Jakarta Data: what it means for Java Community

Mikhail Polivakha

Polivakha Mikhail

- Active Open Source Member (including Spring Projects)
- Senior Software Engineer
- Spring Aio Community Board Member
- Conference Speaker
- Technical Writer on Baeldung

Contacts:

- Telegram: @mipo256
- GitHub: mipo256
- Blog: https://mpolivaha.com
- Email: mikhailpolivakha@gmail.com

Agenda

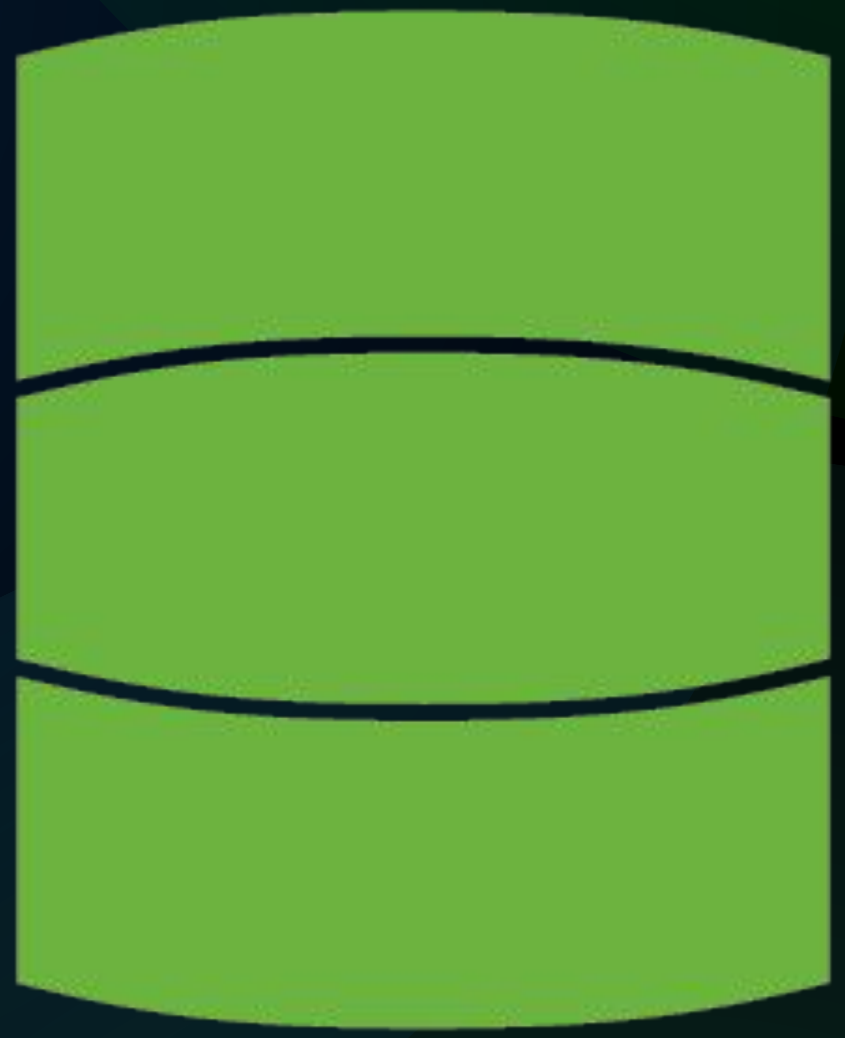
- What is Jakarta Data?
- Why it appeared?
- What are the core design principles of it?
- Why Spring Data has challenges in implementation?
- How it can be integrated into traditional Spring applications?

