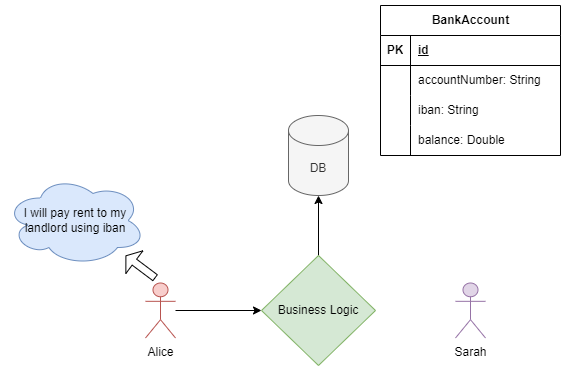
**Spring Boot Transaction Management, Propagation, Isolation Levels**

We are going to cover Transaction Management throughout this article.

Transaction Management encompasses the supervision and handling of errors and processes that may occur during a transaction.



According to this analogy, when Alice wants to pay rent to her landlord, she sends a request to the server-side business logic. The business logic then queries the database (DB) and asks, “Hey, Alice wants to pay rent to Sarah. Could you decrease the balance of Alice’s account by $100?” The DB responds with a confirmation.

This process results in a transaction being created for the payment. However, at some point in this transaction, perhaps at T4 or T7, an error occurs, indicating that Alice has already paid her rent. Unfortunately, the business logic cannot update Sarah’s account balance accordingly. For example, if Alice initially had $500, the business logic reduces her balance to $400, but the $100 is not deposited into Sarah’s account.

At this critical point, the business logic requires transaction management. It must decide whether to rollback the entire transaction, undoing the changes, or commit the transaction, confirming the changes made so far.

**@Transactional**

Spring managing transactions have provided annotation. This annotation can be defined on classes and methods. In Spring Boot, transaction management does not require a configuration. But in Spring, It requires a configuration.

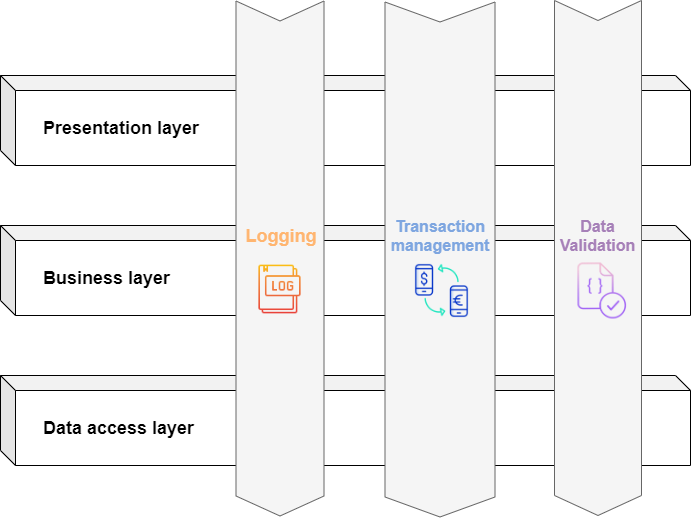
*@Transactional(propagation = Propagation.*REQUIRED*)*

**Deep Look**

**AOP (Aspect Oriented Programming)**

In Spring, transaction management is facilitated through Aspect-Oriented Programming (AOP). AOP effectively separates cross-cutting concerns from the core business logic. In a typical n-layer architecture, which includes the presentation layer (API layer), business layer (service layer), and data access layer (repository layer), cross-cutting concerns are usually defined within the business layer. These concerns encompass aspects such as security, validation, transaction management, logging, performance optimization, and other related processes. This truly enhances the clarity of the code.

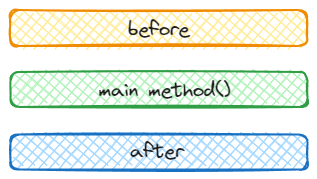
When Alice sends a request to the server-side payment method, which was already declared as a transactional method, interceptors come into play and process either before or after the business logic for transactional annotation using the proxy.



