2.6.1. Example of Appending to Queries for Keyset Pagination

Without keyset pagination, a Jakarta Data provider that is based on Jakarta Persistence might compose the following JPQL for the `findByZipcodeOrderByLastNameAscFirstNameAscIdAsc` repository method from the prior example:

```
SELECT o FROM Customer o WHERE (o.zipCode = ?1)
                         ORDER BY o.lastName ASC, o.firstName ASC, o.id ASC
```

When keyset pagination is used, the keyset values from the `Cursor` of the `Pageable` are available as query parameters, allowing the Jakarta Data provider to append additional query conditions. For example,

```
SELECT o FROM Customer o WHERE (o.zipCode = ?1)
                           AND (   (o.lastName > ?2)
                                OR (o.lastName = ?2 AND o.firstName > ?3)
                                OR (o.lastName = ?2 AND o.firstName = ?3 AND o.id >
?4)
                               )
                         ORDER BY o.lastName ASC, o.firstName ASC, o.id ASC
```

2.6.2. Avoiding Missed and Duplicate Results

Because searching for the next page of results is relative to a last known position, it is possible with keyset pagination to allow some types of updates to data while pages are being traversed without causing missed results or duplicates to appear. If you add entities to a prior position in the traversal of pages, the shift forward of numerical position of existing entities will not cause duplicates entities to appear in your continued traversal of subsequent pages because keyset pagination does not query based on a numerical position. If you remove entities from a prior position in the traversal of pages, the shift backward of numerical position of existing entities will not cause missed entities in your continued traversal of subsequent pages because keyset pagination does not query based on a numerical position.

Other types of updates to data, however, will cause duplicate or missed results. If you modify entity properties which are used as the sort criteria, keyset pagination cannot prevent the same entity from appearing again or never appearing due to the altered values. If you add an entity that you previously removed, whether with different values or the same values, keyset pagination cannot prevent the entity from being missed or possibly appearing a second time due to its changed values.

2.6.3. Restrictions on use of Keyset Pagination

- The repository method signature must return `KeysetAwareSlice` or `KeysetAwarePage`. A repository method with return type of `KeysetAwareSlice` or `KeysetAwarePage` must raise `UnsupportedOperationException` if the database is incapable of keyset pagination.
- The repository method signature must accept a `Pageable` parameter.
- Sort criteria must be provided and should be minimal.
- The combination of provided sort criteria must uniquely identify each entity.
- Page numbers for keyset pagination are estimated relative to prior page requests or the observed absence of further results and are not accurate. Page numbers must not be relied upon when using keyset pagination.

- Page totals and result totals are not accurate for keyset pagination and must not be relied upon.
- A next or previous page can end up being empty. You cannot obtain a next or previous `Pageable` from an empty page because there are no keyset values relative to which to query.
- A repository method that is annotated with `@Query` and performs keyset pagination must omit the `ORDER BY` clause from the provided query and instead must supply the sort criteria via `@OrderBy` annotations or `Sort` parameters of `Pageable`. The provided query must end with a `WHERE` clause to which additional conditions can be appended by the Jakarta Data provider. The Jakarta Data provider is not expected to parse query text that is provided by the application.

2.6.4. Keyset Pagination Example with Sorts

Here is an example where an application uses `@Query` to provide a partial query to which the Jakarta Data provider can generate and append additional query conditions and an `ORDER BY` clause.

```
@Repository
public interface CustomerRepository extends CrudRepository<Customer, Long> {
  @Query("SELECT o FROM Customer o WHERE (o.totalSpent / o.totalPurchases > ?1)")
  KeysetAwareSlice<Customer> withAveragePurchaseAbove(float minimum, Pageable
pagination);
}
```

Example traversal of pages:

```
for (Pageable p = Pageable.ofSize(25).sortBy(Sort.desc("yearBorn"), Sort.asc("name"),
Sort.asc("id")));
     p != null; ) {
  page = customers.withAveragePurchaseAbove(50.0f, p);
  ...
  p = page.nextPageable();
}
```

Chapter 3. Interoperability with other Jakarta EE Specifications

In this section, we will delve into the robust capabilities of Jakarta EE Data and its seamless integration with other Jakarta EE specifications. This integration offers a comprehensive data persistence and management solution in Java applications. When operating within a Jakarta EE product, the availability of other Jakarta EE Technologies depends on the profile. This section outlines how related technologies from other Jakarta EE Specifications work together with Jakarta Data to achieve interoperability.

3.1. Jakarta Context Dependency Injection

Contexts and Dependency Injection (CDI) is a core specification in Jakarta EE that provides a powerful and flexible dependency injection framework for Java applications. CDI lets you decouple components and manage their lifecycle through dependency injection, enabling loose coupling and promoting modular and reusable code.