Disclaimer №1

HIBERNATE

Disclaimer №1

Disclaimer №2

I ❤ 

HIBERNATE

JBoss®

by Red Hat

HIBERNATE

JBoss®

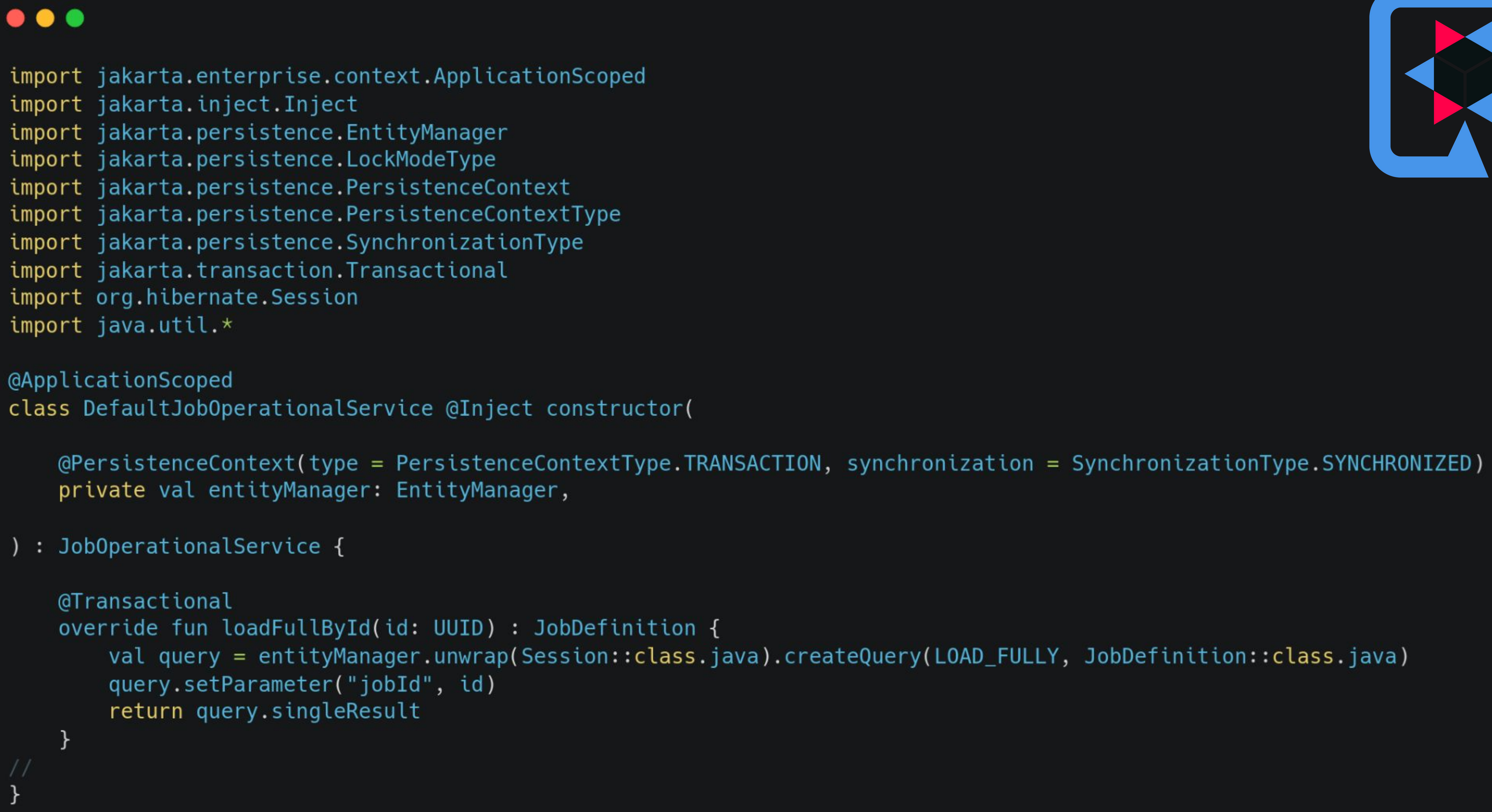
by Red Hat

HIBERNATE

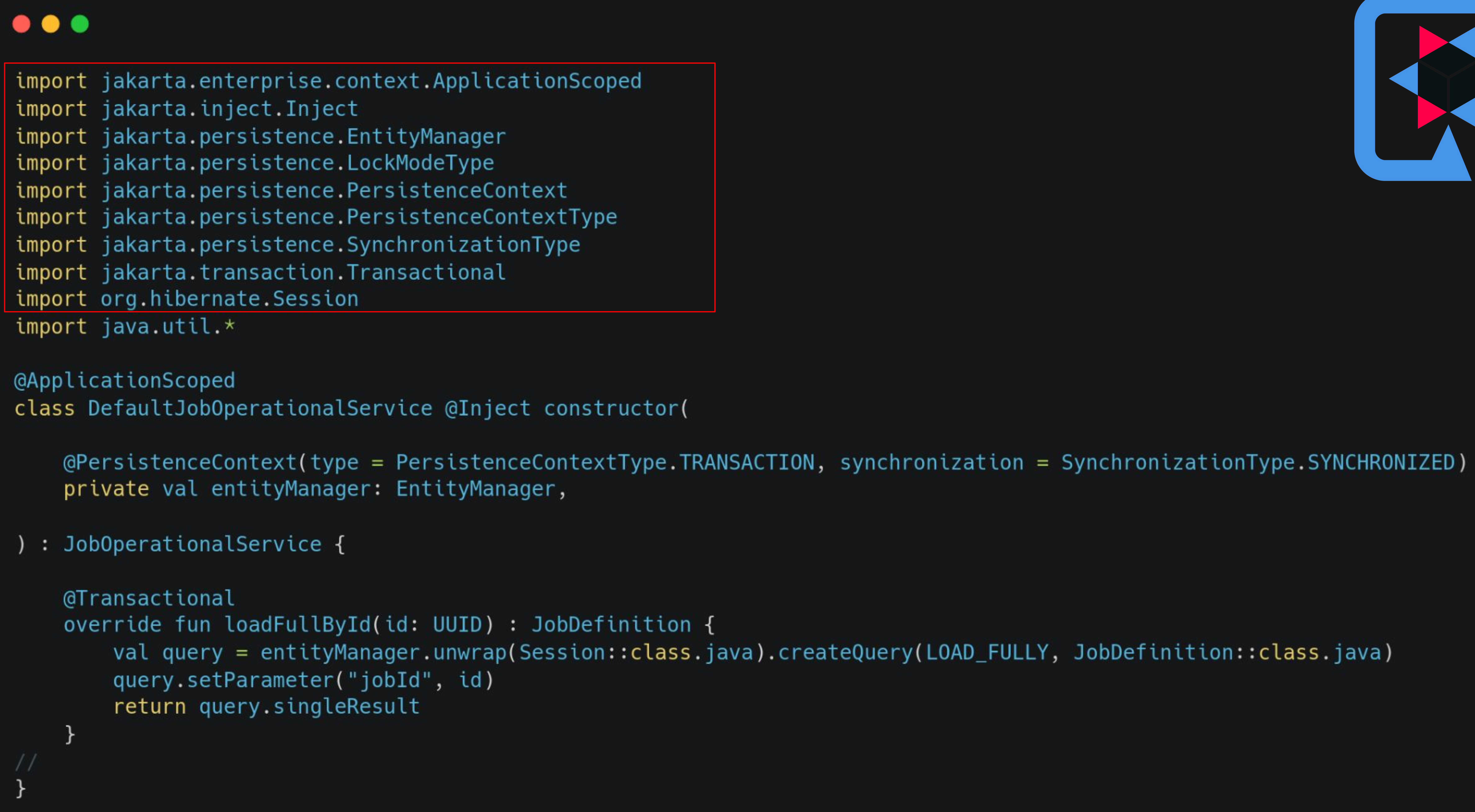
JBoss®

by Red Hat

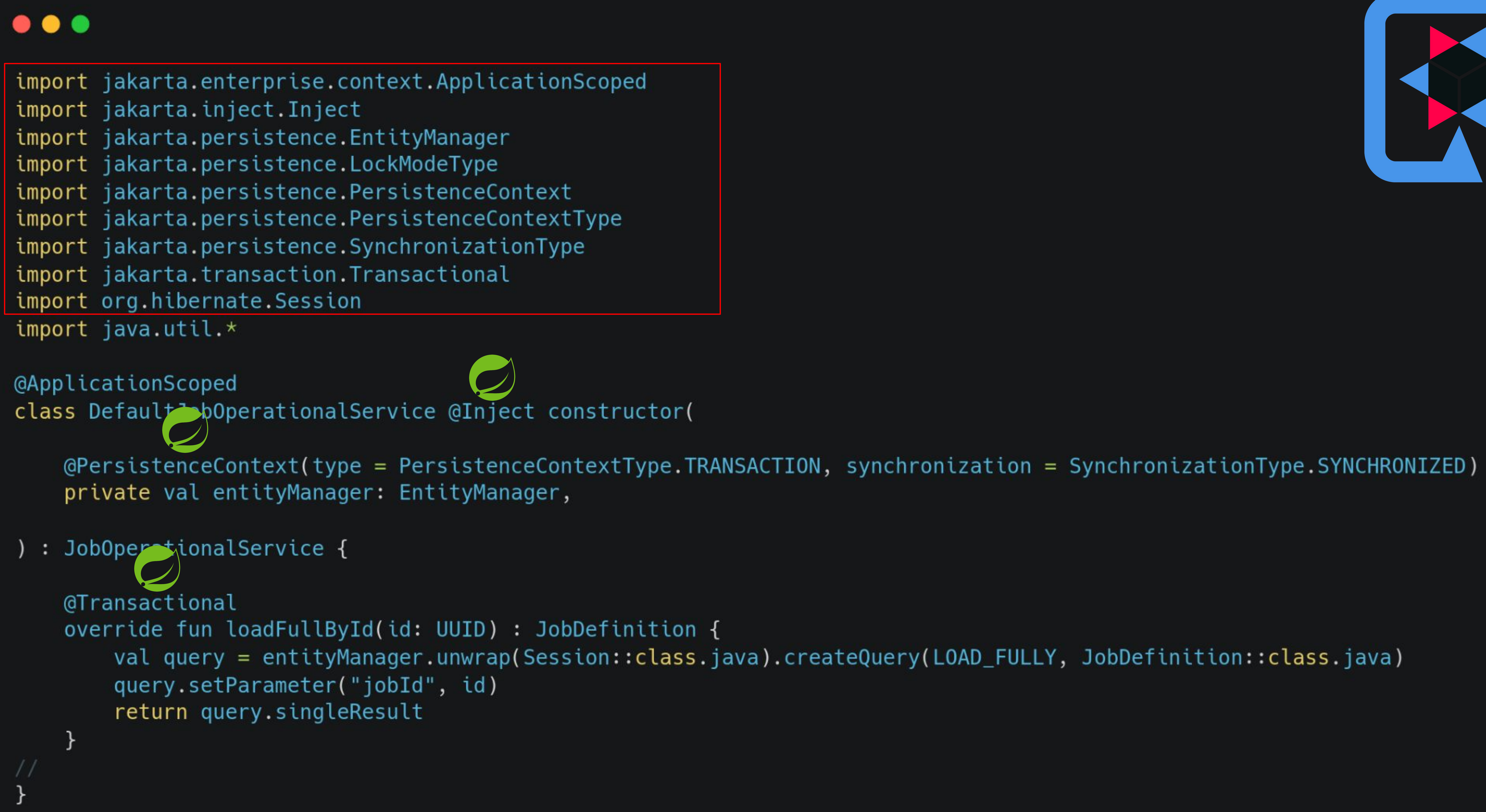
JAKARTA® EE



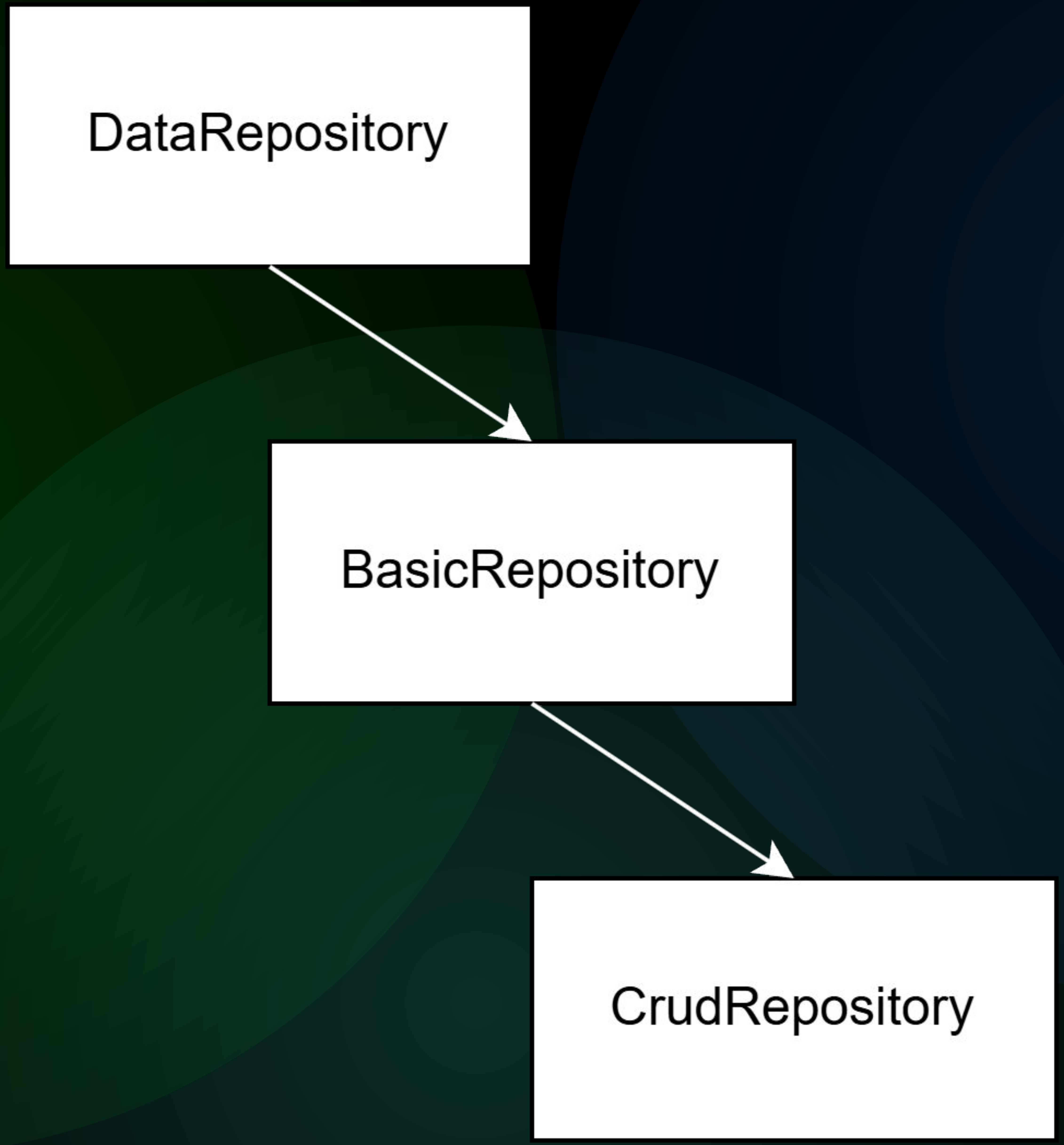
```kotlin
import jakarta.enterprise.context.ApplicationScoped
import jakarta.inject.Inject
import jakarta.persistence.EntityManager
import jakarta.persistence.LockModeType
import jakarta.persistence.PersistenceContext
import jakarta.persistence.PersistenceContextType
import jakarta.persistence.SynchronizationType
import jakarta.transaction.Transactional
import org.hibernate.Session
import java.util.*

@ApplicationScoped
class DefaultJobOperationalService @Inject constructor(

    @PersistenceContext(type = PersistenceContextType.TRANSACTION, synchronization = SynchronizationType.SYNCHRONIZED)
    private val entityManager: EntityManager,

) : JobOperationalService {

    @Transactional
    override fun loadFullById(id: UUID) : JobDefinition {
        val query = entityManager.unwrap(Session::class.java).createQuery(LOAD_FULLY, JobDefinition::class.java)
        query.setParameter("jobId", id)
        return query.singleResult
    }
//
}
```



```kotlin
import jakarta.enterprise.context.ApplicationScoped
import jakarta.inject.Inject
import jakarta.persistence.EntityManager
import jakarta.persistence.LockModeType
import jakarta.persistence.PersistenceContext
import jakarta.persistence.PersistenceContextType
import jakarta.persistence.SynchronizationType
import jakarta.transaction.Transactional
import org.hibernate.Session
import java.util.*

@ApplicationScoped
class DefaultJobOperationalService @Inject constructor(

    @PersistenceContext(type = PersistenceContextType.TRANSACTION, synchronization = SynchronizationType.SYNCHRONIZED)
    private val entityManager: EntityManager,

) : JobOperationalService {

    @Transactional
    override fun loadFullById(id: UUID) : JobDefinition {
        val query = entityManager.unwrap(Session::class.java).createQuery(LOAD_FULLY, JobDefinition::class.java)
        query.setParameter("jobId", id)
        return query.singleResult
    }
//
}
```



```kotlin
import jakarta.enterprise.context.ApplicationScoped
import jakarta.inject.Inject
import jakarta.persistence.EntityManager
import jakarta.persistence.LockModeType
import jakarta.persistence.PersistenceContext
import jakarta.persistence.PersistenceContextType
import jakarta.persistence.SynchronizationType
import jakarta.transaction.Transactional
import org.hibernate.Session
import java.util.*

@ApplicationScoped
class DefaultJobOperationalService @Inject constructor(

    @PersistenceContext(type = PersistenceContextType.TRANSACTION, synchronization = SynchronizationType.SYNCHRONIZED)
    private val entityManager: EntityManager,

) : JobOperationalService {

    @Transactional
    override fun loadFullById(id: UUID) : JobDefinition {
        val query = entityManager.unwrap(Session::class.java).createQuery(LOAD_FULLY, JobDefinition::class.java)
        query.setParameter("jobId", id)
        return query.singleResult
    }
//
}
```

