BACK ENDHANCE

Maximizing your backend potential

Pragmatic Spring

by Marcus Held

Marcus Held

Programming since 2004
12 years+ on the JVM

Freelancer, Leader, Speaker, Blogger, Father of two

+

Be excited for:

01

The Pragmatic Programmer

What is this about?

02

Design by Contract Preconditions

How Spring supports you

03

Juggling the Real World

Features that helps to decouple

The Pragmatic Programmer

Early adapter / fast adapter

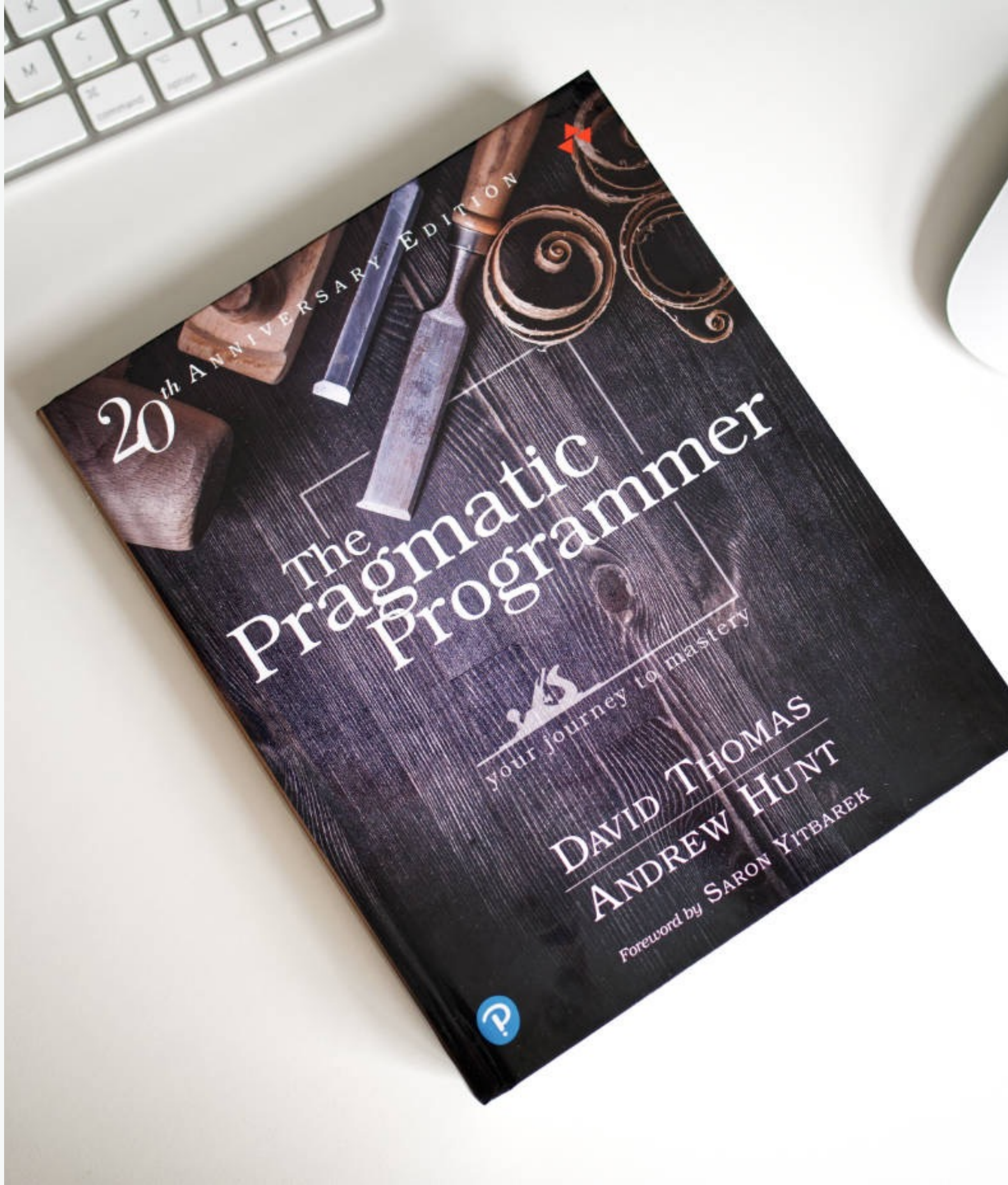
Inquisitive

Critical thinker

Realistic

Jack of all trades

Care about your craft



Design by Contract

Described by Bertrand Meyer in 1997

Document (and agree) on the rights and responsibilities of a software module/service/class/function.

Preconditions