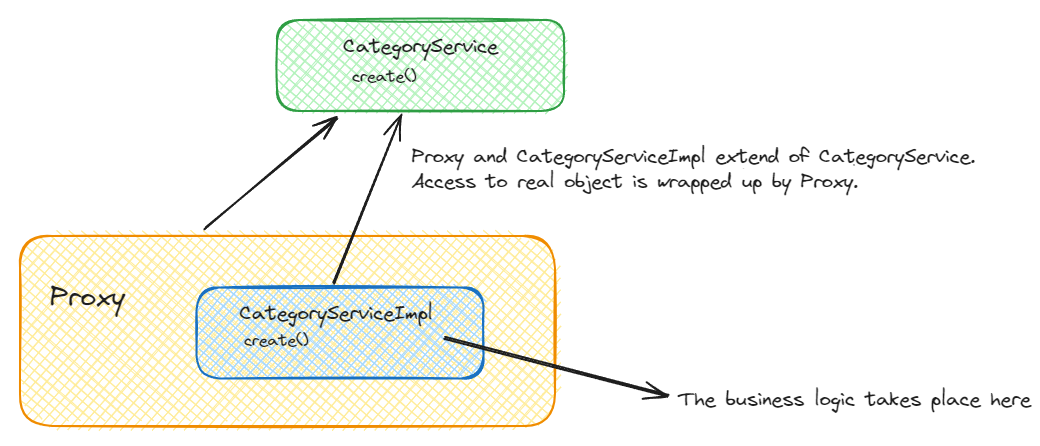
**Proxy**

Spring uses dynamic proxies, which means it creates proxies at runtime. These proxies are generated for objects marked with the @Transactional annotation. When a method with this annotation is called, it interrupts the normal flow and allows the processing of additional business logic either before or after the main logic.

The Proxy controls access to the original object, enabling processing either before or after the main object.

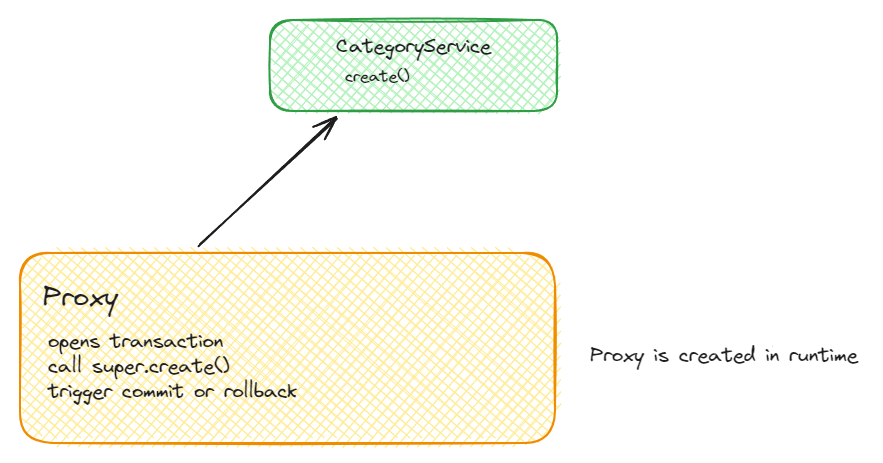


**Example of Proxy**



Typically, the application includes one service named CategoryService, which has a create method. CategoryServiceImpl and the Proxy that is created by spring in runtime extend CategoryService. This service is wrapped with a proxy, which means the proxy manages access to the real object and implements some business logic either before or after, such as rollback or commit transaction.