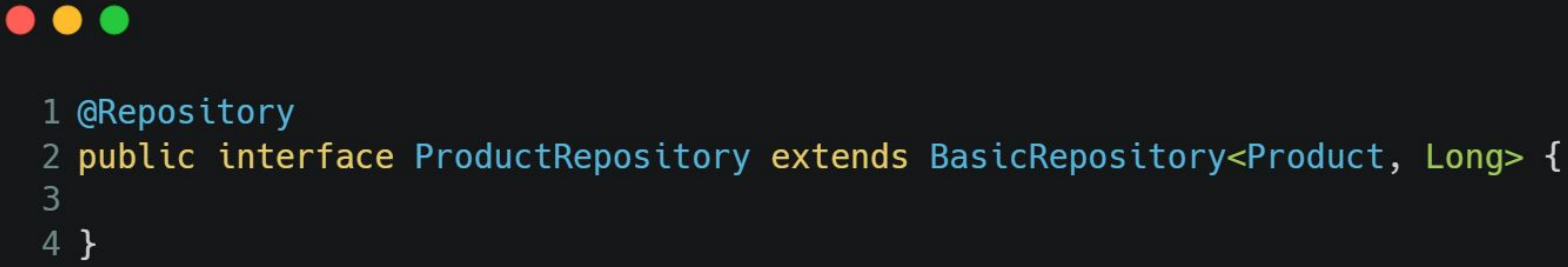


DataRepository

BasicRepository

CrudRepository

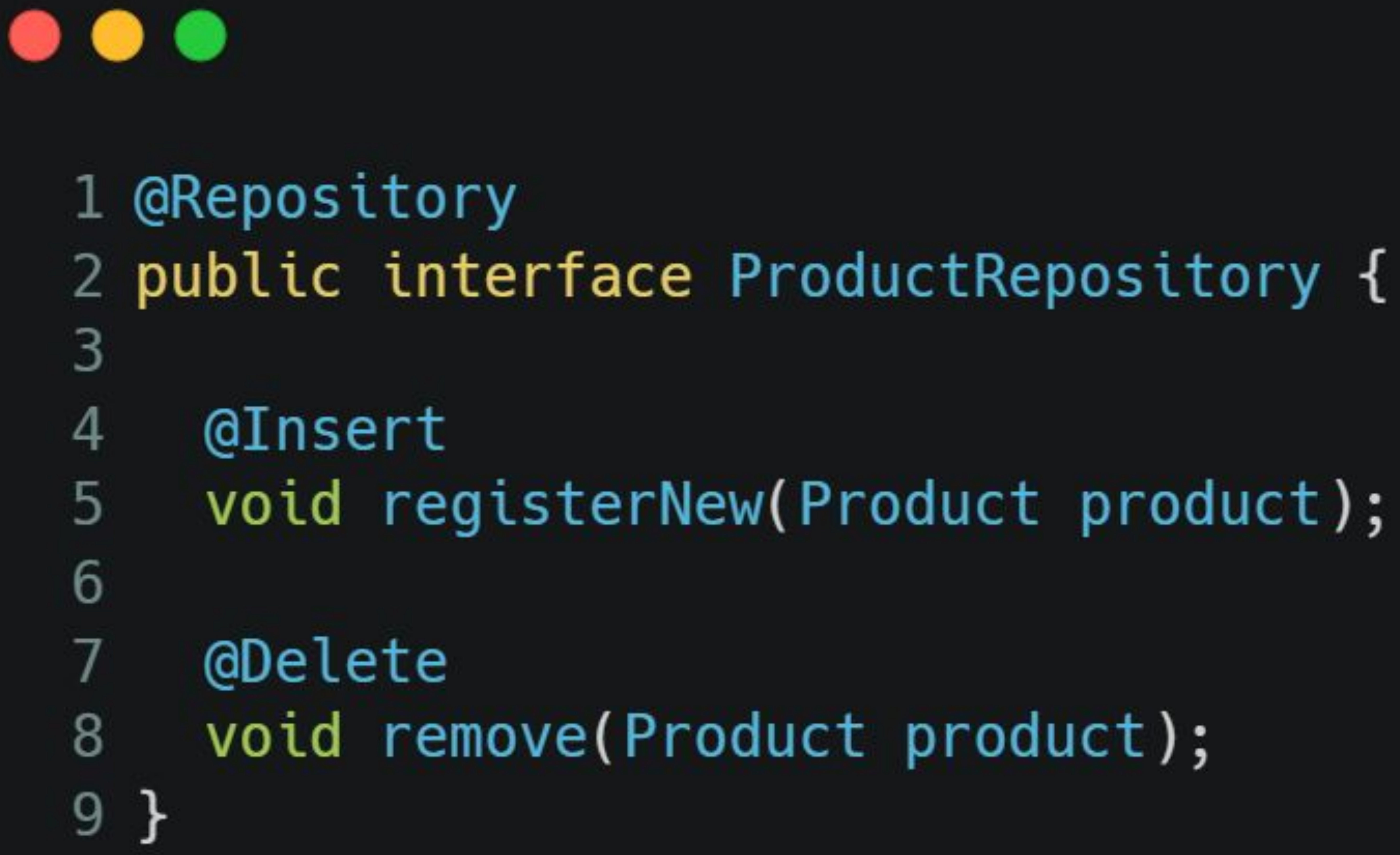
JAKARTA® EE



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

}
```

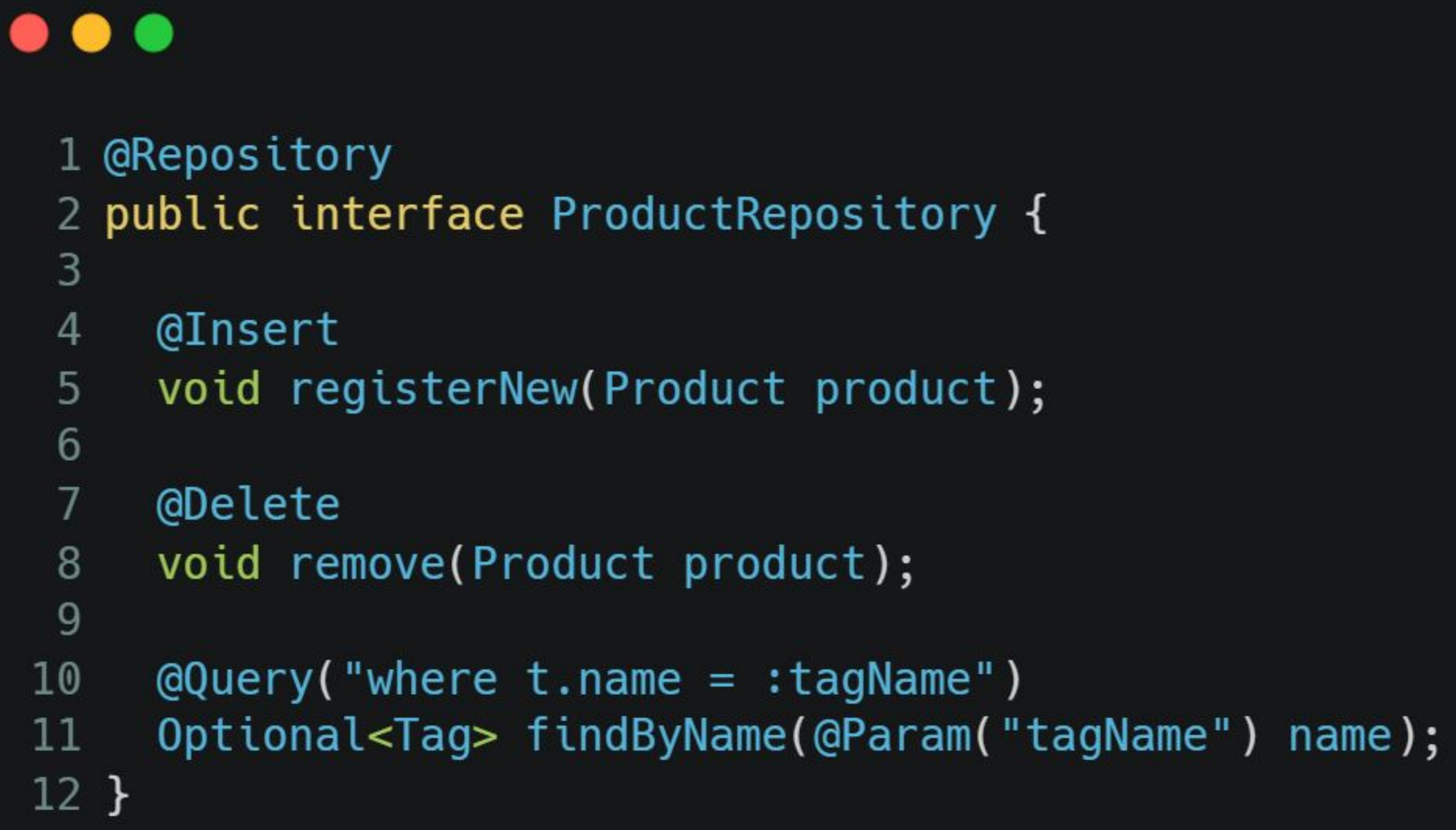
```
@Repository
public interface Prod                                    ository<Product, Long> {

}
```

JAKARTA® EE



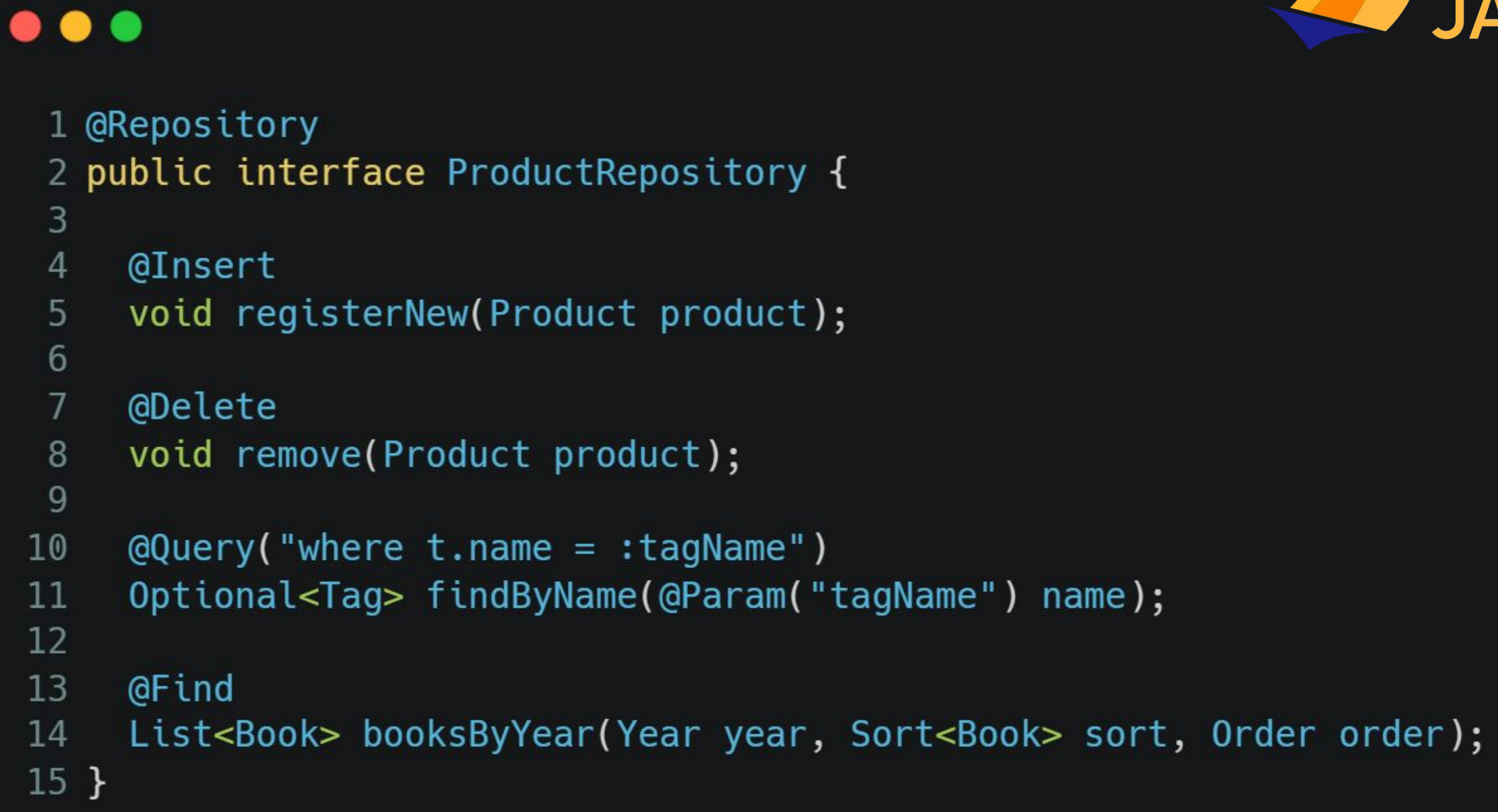
```java
@Repository
public interface ProductRepository {