One of the critical aspects of CDI integration with Jakarta EE Data is the ability to create repository instances using the `@Inject` annotation effortlessly. Repositories are interfaces that define data access and manipulation operations for a specific domain entity. Let's take an example to illustrate this integration:

```java
@Repository
public interface CarRepository extends CrudRepository<Car, Long> {

  List<Car> findByType(CarType type);

  Optional<Car> findByName(String name);

}
```

In the above example, we have a `CarRepository` interface that extends the `CrudRepository` interface provided by Jakarta EE Data. The `CrudRepository` interface offers a set of generic CRUD (Create, Read, Update, Delete) operations for entities.

By annotating the `CarRepository` interface with `@Repository`, we indicate that it is a repository component managed by the Jakarta EE Data framework. It allows the Jakarta Data provider to automatically generate the necessary implementations for the defined methods, saving us from writing boilerplate code.

Now, with CDI and the `@Inject` annotation, we can easily inject the CarRepository instance into our code and utilize its methods:

```java
@Inject
CarRepository repository;
```

```
// ...

Car ferrari = Car.id(10L).name("Ferrari").type(CarType.SPORT);
repository.save(ferrari);
```

In the above snippet, we inject the `CarRepository` instance into our code using the `@Inject` annotation. Once injected, we can use the repository object to invoke various data access methods provided by the `CarRepository` interface, such as `save()`, `findByType()`, and `findByName()`. This integration between CDI and Jakarta EE Data allows for seamless management of repository instances and enables easy data access and manipulation within your Jakarta EE applications.

3.1.1. CDI Extension

Jakarta Data and CDI integrate through CDI extensions, which provide a mechanism to customize and enhance the behavior of CDI in your Jakarta EE application. Jakarta Data integration with Jakarta CDI applies to both CDI full and CDI Lite.

CDI Full Integration:

For CDI Full integration, Jakarta Data providers must implement the `jakarta.enterprise.inject.spi.Extension` interface to produce the bean instances defined via the `@Repository` annotation and injected using the `@Inject` annotation. By implementing this interface, Jakarta Data providers can define custom logic to create and manage the repository instances within the CDI container.

The CDI specification employs strategies to reduce conflicts between Jakarta EE Data providers. The primary approach is based on the entity annotation class, which ensures that different providers can coexist harmoniously within the same application. The secondary strategy is based on the Jakarta EE Data provider name, which further helps avoid conflicts and provides flexibility in choosing the desired provider for a specific repository.

CDI Lite Integration:

CDI Lite is a lightweight version of CDI that focuses on compile-time exploration and build-time extensions. For CDI Lite integration, Jakarta Data providers must implement the `jakarta.enterprise.inject.build.compatible.spi.BuildCompatibleExtension` interface to produce the bean instances defined via the `@Repository` annotation and injected using the `@Inject` annotation.

CDI Lite leverages compile-time and build-time exploration to ensure compatibility and reduce conflicts between Jakarta EE Data providers. The primary strategy for conflict reduction is similar to CDI Full and is based on the entity annotation class. The secondary approach relies on the Jakarta EE Data provider name, providing additional flexibility in choosing the appropriate provider for a given repository.

CDI Full and CDI Lite approaches enable seamless integration between CDI and Jakarta EE Data. They provide mechanisms to produce and manage repository instances, ensuring that the desired implementation is available for injection when using the `@Inject` annotation.

By leveraging CDI extensions and the provided strategies, Jakarta Data providers effectively integrate Jakarta EE Data with CDI and take advantage of its powerful dependency injection capabilities, promoting modular and reusable code.

3.2. Mapping an Entity

A Jakarta Data provider is an implementation of Jakarta Data for one or more types of entities. An entity refers to a class that represents objects in a storage engine, such as SQL or NoSQL databases.

The `jakarta.persistence.Entity` annotation from the Jakarta Persistence specification can be used by repository entity classes for Jakarta Data providers that are backed by a Jakarta Persistence provider. Other Jakarta Data providers must not support the `jakarta.persistence.Entity` annotation.

The `jakarta.nosql.Entity` annotation from the Jakarta NoSQL specification can be used by repository entity classes for Jakarta Data providers that are backed by NoSQL databases. Other Jakarta Data providers must not support the `jakarta.nosql.Entity` annotation.

Jakarta Data providers that define custom entity annotations must follow the convention that the class name of all supported entity annotation types ends with `Entity`. This enables Jakarta Data providers to identify if a repository entity class contains entity annotations from different Jakarta Data providers so that the corresponding `Repository` can be ignored by Jakarta Data providers that should not provide it.

Jakarta Data provider CDI Extensions must ignore all `Repository` annotations where annotations for the corresponding entity are available at run time and none of the entity annotations are supported by the Jakarta Data provider. Ignoring these `Repository` annotations allows other Jakarta Data providers to handle them.

3.2.1. Jakarta Data Provider Name

The entity annotation class is the primary strategy to avoid conflicts between Jakarta Data providers. In most cases, it is sufficient to differentiate between providers. However, there may be situations where more than the entity annotation class is needed. In such cases, the application can specify the name of the desired Jakarta Data provider using the optional `provider` attribute of the `Repository` annotation.