Spring creates these proxies in runtime. What if the application does not have an interface? If the application lacks an interface, the solution is straightforward. The Application Context extends a proxy directly from the concrete class. For example, if the application includes only a concrete class like CategoryService, the proxy simply extends CategoryService.



public interface CategoryService {  
 void create();  
}

import org.springframework.transaction.annotation.Transactional;  
  
public class CategoryServiceImpl implements CategoryService {  
  
 public CategoryServiceImpl(){  
  
 }  
  
 @Override  
 @Transactional  
 public void create() {  
 // CREATE CATEGORY  
 System.out.println("Category Service");  
 }  
}

public class ProxyCategoryService implements CategoryService {  
  
 private CategoryServiceImpl categoryService;  
  
 public ProxyCategoryService(){  
 System.out.println("Proxy Category Service");  
 }  
  
  
  
 @Override  
 public void create() {  
 if (categoryService == null){  
 categoryService = new CategoryServiceImpl();  
 }  
  
 try {  
 System.out.println("Get Transaction");  
 categoryService.create();  
 throw new RuntimeException();  
 }catch (Exception e){  
 System.out.println("Rollback Transaction");  
 }  
  
 System.out.println("Commit Transaction");  
 }  
}

public class Test {  
 public static void main(String[] args) {  
 CategoryService categoryService = new ProxyCategoryService();  
 categoryService.create();  
 }  
}

As you can see in this code, we can provide some business logic such as commit, rollback or more either before or after categoryService.create() function.

**PROPAGATION**

**REQUIRED**

REQUIRED is the default propagation type in Spring, and it includes all other transactional methods that are called after it. This means that it is required to create one session for all transactions that are called after it.

For instance, you want to save 20 lines of data. On the 18th line, the application throws an error, and the transaction calls the rollback function. Required is the most usage type.

@Transactional(propagation = Propagation.REQUIRED)  
public void create(Product product) {  
 productRepository.save(product);  
}

**REQUIRES\_NEW**

REQUIRES\_NEW This propagation type is significantly useful for most cases. If a session is initiated before REQUIRES\_NEW, it suspends the previous session and creates a new one. The previous transaction cannot impact this. It means that if an error occurs in a previous transaction, REQUIRES\_NEW will not be impacted by this error.

@Transactional(propagation = Propagation.REQUIRED)  
 public void create(Product product, List<Image> images) {  
 saveImages(images);  
 productRepository.save(product);  
   
 }  
   
 @Transactional(propagation = Propagation.REQUIRES\_NEW)  
 public void saveImages(List<Image> images) {  
 // save images   
 }

**SUPPORTS**

If a transaction is started before the SUPPORTS transactional type, It gets involved in the previous transaction. Otherwise, it continues non-transactional.

@Transactional(propagation = Propagation.SUPPORTS)  
 public void create(Product product) {  
 productRepository.save(product);  
 }

**NOT\_SUPPORTED**

NOT\_SUPPORTED from the point of call suspend all transactions and continue its way non-transactional.

When you have methods that only read data from the database but don't modify it, you can use NOT\_SUPPORTED. It allows these methods to run without a transaction, potentially improving performance.

@Transactional(propagation = Propagation.REQUIRED)  
 public void create(Product product, List<Image> images) {  
 saveImages(images);  
 productRepository.save(product);  
 }  
  
 @Transactional(propagation = Propagation.NOT\_SUPPORTED)  
 public void saveImages(List<Image> images) {  
 createImages(images);  
 }

**NEVER**

If a transaction is started before NEVER, It throws an exception. That means NEVER should use an independent process. You must use it outside of any transactional context.