  @Insert
  void registerNew(Product product);

  @Delete
  void remove(Product product);
}
```

JAKARTA® EE



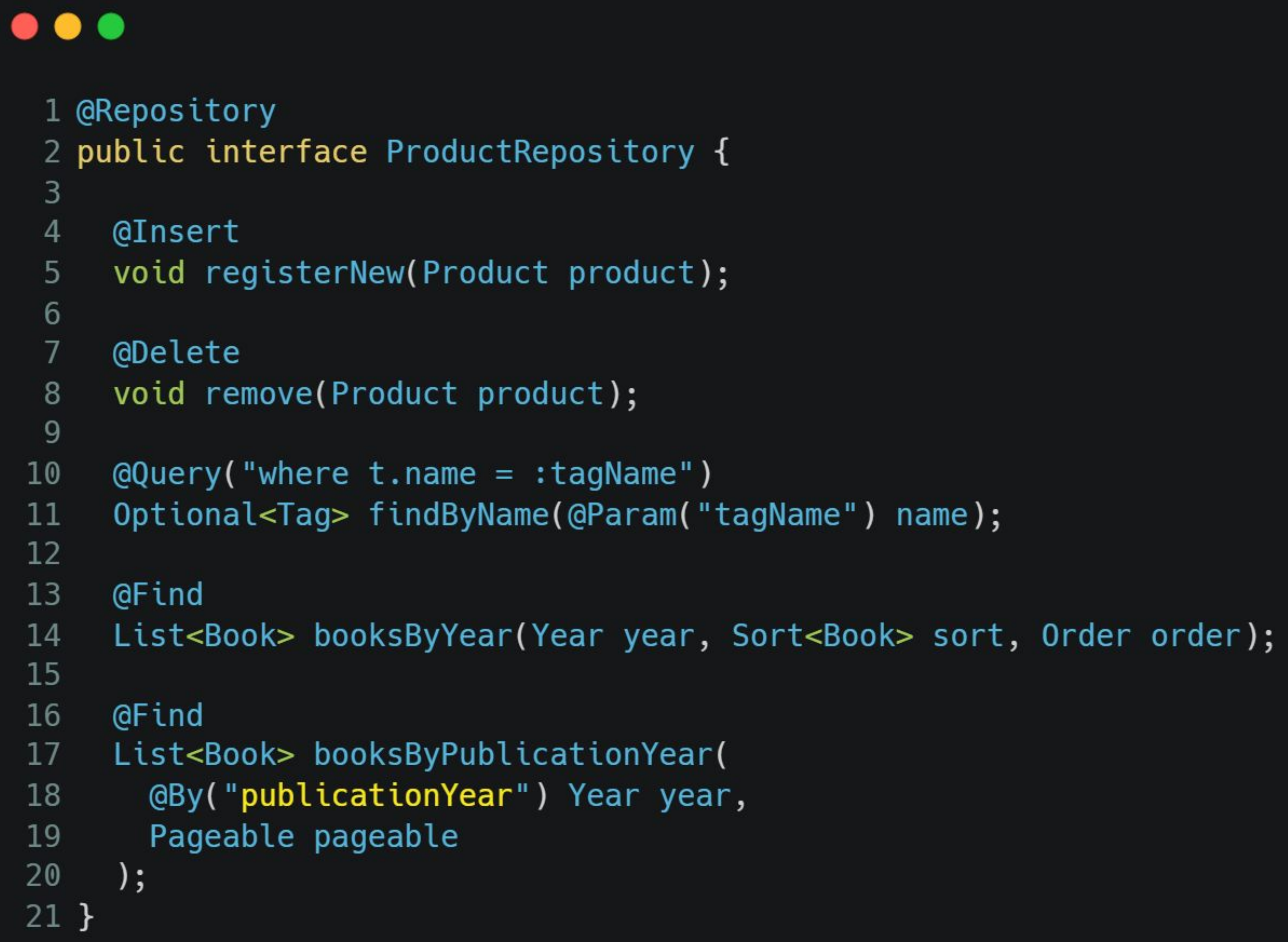
```java
@Repository
public interface ProductRepository {

  @Insert
  void registerNew(Product product);

  @Delete
  void remove(Product product);

  @Query("where t.name = :tagName")
  Optional<Tag> findByName(@Param("tagName") name);
}
```

JAKARTA® EE



```java
@Repository
public interface ProductRepository {

  @Insert
  void registerNew(Product product);

  @Delete
  void remove(Product product);

  @Query("where t.name = :tagName")
  Optional<Tag> findByName(@Param("tagName") name);

  @Find
  List<Book> booksByYear(Year year, Sort<Book> sort, Order order);
}
```

JAKARTA® EE



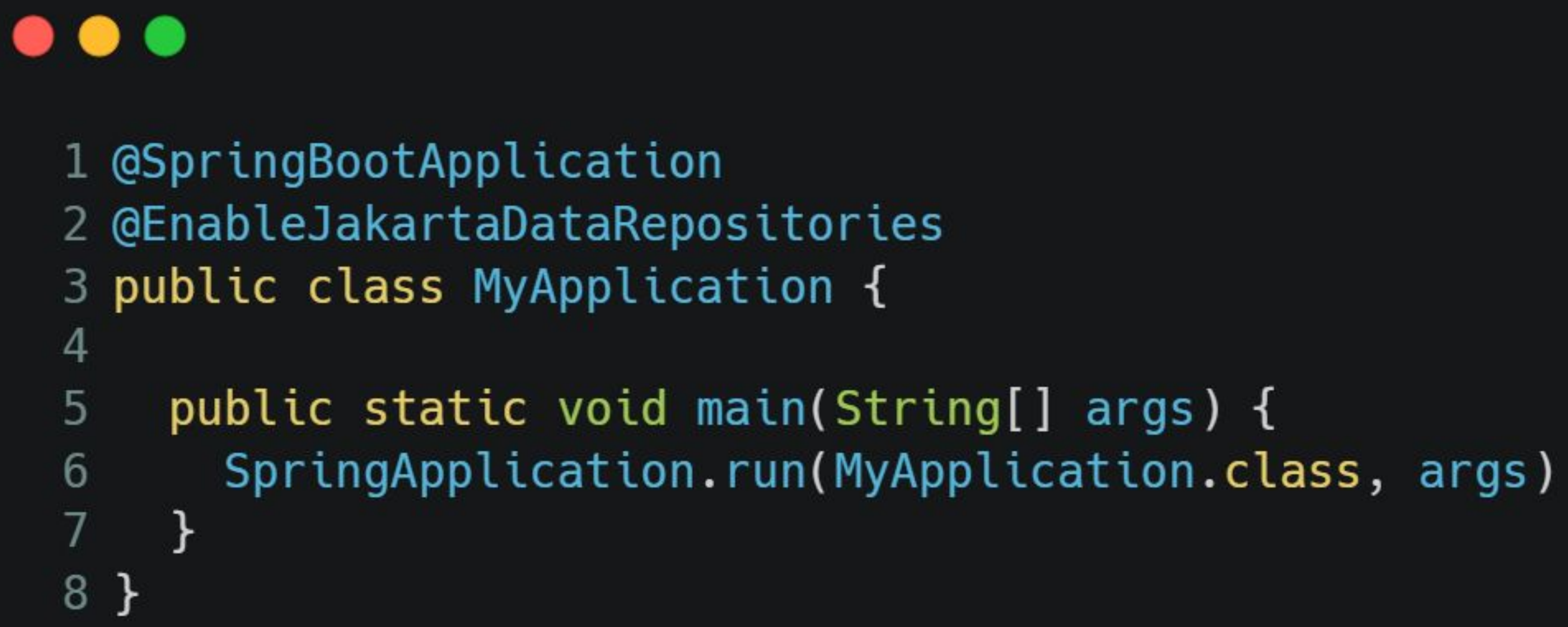
```java
@Repository
public interface ProductRepository {

  @Insert
  void registerNew(Product product);

  @Delete
  void remove(Product product);

  @Query("where t.name = :tagName")
  Optional<Tag> findByName(@Param("tagName") name);

  @Find
  List<Book> booksByYear(Year year, Sort<Book> sort, Order order);

  @Find
  List<Book> booksByPublicationYear(
    @By("publicationYear") Year year,
    Pageable pageable
  );
}
```

mp911de commented on Feb 16

We've been in touch with the project and contributed several suggestions to align Jakarta Data with Spring Data's concepts as the initial specification goal was to standardize what people use for many years.

The project's direction has unfortunately turned into a direction that goes against established design practices in Spring Data with several conflicting concepts and we're no longer interested in adopting the current specification state. Strong voices argue for a more annotation-driven declaration of query methods that are not compatible with how Spring Data works.

That being said, I do not see us adopting that specification. Also, it is currently not clear to us what tangible benefit such a standard provides if it deviates from standardizing what's out there for years.



```java
@SpringBootApplication
@EnableJakartaDataRepositories
public class MyApplication {

  public static void main(String[] args) {
    SpringApplication.run(MyApplication.class, args)
  }
}
```

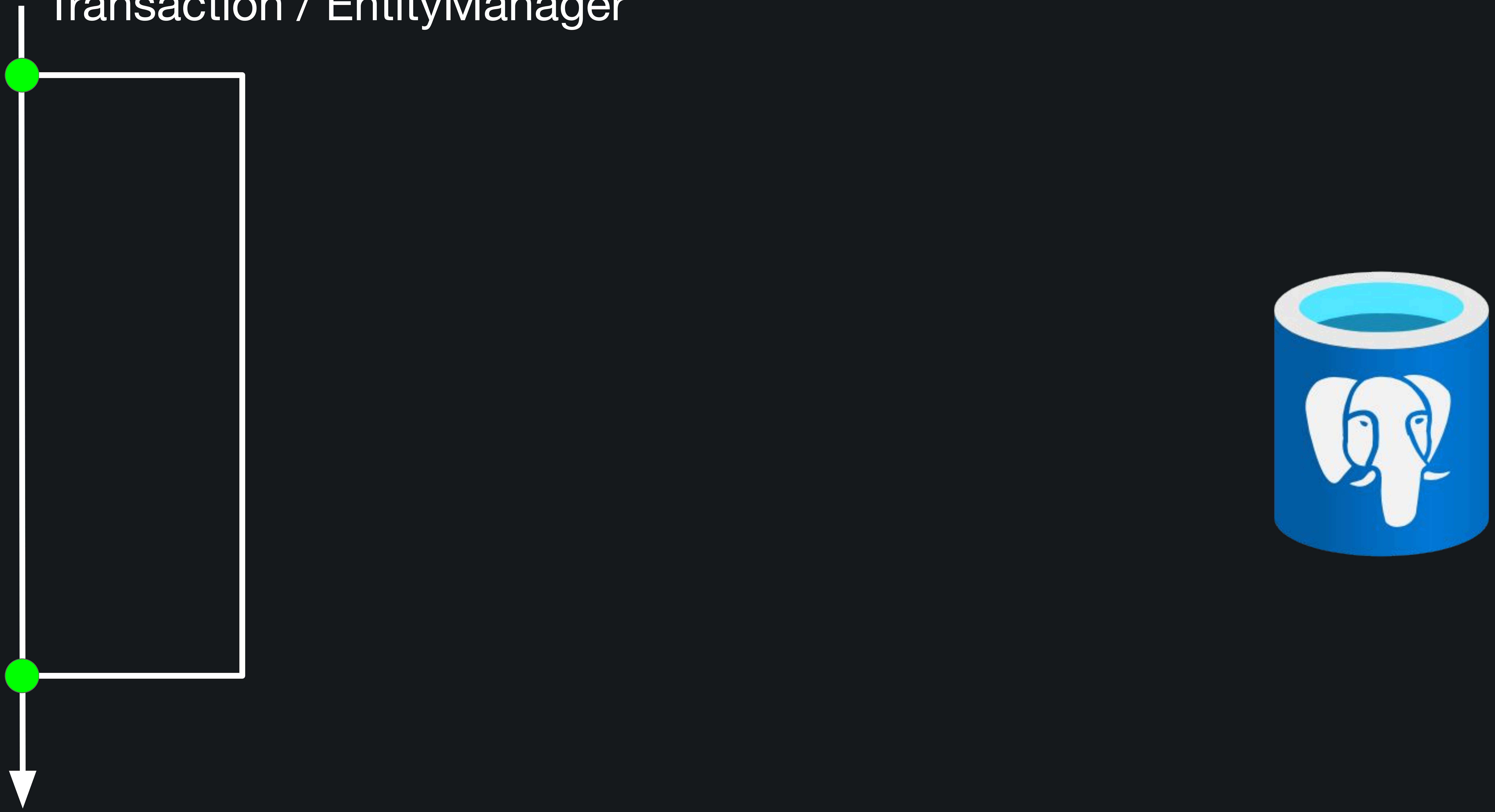
JPA