Which requirements apply to execute the routine?

Postconditions

What is the routine doing?

Class invariants

From the perspective of a caller – the class ensures that this condition is always true.

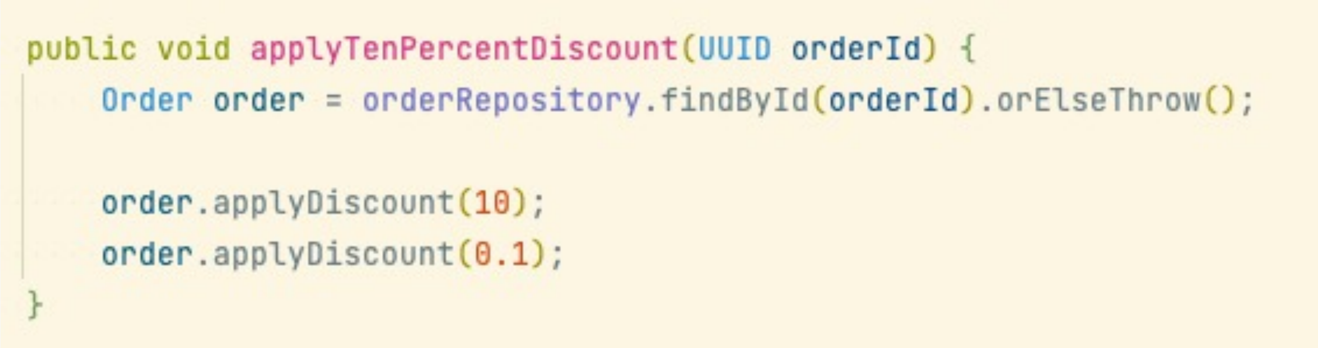
Preconditions

What's correct?

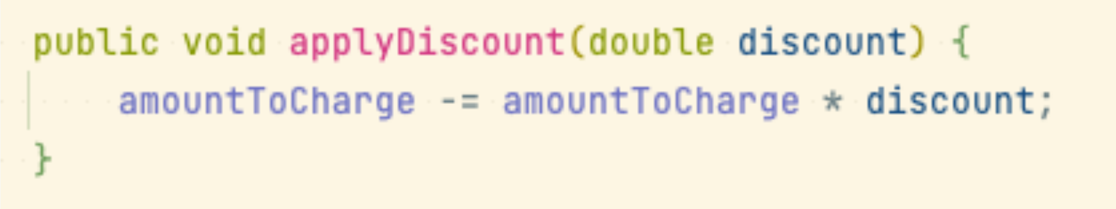
The caller doesn't know – without checking the code

Possible Solutions:

- Make the type explicit
- Add parameter documentation
- Do an assertion in applyDiscount



```java
public void applyTenPercentDiscount(UUID orderId) {
    Order order = orderRepository.findById(orderId).orElseThrow();

    order.applyDiscount(10);
    order.applyDiscount(0.1);
}
```



```java
public void applyDiscount(double discount) {
    amountToCharge -= amountToCharge * discount;
}
```