**MANDATORY**

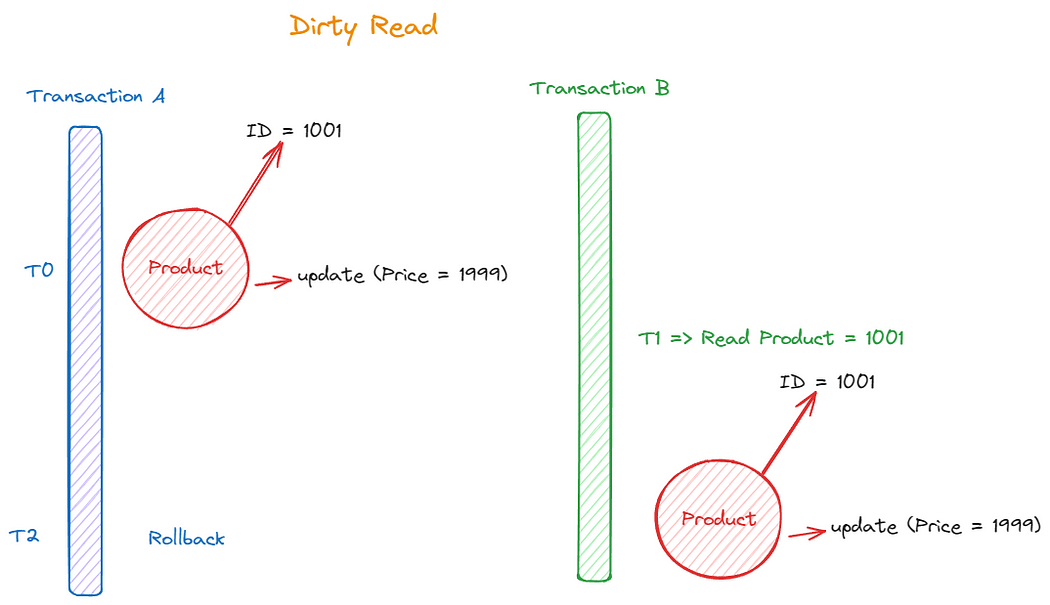
A transaction must be included before it. Otherwise, it throws an error. Always behaves like REQUIRED. When you want to ensure that a specific method is always executed within the context of a transaction, it can be crucial for maintaining data consistency and integrity. You can use of this such this scenario, for instance, payment. You want to ensure a payment method must be included in a transaction.

@Transactional(propagation = Propagation.MANDATORY)  
 public void payment(Payment payment) {  
 System.out.println("Payment");  
 }

**ISOLATION LEVEL**

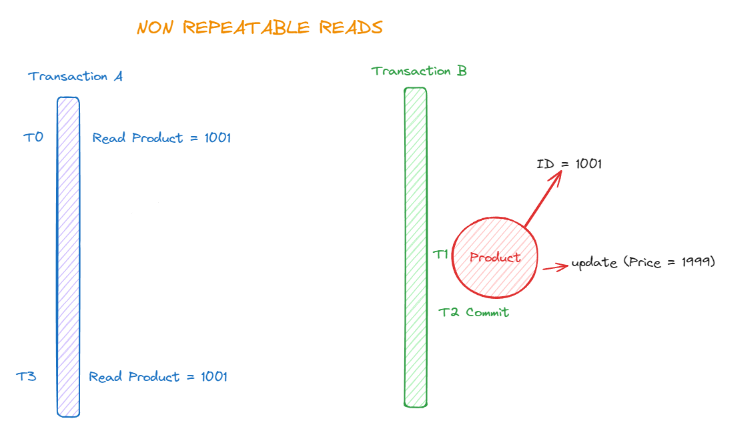
It ensures the consistency of data in simultaneous operations. And is a structure that isolates. Protects the integrity of the data.

**DIRTY READ**



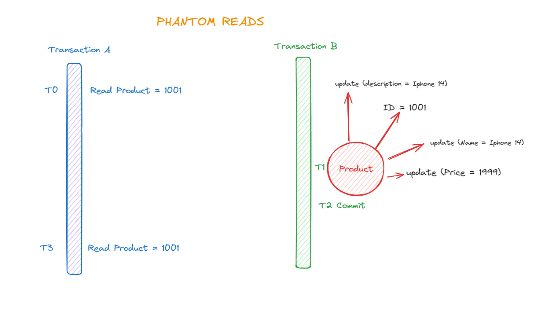
At time T0, Transaction A attempts to update the ‘price’ column of a product to $1999. At time T1, Transaction B reads the data for a product with ID 1001. By time T2, Transaction A encounters an exception, leading to the rollback of the entire update process. However, Transaction B has already read the updated data, resulting in a situation where the client observes incorrect information. This phenomenon is referred to as a ‘Dirty Read.