```
@org.springframework.transaction.annotation.Transactional
void addPostComment(Long postId, String content, Long authorId) {
    Post post = postRepository.findById(postId).orElseThrow();
    var postComments = post.getItems();

    if (containsCommentFromAuthor(postComments, authorId)) {
        throw new IllegalArgumentException();
    }

    postComments.add(
        new PostComment()
            .setContent(content)
            .setAuthor(entityManager.getReference(Author.class, authorId))
    );
}
```

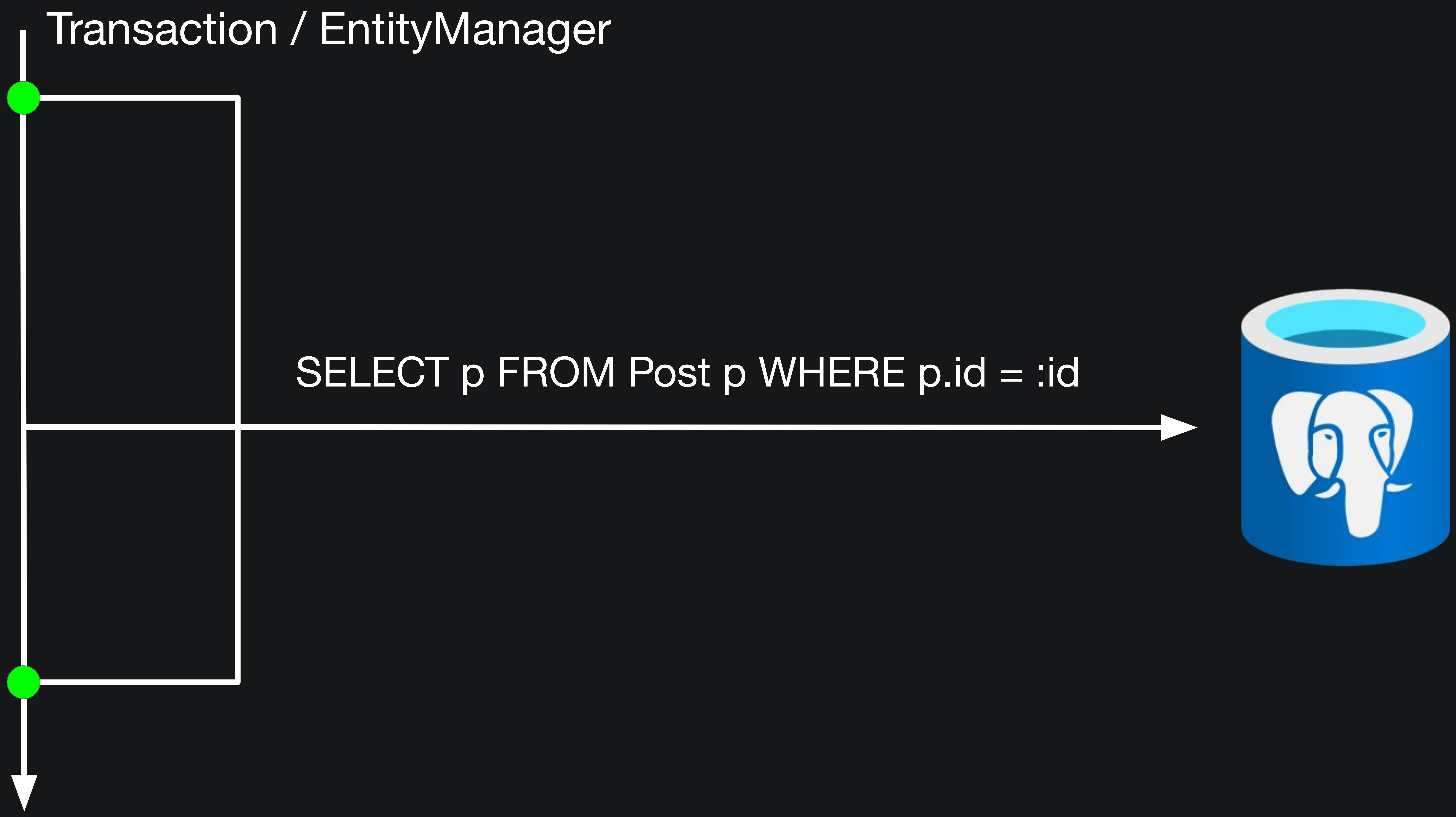
JPA

```
@org.springframework.transaction.annotation.Transactional
void addPostComment(Long postId, String content, Long authorId) {
    Post post = postRepository.findById(postId).orElseThrow();
    var postComments = post.getItems();

    if (containsCommentFromAuthor(postComments, authorId)) {
        throw new IllegalArgumentException();
    }