To ensure compatibility and prevent conflicts, CDI extensions of Jakarta Data providers must disregard any `Repository` annotations that specify a different provider's name through the `Repository.provider()` attribute. By ignoring these annotations, CDI extensions allow other Jakarta Data providers to handle them appropriately and avoid any conflicts that may arise.

3.3. Jakarta Transactions Usage

When running in an environment where Jakarta Transactions is available and a global transaction is active on the thread of execution for a repository operation and the data source backing the repository is capable of transaction enlistment, the repository operation enlists the data source resource as a participant in the transaction. The repository operation does not commit or roll back the transaction that was already present on the thread, but it might cause the transaction to be marked as rollback only (`jakarta.transaction.Status.STATUS_MARKED_ROLLBACK`) if the repository operation fails.

When running in an environment where Jakarta Transactions and Jakarta CDI are available, a repository method can be annotated with the `jakarta.transaction.Transactional` annotation, which is applied to the execution of the repository method.

3.4. Interceptor Annotations on Repository Methods

When a repository method is annotated with an interceptor binding annotation, the interceptor is bound to the repository bean according to the interceptor binding annotation of the repository interface method, causing the bound interceptor to be invoked around the repository method when it runs. This enables the use of interceptors such as `jakarta.transaction.Transactional` on repository methods when running in an environment where the Jakarta EE technology that provides the interceptor is available.

3.5. Jakarta Persistence

When integrating Jakarta Data with Jakarta Persistence, developers can leverage the JPA annotations to define the mapping of entities in repositories. Entities in Jakarta Persistence are typically annotated with `jakarta.persistence.Entity` to indicate their persistence capability.

A Jakarta Data provider that supports Jakarta Persistence allows you to define repositories for classes marked with the `jakarta.persistence.Entity` annotation.

By supporting Jakarta Persistence annotations, Jakarta Data providers enable Java developers to utilize familiar and standardized mapping techniques when defining entities in repositories, ensuring compatibility and interoperability with the respective technologies.

3.6. Jakarta NoSQL

When integrating Jakarta Data with Jakarta NoSQL, developers can use the NoSQL annotations to define the mapping of entities in repositories. Entities in Jakarta NoSQL are typically annotated with `jakarta.nosql.Entity` to indicate their suitability for persistence in NoSQL databases.

A Jakarta Data provider that supports Jakarta NoSQL will scan classes marked with the `jakarta.nosql.Entity` annotation.

By supporting Jakarta NoSQL annotations, Jakarta Data providers enable Java developers to utilize familiar and standardized mapping techniques when defining entities in repositories, ensuring compatibility and interoperability with the respective technologies.

3.7. Jakarta Bean Validation

Integrating with Jakarta Validation ensures data consistency within the Java layer. By applying validation rules to the data, developers can enforce constraints and business rules, preventing invalid or inconsistent information from being processed or persisted.

Incorporating Jakarta Validation into your Jakarta Data applications helps enforce data consistency at the Java layer. When you invoke the `save` or `saveAll` method on repositories, the Jakarta Data provider ensures that the validation rules defined in the entity are automatically executed prior to updating the database if a Jakarta Validation provider is present. For instance, when persisting a `Student` object, the `name` field will be validated to ensure it is not blank.

Using Jakarta Validation brings several advantages. It helps maintain data integrity, improves data quality, and enhances the reliability of the application. Catching validation errors early in the Java layer can identify and resolve potential issues before further processing or persistence occurs. Additionally, Jakarta Validation allows for declarative validation rules, simplifying the validation logic and promoting cleaner and more maintainable code.

The following code snippet demonstrates the usage of Jakarta Validation annotations in the `Student` entity class:

```java
@Schema(name = "Student")
@Entity
public class Student {

    @Id
    private String id;

    @Column
    @NotBlank
    private String name;

    @Positive
    @Min(18)
    @Column
    private int age;
}
```

In this example, the `name` field is annotated with `@NotBlank`, indicating that it must not be blank. The `age` field is annotated with both `@Positive` and `@Min(18)`, ensuring it is a positive integer greater than or equal to 18.

The `School` repository interface, shown below, utilizes the validation rules defined in the `Student` entity:

```java
@Repository
public interface School extends PageableRepository<Student, String> {

}
```

In the Jakarta Data specification, when integrating with Jakarta Bean Validation, the validation process must occur for `save` and `saveAll` methods of Repository interfaces prior to making inserts or updates in the database. It means that when using the save methods to perform an insert or update operation on a database, Jakarta Bean Validation will execute the defined constraints to validate the data being inserted or updated.

Regarding the delete and find operations, the integration with Jakarta Bean Validation can be included or made optional by the provider. It means that the validation process for delete and find operations may or may not be applied, depending on the configuration or implementation of a provider. It is up to the provider to determine whether or not to include validation for these database operations.