**NON-REPEATABLE READS**



At time T0, Transaction A reads the product with ID 1001. By time T1, Transaction B updates the ‘price’ column of the product with ID 1001 to $1999 and commits at time T2. Transaction A needs to read the product with ID 1001 again at time T3. It compares the two columns of data: the initial read at time T0 and the latest read at T3. However, Transaction A determines that the data are not equivalent. As a result, this phenomenon is referred to as non-repeatable reads.

**PHANTOM READ**



Phantom read is pretty similar to non-repeatable reads. There is only one exception among them. “Non-repeatable reads” is valid for one column. Phantom read is valid for a dataset. For instance, At time T0, Transaction A reads the product with ID 1001. By time T1, Transaction B updates the data set of the product (name, price, description etc.) with ID 1001 and commits at time T2. Transaction A needs to read the product with ID 1001 again at time T3. It compares the two sets of data: the initial read at time T0 and the latest read at T3. However, Transaction A determines that the data are not equivalent. As a result, this phenomenon is referred to as Phantom read.

In spring, transactional annotation contains isolation-level property, such as DEFAULT, READ\_UNCOMMITTED, READ\_COMMITTED, REPEATABLE\_READ, and SERIALIZABLE. But Spring Transaction uses DEFAULT isolation level. It means the default isolation level of the database.

**LEVELS**

**DEFAULT**

default isolation level of the database.

Isolation isolation() default Isolation.DEFAULT;

**READ\_UNCOMMITTED**

This is the lowest level of isolation and allows many simultaneous accesses. It cannot prevent problems with concurrency.

@Transactional(propagation = Propagation.REQUIRED, isolation = Isolation.READ\_UNCOMMITTED)  
public Product getById(Integer id) {  
 return productRepository.findById(id).orElse(null);  
}

**READ\_COMMITTED**

2nd level isolation stop. DIRTY READ prevents it. But other problems of synchronicity can occur.

@Transactional(propagation = Propagation.REQUIRED, isolation = Isolation.READ\_COMMITTED)  
public Product getById(Integer id) {  
 return productRepository.findById(id).orElse(null);  
}

**REPEATABLE\_READ**

3rd level is isolation. Prevents DIRTY READ and NON-REPEATABLE READ.

@Transactional(propagation = Propagation.REQUIRED, isolation = Isolation.REPEATABLE\_READ)  
public Product getById(Integer id) {  
 return productRepository.findById(id).orElse(null);  
}

**SERIALIZABLE**

It’s the highest level of isolation. It prevents all concurrency problems. Executes concurrent calls sequentially. It is expensive in terms of performance.

@Transactional(propagation = Propagation.REQUIRED, isolation = Isolation.SERIALIZABLE)  
public Product getById(Integer id) {  
 return productRepository.findById(id).orElse(null);  
}

**SOME OTHER Attributes**

**Timeout**

The timeout for a transaction is in seconds. 30 seconds

@Transactional(timeout = 30)

**Read-Only**

If it is set as true, the transaction is allowed only to read operations, not modify them. The default is always false.

@Transactional(readOnly = true)

**rollbackFor**

Custom exception. Transaction only triggers rollback when encountering a custom exception.

@Transactional(rollbackFor = CustomException.class)  
public void create(Product product, List<Image> images) {  
 saveImages(images);  
  
 if (product.getName().equals("Berat")) {  
 throw new CustomException("");  
 }  
 productRepository.save(product);  
}