    postComments.add(
        new PostComment()
            .setContent(content)
            .setAuthor(entityManager.getReference(Author.class, authorId))
    );
}
```

Dirty check!

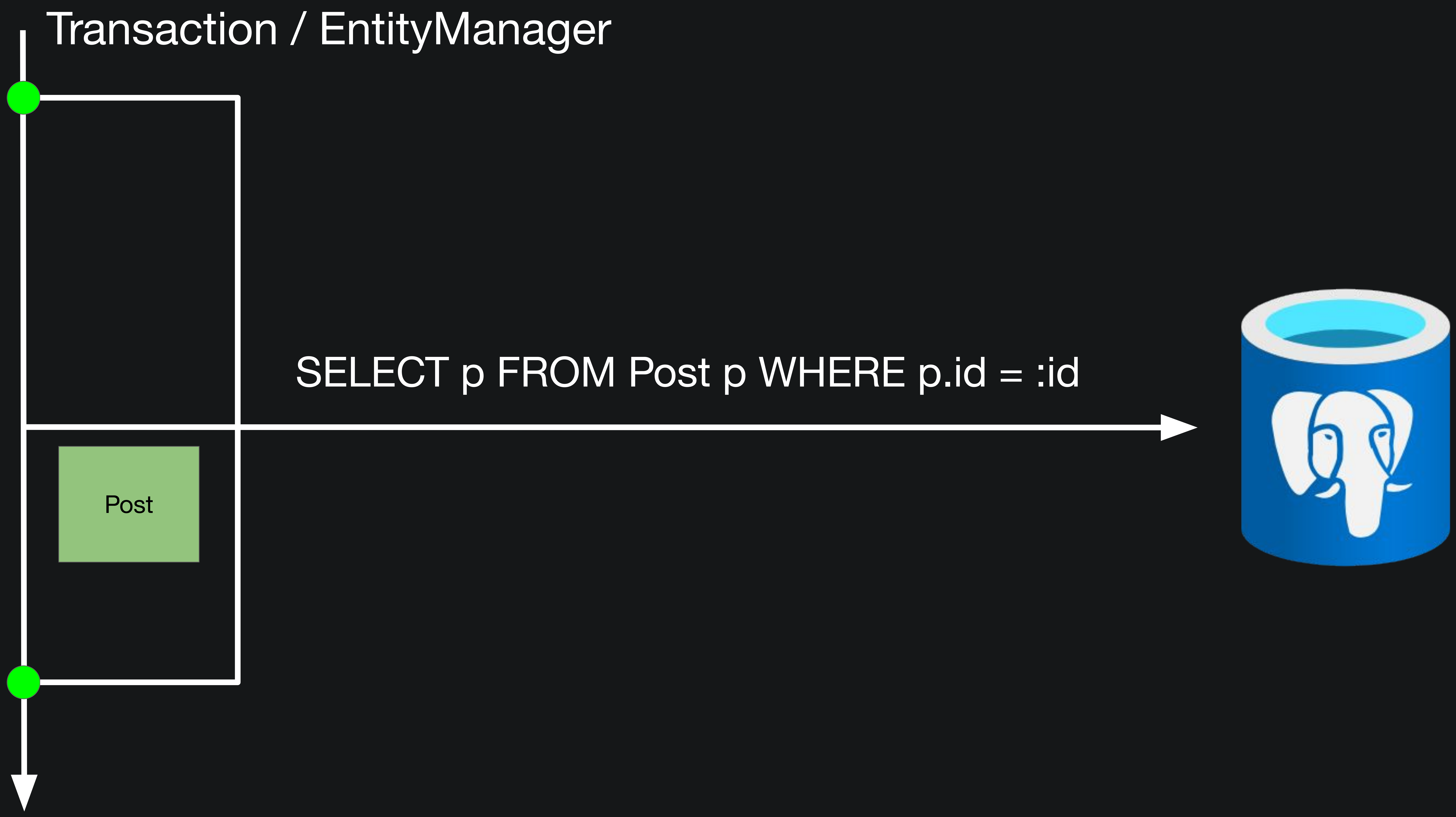


Transaction / EntityManager



Transaction / EntityManager

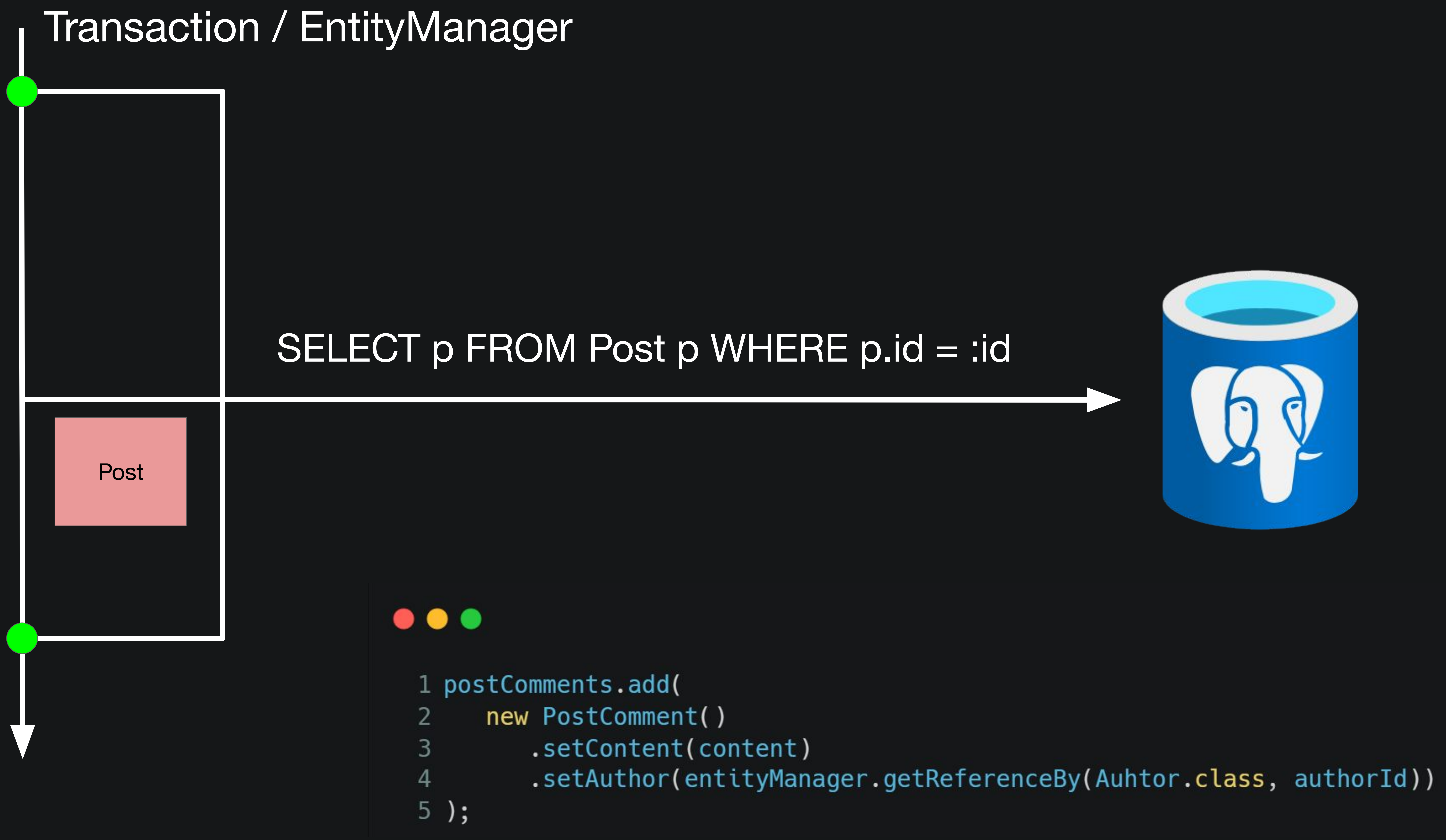
SELECT p FROM Post p WHERE p.id = :id



Transaction / EntityManager

SELECT p FROM Post p WHERE p.id = :id

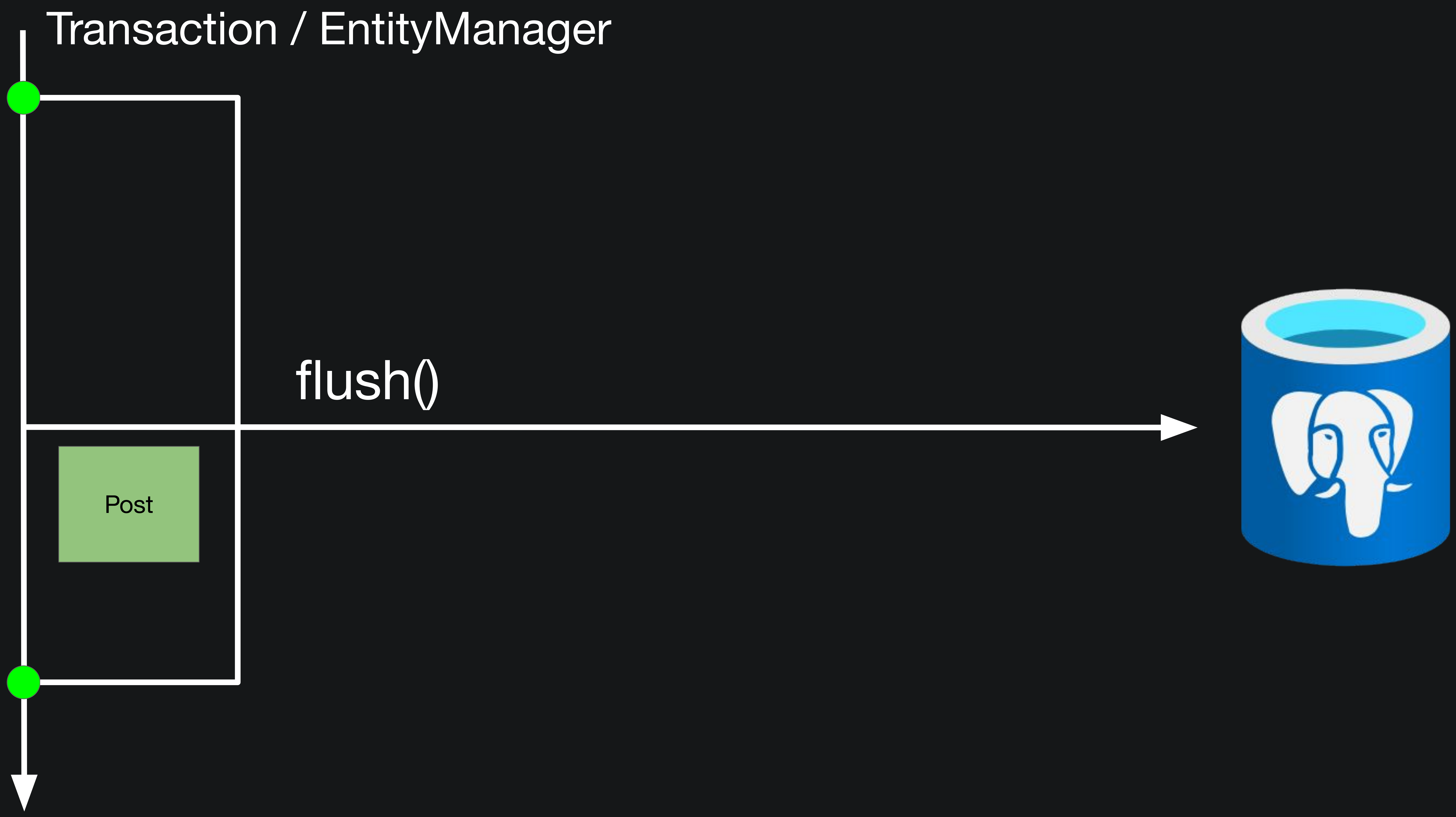
Post



Transaction / EntityManager

SELECT p FROM Post p WHERE p.id = :id

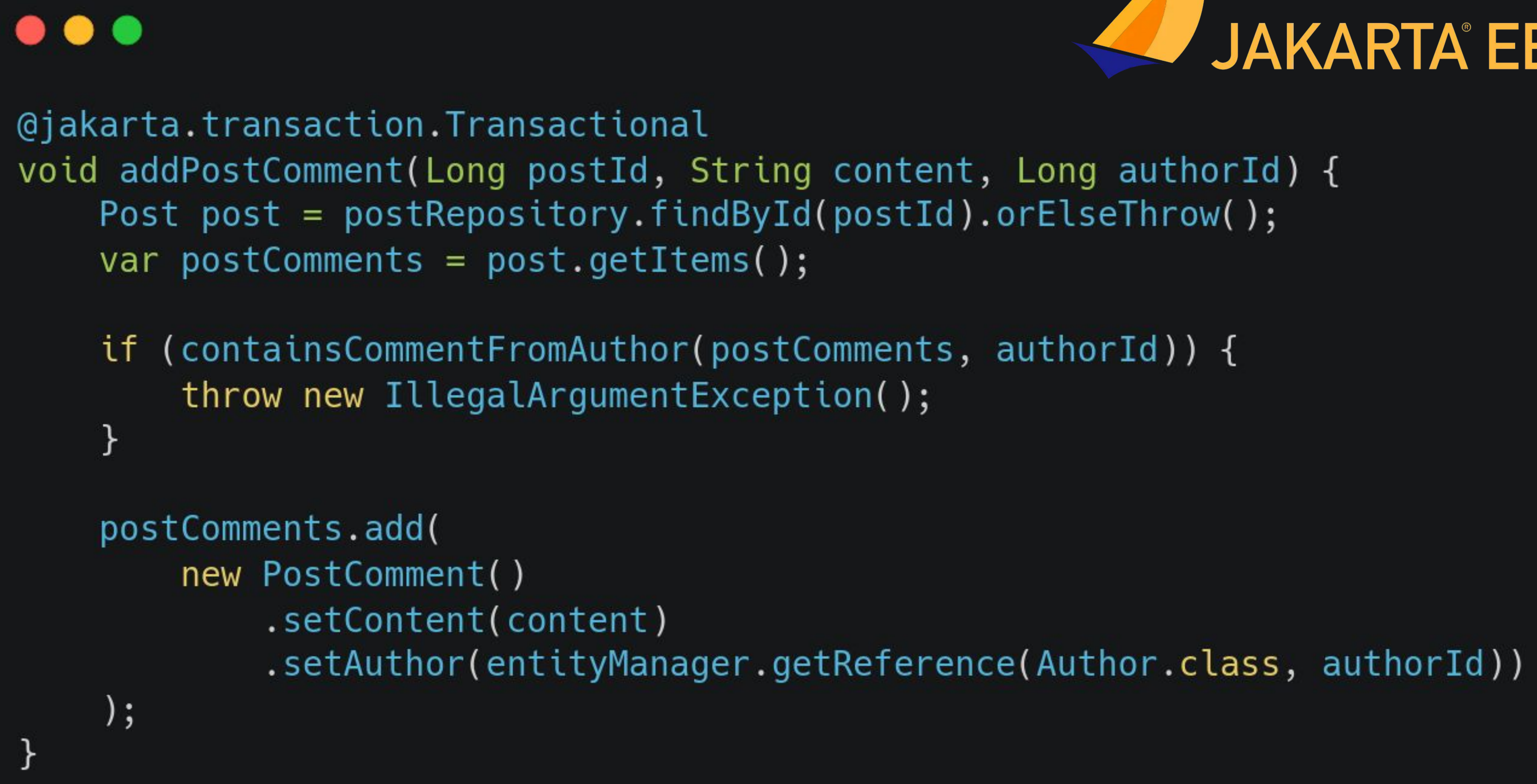
```
postComments.add(
    new PostComment()
        .setContent(content)
        .setAuthor(entityManager.getReferenceBy(Auhtor.class, authorId))
);
```



Transaction / EntityManager

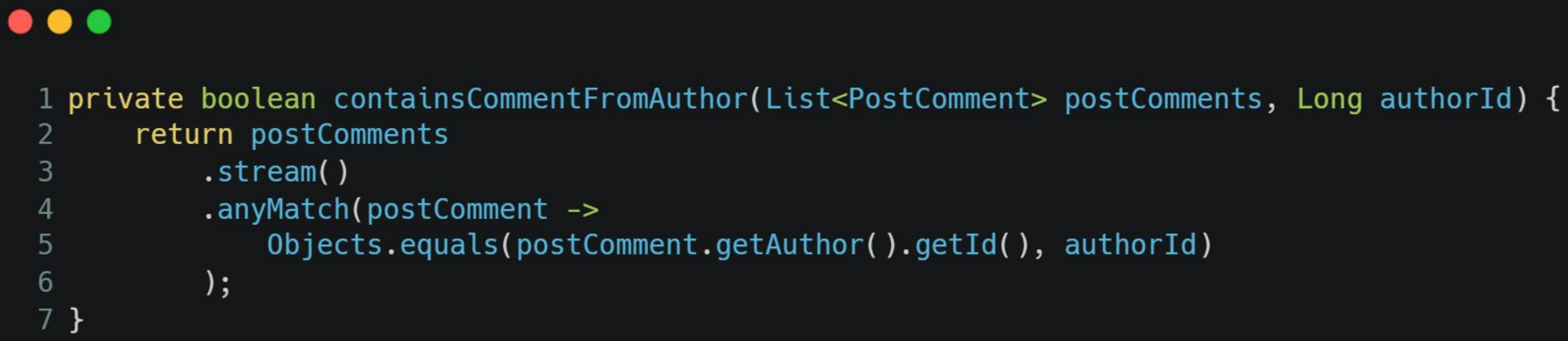
flush()

Post