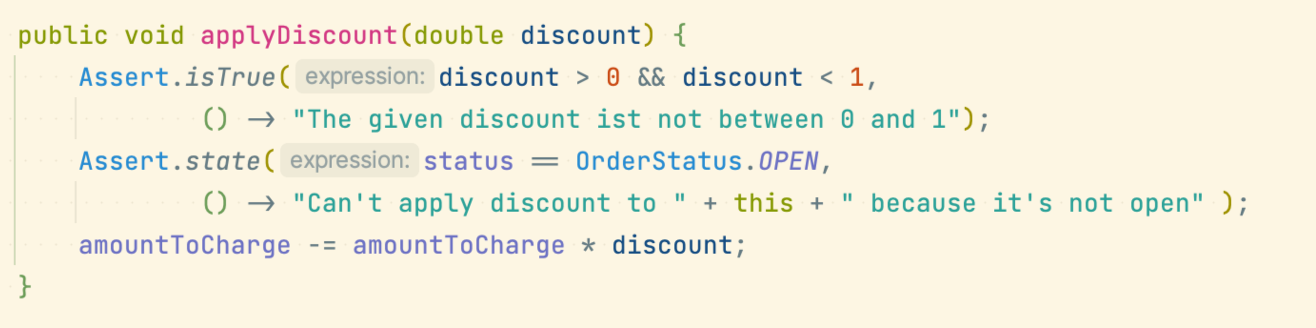
Spring Assert

Already on your classpath

Consistency with the framework

Correct ExceptionTypes for asserts

MessageSupliers to guard the message



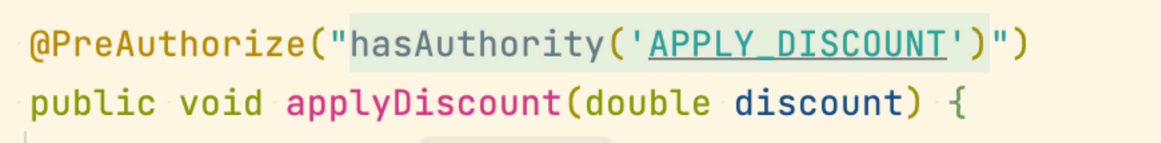
```
public void applyDiscount(double discount) {
    Assert.isTrue( expression: discount > 0 && discount < 1,
            () -> "The given discount ist not between 0 and 1");
    Assert.state( expression: status == OrderStatus.OPEN,
            () -> "Can't apply discount to " + this + " because it's not open" );
    amountToCharge -= amountToCharge * discount;
}
```

Method Security

Many routines require permissions to run

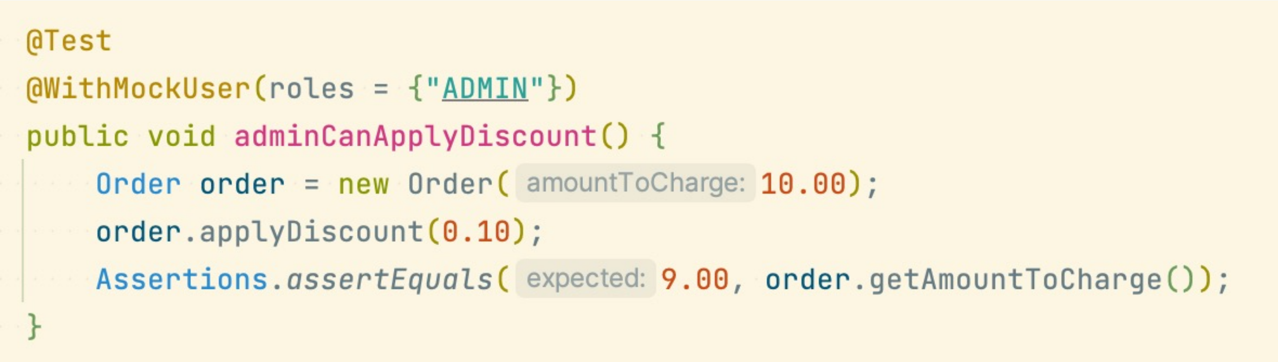
Authorization is a cross cutting concern and a good candidate to apply AOP

