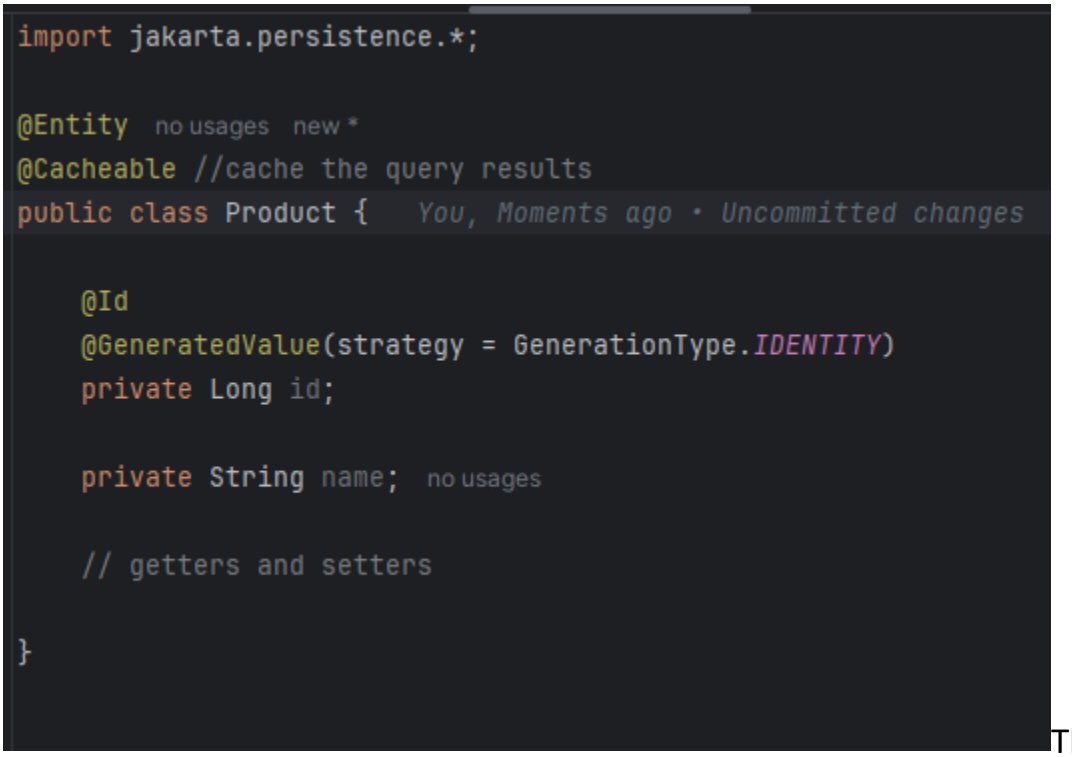
Caching

Caching is a technique that stores frequently accessed data in memory. This can improve performance by reducing the number of times that the database needs to be accessed. Spring Data JPA provides a number of caching options, including:

- **Entity Cache:** The entity cache stores entities in memory. This can improve performance by reducing the number of times that the database needs to be queried to load an entity.
- **Query Cache:** The query cache stores the results of queries in memory. This can improve performance by reducing the number of times that the database needs to be executed to execute a query.

Using the Entity Cache

The following code example shows how to use the entity cache to cache the results of a query:



The **@Cacheable** annotation tells Spring Data JPA to cache the results of the query for the Product entity. This means that the next time the query is executed, Spring

Data JPA will check the cache to see if the results are already available. If the results are available in the cache, Spring Data JPA will return the results from the cache instead of executing the query again.

Using the Query Cache

The following code example shows how to use the query cache to cache the results of a query:

```
@Query("SELECT p FROM Product p WHERE p.name = :name") //caches the result of the query
List<Product> findProductsByName(@Param("name") String name);
```

The @Query annotation tells Spring Data JPA to cache the results of the query. This means that the next time the query is executed, Spring Data JPA will check the cache to see if the results are already available. If the results are available in the cache, Spring Data JPA will return the results from the cache instead of executing the query again.

@CacheEvict

The **@CacheEvict** annotation is used to evict one or more entries from a cache. When a method annotated with **@CacheEvict** is called, Spring will remove the cached data associated with the specified cache name and key (or keys) along with the method execution.