```java
@jakarta.transaction.Transactional
void addPostComment(Long postId, String content, Long authorId) {
    Post post = postRepository.findById(postId).orElseThrow();
    var postComments = post.getItems();

    if (containsCommentFromAuthor(postComments, authorId)) {
        throw new IllegalArgumentException();
    }

    postComments.add(
        new PostComment()
            .setContent(content)
            .setAuthor(entityManager.getReference(Author.class, authorId))
    );
}
```

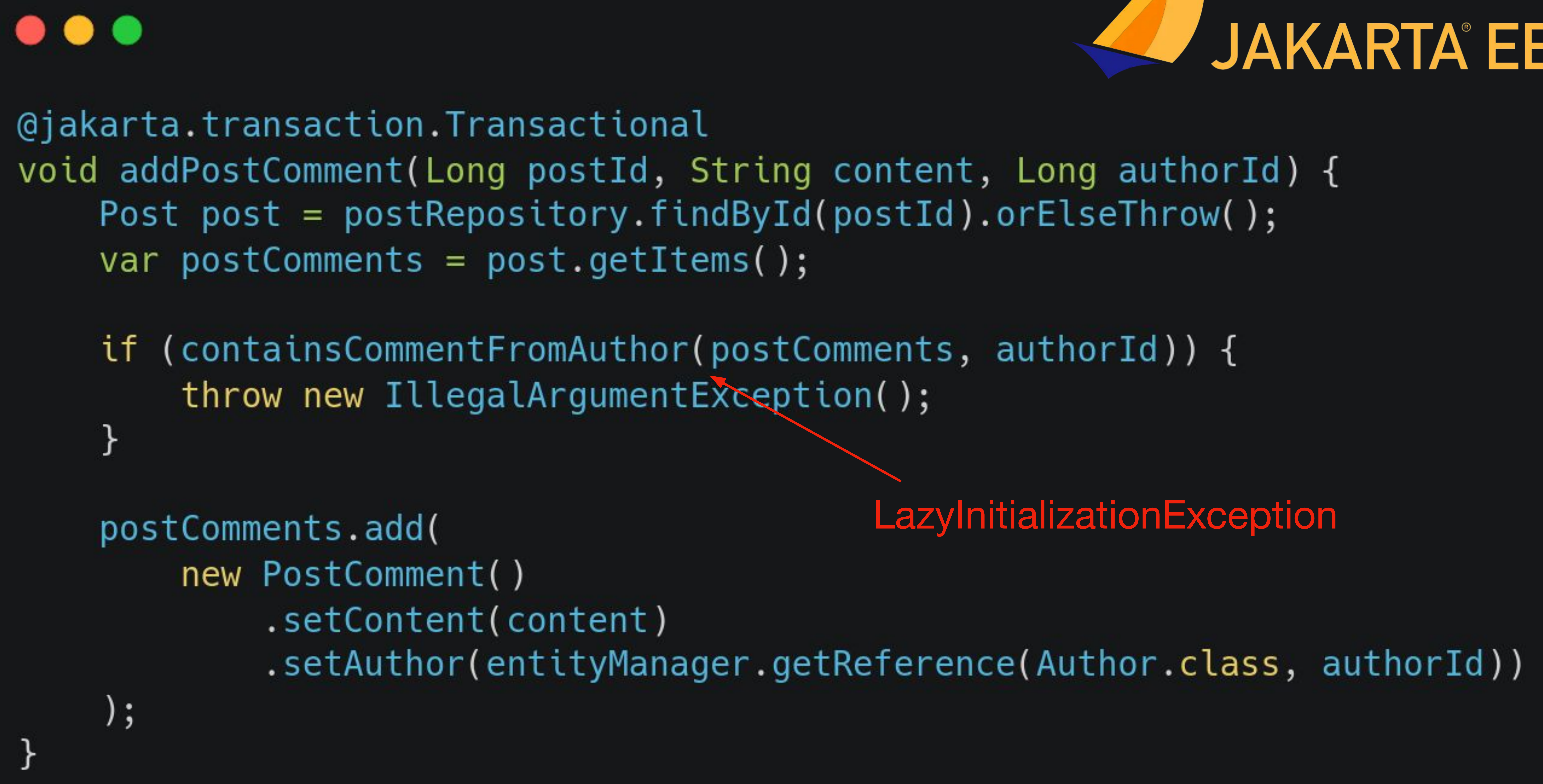
JAKARTA® EE



```java
private boolean containsCommentFromAuthor(List<PostComment> postComments, Long authorId) {
    return postComments
        .stream()
        .anyMatch(postComment ->
            Objects.equals(postComment.getAuthor().getId(), authorId)
        );
}
```

JAKARTA® EE

```
@jakarta.transaction.Transactional
void addPostComment(Long postId, String content, Long authorId) {
    Post post = postRepository.findById(postId).orElseThrow();
    var postComments = post.getItems();

    if (containsCommentFromAuthor(postComments, authorId)) {
        throw new IllegalArgumentException();
    }

    postComments.add(
        new PostComment()
            .setContent(content)
            .setAuthor(entityManager.getReference(Author.class, authorId))
    );
}
```

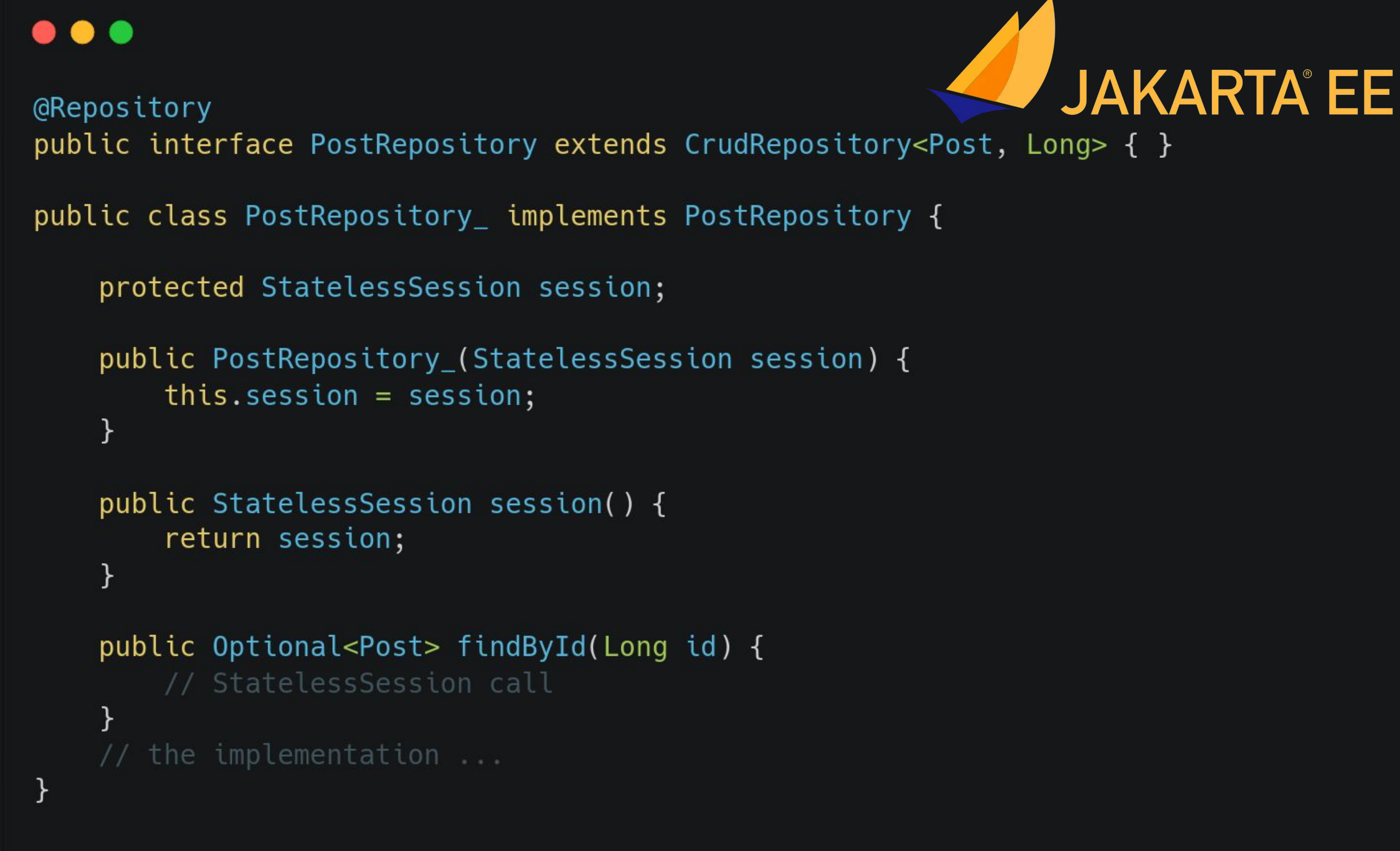


```
@jakarta.transaction.Transactional
void addPostComment(Long postId, String content, Long authorId) {
    Post post = postRepository.findById(postId).orElseThrow();
    var postComments = post.getItems();

    if (containsCommentFromAuthor(postComments, authorId)) {
        throw new IllegalArgumentException();
    }

    postComments.add(
        new PostComment()
            .setContent(content)
            .setAuthor(entityManager.getReference(Author.class, authorId))
    );
}
```

LazyInitializationException

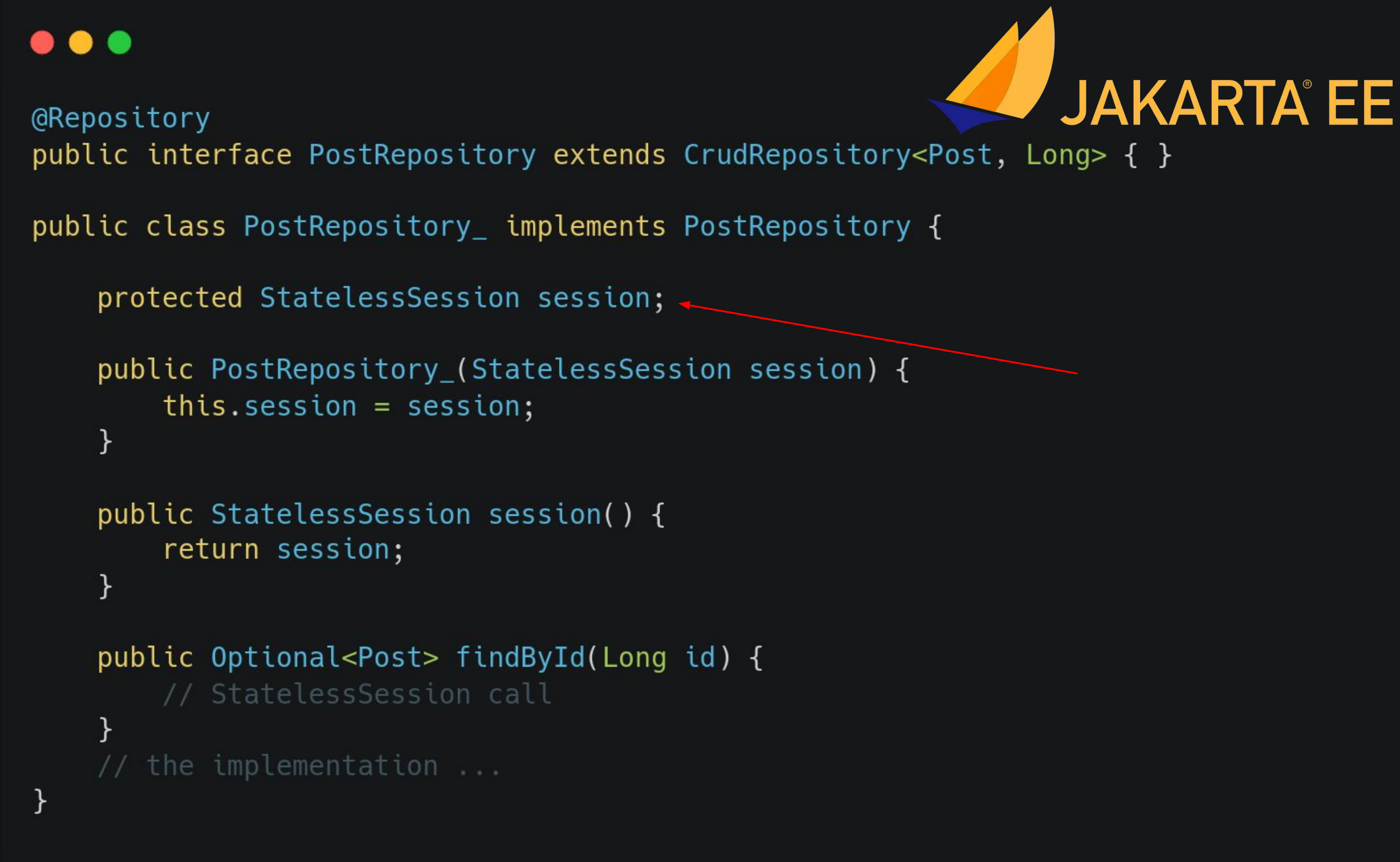
Stateless Repositories

JAKARTA® EE