Also: Every routine has an (implicit) security contract



```
@PreAuthorize("hasAuthority('APPLY_DISCOUNT')")
public void applyDiscount(double discount) {
```

Method Security



```
@Test
@WithMockUser(roles = {"ADMIN"})
public void adminCanApplyDiscount() {
    Order order = new Order( amountToCharge: 10.00);
    order.applyDiscount(0.10);
    Assertions.assertEquals( expected: 9.00, order.getAmountToCharge());
}
```

@PreAuthorize, @PostAuthorize, @Secured, @RolesAllowed , @PreFilter, @PostFilter

Unit tests to test business logic are not cluttered by obligatory security checks

Can be combined in meta annotations

! Spring AOP proxying rules apply
! SecurityContext is thread-bound

Juggling the Real World

"Computers have to integrate into our world, not the other way around. And our world is messy: things are constantly happening, stuff gets moved around, we change our minds, ... And the applications we write somehow have to work out what to do."

Our applications must be responsive to change

Application Events

Part of Spring core framework

Works with POJOs

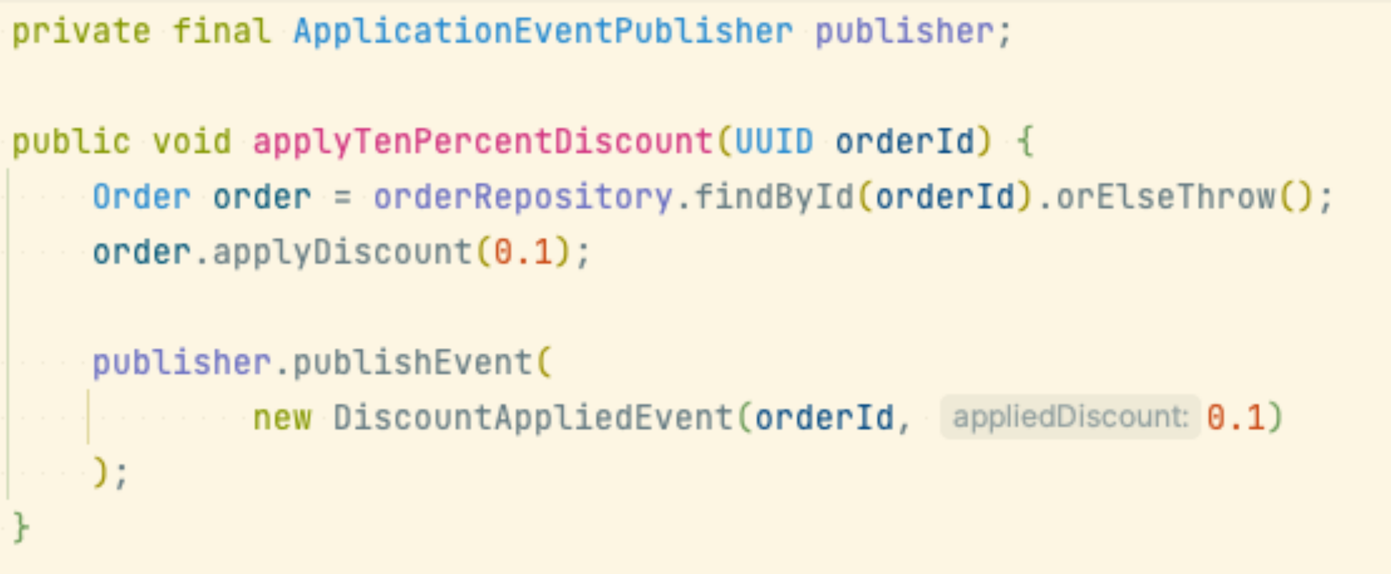
Testing support

Transaction support

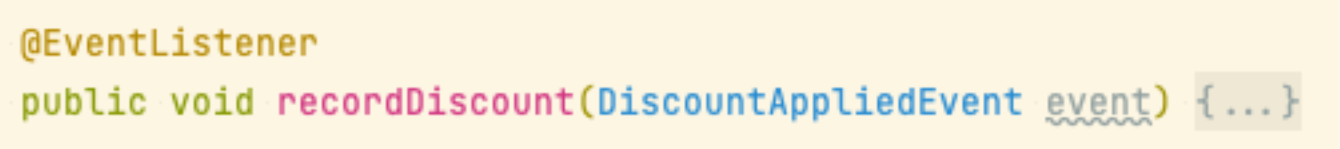
@Async support

@Order support

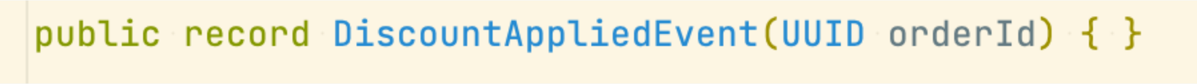
Conditional listeners



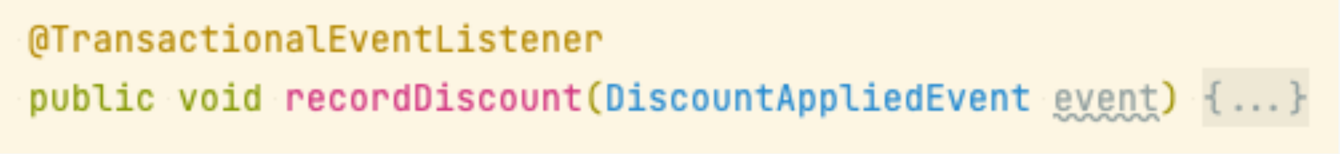
```
private final ApplicationEventPublisher publisher;

public void applyTenPercentDiscount(UUID orderId) {
    Order order = orderRepository.findById(orderId).orElseThrow();
    order.applyDiscount(0.1);

    publisher.publishEvent(
            new DiscountAppliedEvent(orderId,  appliedDiscount: 0.1)
    );
}
```



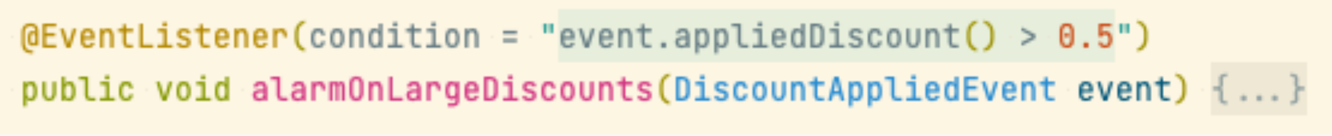
```
@EventListener
public void recordDiscount(DiscountAppliedEvent event) {...}
```



```
public record DiscountAppliedEvent(UUID orderId) { }
```



```
@TransactionalEventListener
public void recordDiscount(DiscountAppliedEvent event) {...}
```



```
@EventListener(condition = "event.appliedDiscount() > 0.5")
public void alarmOnLargeDiscounts(DiscountAppliedEvent event) {...}
```