Key features of the **@CacheEvict** annotation:

- It can be used to evict all entries from a cache, or a specific subset of entries.
- It can be used to evict entries from a single cache, or multiple caches.
- It can be used to evict entries based on their key, or based on the method arguments.

- It can be used to evict entries unconditionally, or conditionally based on the return value of the method.

Benefits of Using Caching

- **Improved performance:** Caching can improve performance by reducing the number of times that the database needs to be accessed.
- **Reduced load on the database:** Caching can reduce the load on the database by reducing the number of queries that need to be executed.
- **Increased scalability:** Caching can help your application to scale to handle a larger number of concurrent users and transactions.

Transaction Management in Spring: ACID Properties and Declarative Transactions

1. Understanding ACID Properties in Transaction Management

ACID properties are the core principles of any transaction management system, ensuring that database transactions are processed reliably and accurately. These properties are:

- **Atomicity**: Ensures that a transaction is all-or-nothing. If any part of the transaction fails, the entire transaction is rolled back, and the database remains unchanged. For example, in a banking transaction, if transferring money from Account A to Account B fails, neither account is updated.
- **Consistency**: Ensures that a transaction brings the database from one valid state to another, maintaining all predefined rules such as constraints, cascades, and triggers. The integrity of the database must be maintained before and after the transaction.
- **Isolation**: Ensures that the operations of one transaction are isolated from other transactions. The intermediate states of a transaction are invisible to other transactions, preventing dirty reads, non-repeatable reads, and phantom reads.

- **Durability**: Ensures that once a transaction is committed, it will remain so, even in the event of a system failure. This is achieved through the use of transaction logs and other mechanisms that guarantee that committed changes are saved permanently.

2. Transaction Management in Spring Framework

Spring Framework provides a robust transaction management abstraction that can be used in any Java environment. It supports both declarative and programmatic transaction management, allowing for flexibility and ease of use.

- **Programmatic Transaction Management**: Allows developers to manage transactions in code explicitly using PlatformTransactionManager. This approach provides fine-grained control but can lead to boilerplate code and is error-prone.
- **Declarative Transaction Management**: This is the recommended approach as it separates transaction management from business logic. It is managed using the @Transactional annotation, which allows for transaction management via configuration rather than code.

3. Using @Transactional Annotation

The @Transactional annotation in Spring is a declarative approach to handle transaction management without having to explicitly manage transactions in your code. Here's how it works:

- **Basic Usage**: The annotation can be applied to classes or specific methods to demarcate transactional boundaries. When a method annotated with @Transactional is called, Spring will automatically start a transaction, commit it if the method completes successfully, or roll it back if an exception occurs.
- **Propagation**: Specifies how transactions should propagate across method calls. For example, REQUIRED (default) will join an existing transaction or create a new one if none exists.
- **Isolation Levels**: Determines the level of visibility one transaction has to the data changes made by other transactions. For example,

Isolation.READ_COMMITTED ensures that a transaction only reads committed data from other transactions.

- **Read-Only**: Indicates that the transaction is read-only, which can help optimize performance by avoiding unnecessary locks on the database.
- **Rollback Rules**: Specifies which exceptions should trigger a rollback. By default, only unchecked exceptions (subclasses of RuntimeException) cause a rollback, but this can be customized.
- **Timeouts**: Defines a maximum time (in seconds) the transaction is allowed to run before being rolled back automatically.

4. Advantages of Declarative Transactions with @Transactional

- **Simplicity**: Reduces boilerplate code and keeps business logic clean and free from transaction management concerns.
- **Consistency**: Ensures consistent transaction handling across the application with minimal configuration.
- **Flexibility**: Easily configure transaction behavior through annotations, including setting propagation, isolation, and rollback rules.

Best Practices

- **Use on Service Layer**: Typically, @Transactional is best applied to the service layer rather than the DAO or repository layer, promoting a clean separation of concerns.
- **Keep Transactions Short**: To avoid locking and performance issues, keep transactions as brief as possible, containing only the critical part of the business logic that requires consistency.
- **Handle Exceptions Properly**: Be mindful of which exceptions trigger rollbacks and manage checked exceptions in a way that aligns with transaction requirements.

By understanding and implementing ACID properties with declarative transactions in Spring, you can ensure robust, efficient, and reliable transaction management in your applications.