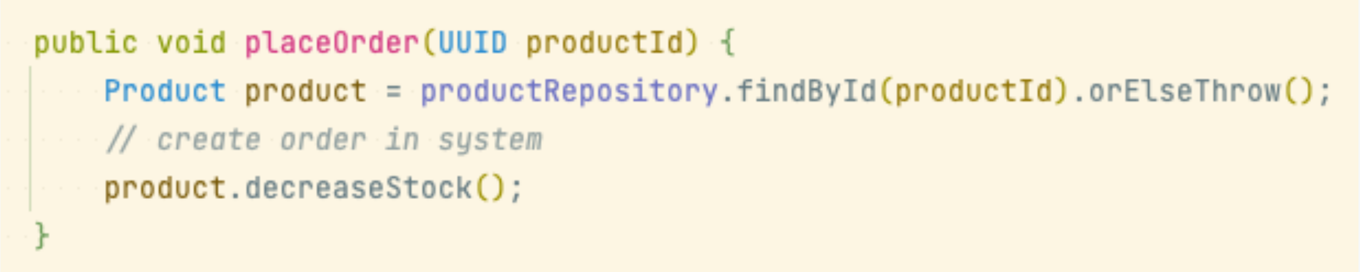
Shared State is Incorrect State

Imagine: You ask the waiter in the restaurant:
“Is the 1995 Cabernet Sauvignon available?”
Waiter looks to the bar
“You are lucky, there’s one bottle left”

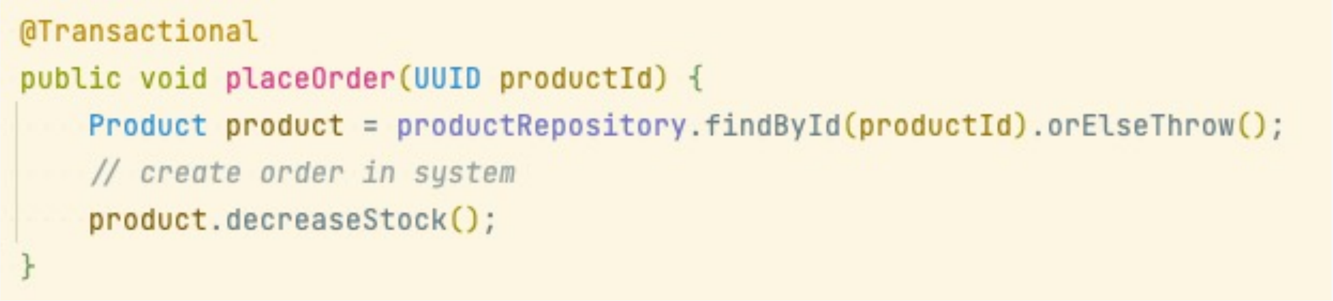
At the same time, on the other side of the restaurant, someone else also asks for the wine.

The problem here: shared state.

Transactions



```java
public void placeOrder(UUID productId) {
    Product product = productRepository.findById(productId).orElseThrow();
    // create order in system
    product.decreaseStock();
}
```



```java
@Transactional
public void placeOrder(UUID productId) {
    Product product = productRepository.findById(productId).orElseThrow();
    // create order in system
    product.decreaseStock();
}
```

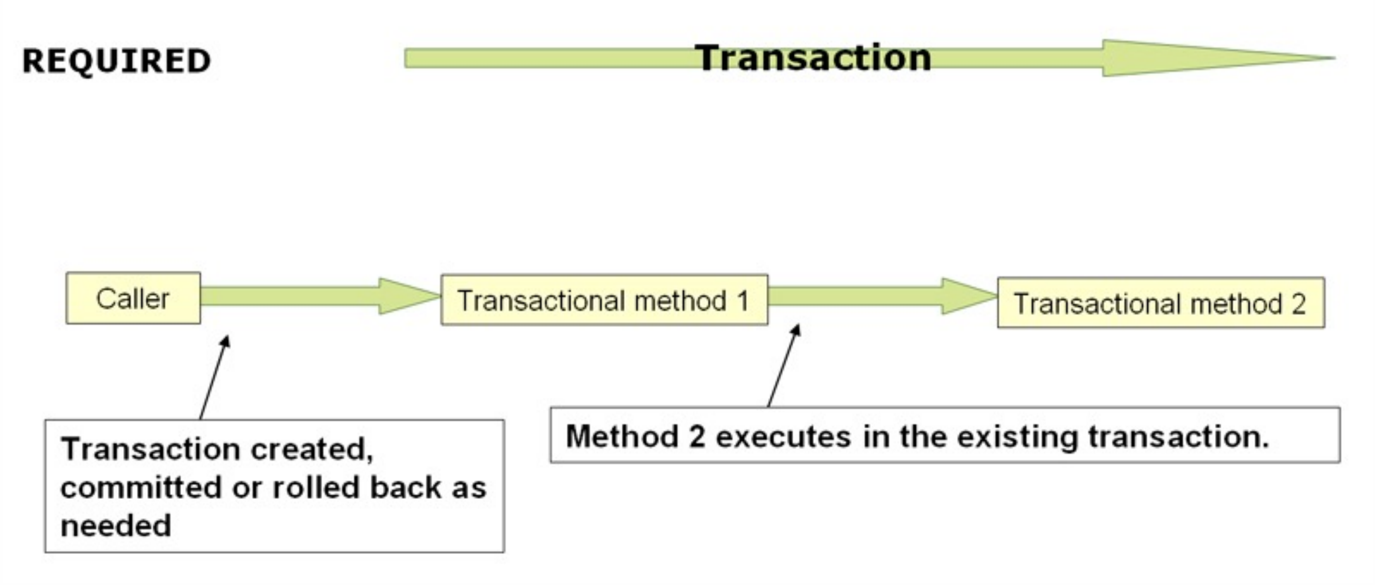
Spring Transactions

Propagations

Isolation Levels

Specify rollback scenarios

readOnly flag



REQUIRED

Transaction

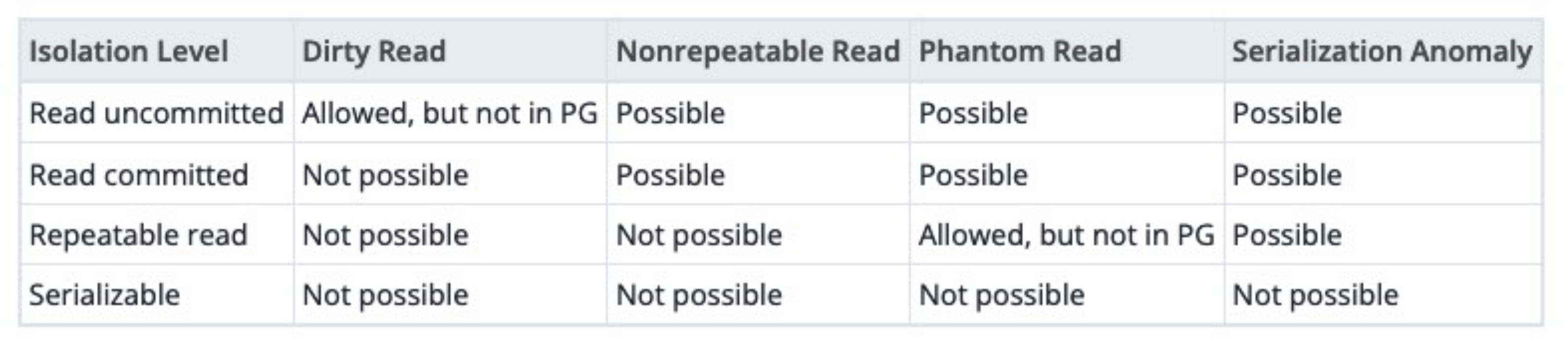
Caller

Transactional method 1

Transactional method 2

Transaction created, committed or rolled back as needed

Method 2 executes in the existing transaction.



Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Read uncommitted	Allowed, but not in PG	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed, but not in PG	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

Scopes

By default, every Bean is a singleton
Prototype: New object per invocation
RequestScope: New object per request

Implement the scope interface for your needs

Thanks for Listening!

New: Spring Performance Workshop

marcus@backendhance.com

https://backendhance.com

+49 151 / 449 351 67
linkedin.com/in/marcus-held/

BACKENDHANCE

Maximizing your backend potential