```java
@Repository
public interface PostRepository extends CrudRepository<Post, Long> { }

public class PostRepository_ implements PostRepository {

    protected StatelessSession session;

    public PostRepository_(StatelessSession session) {
        this.session = session;
    }

    public StatelessSession session() {
        return session;
    }

    public Optional<Post> findById(Long id) {
        // StatelessSession call
    }
    // the implementation ...
}
```

JAKARTA® EE



```
@Repository
public interface PostRepository extends CrudRepository<Post, Long> { }

public class PostRepository_ implements PostRepository {

    protected StatelessSession session;

    public PostRepository_(StatelessSession session) {
        this.session = session;
    }

    public StatelessSession session() {
        return session;
    }

    public Optional<Post> findById(Long id) {
        // StatelessSession call
    }
    // the implementation ...
}
```

StatelessSession

1. Is not *yet* standardized, but probably would in JPA 4.0

StatelessSession

1. Is not *yet* standardized, but probably would in JPA 4.0
2. Is Stateless by definition, meaning, it does not have any Persistence Context backing it

StatelessSession

1. Is not *yet* standardized, but probably would in JPA 4.0

2. Is Stateless by definition, meaning, it does not have any Persistence Context backing it

3. Therefore, features like:
   1. Dirty checking
   2. Lazy Loading
   3. Cascading
   4. etc.

   are absent.

StatelessSession

1. Is not *yet* standardized, but probably would in JPA 4.0

2. Is Stateless by definition, meaning, it does not have any Persistence Context backing it

3. Therefore, features like:
   1. Dirty checking
   2. Lazy Loading
   3. Cascading
   4. etc.

   are absent.

4. Were introduced for simplicity and to intentionally ease the process of working with database from Hibernate

StatelessSession

1. Is not *yet* standardized, but probably would in JPA 4.0

2. Is Stateless by definition, meaning, it does not have any Persistence Context backing it

3. Therefore, features like:
    1. Dirty checking
    2. Lazy Loading
    3. Cascading
    4. etc.

   are absent.

4. Were introduced for simplicity and to intentionally ease the process of working with database from Hibernate

Spring Data JDBC?

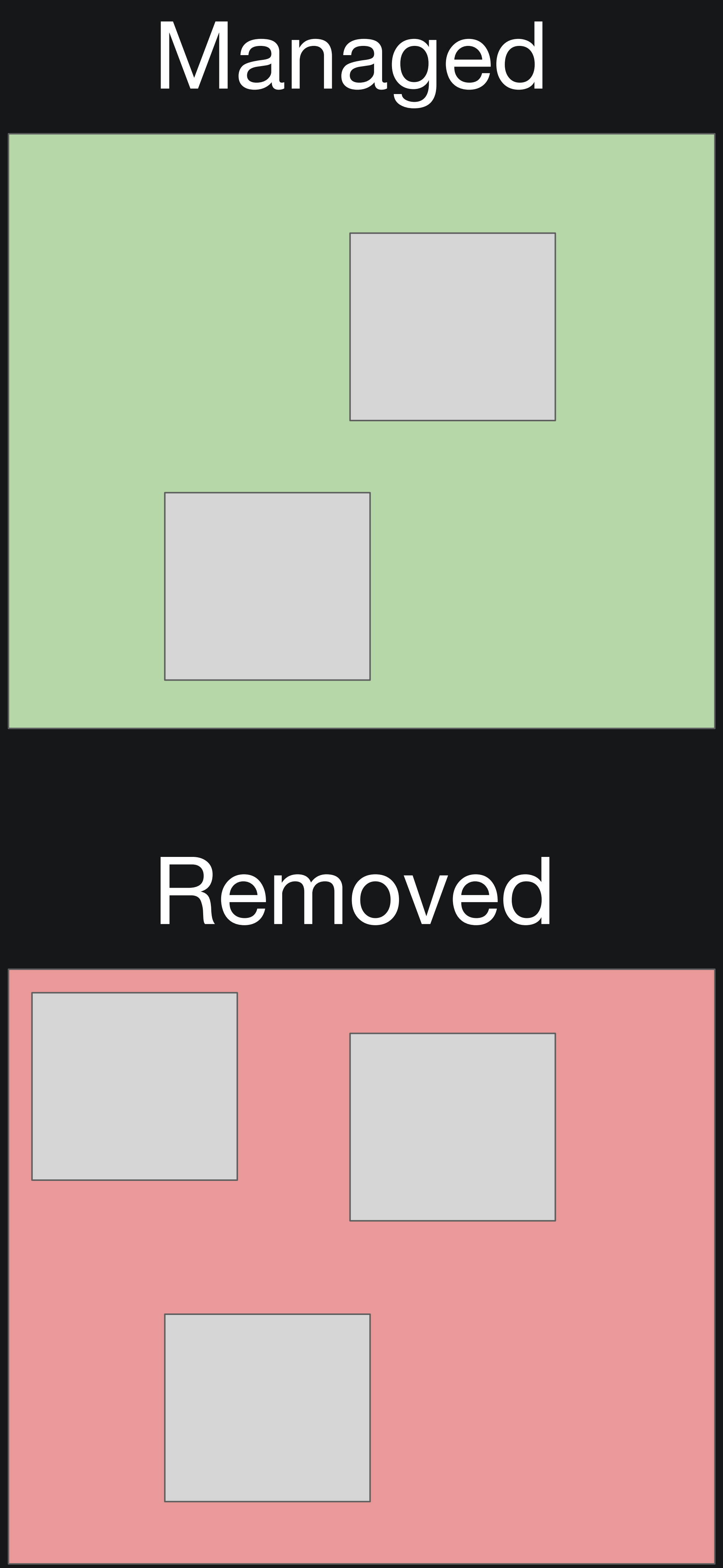
So, what is the problem for Spring Data JPA?

1. Spring Data JPA is not *really* designed to work with Hibernate only

So, what is the problem for Spring Data JPA?

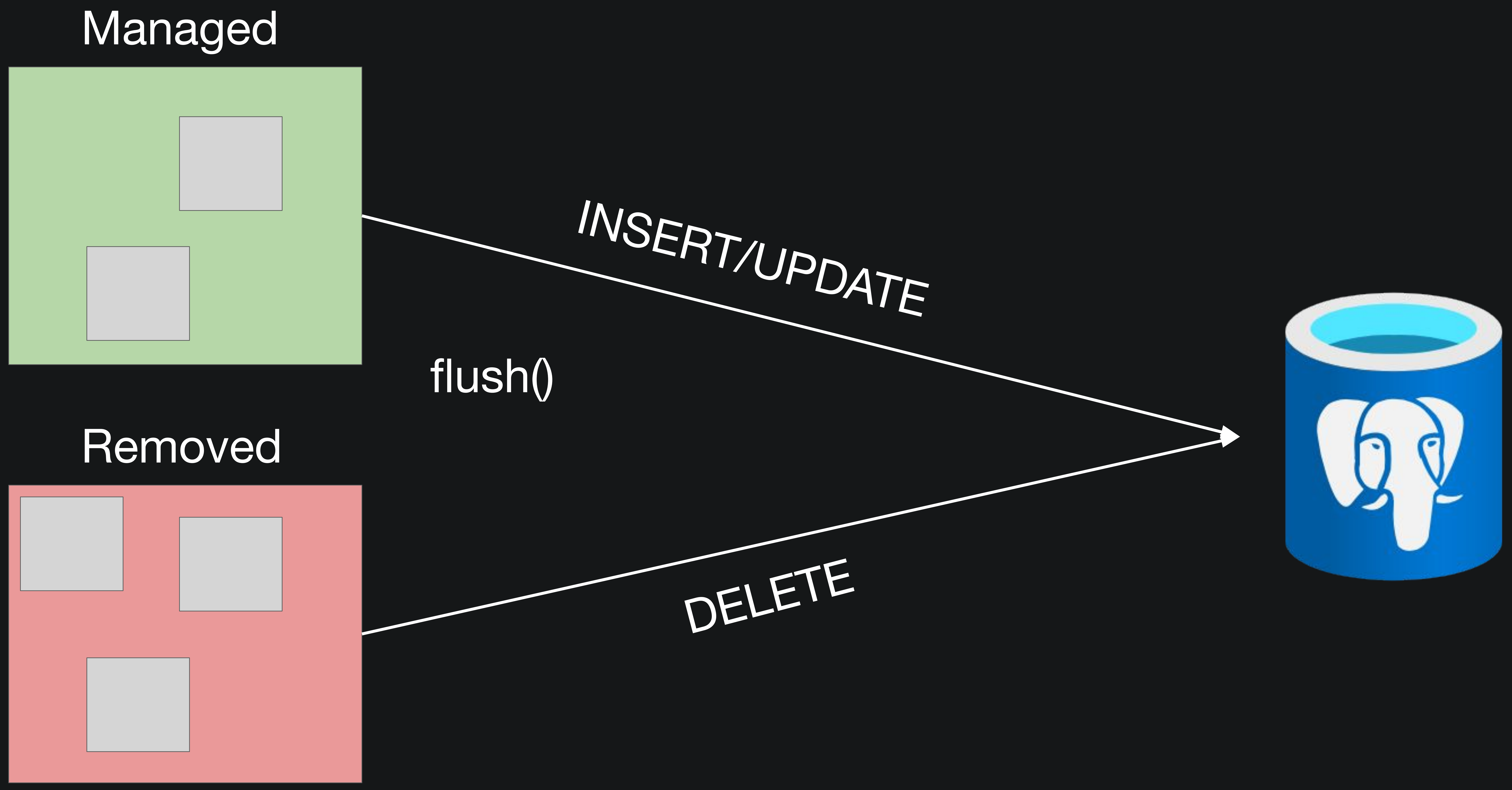
1. Spring Data JPA is not *really* designed to work with Hibernate only
2. As a consequence, Spring Data JPA always relied on the **stateful persistence context**.

```java
@NoRepositoryBean
public interface JpaRepository<T, ID> extends ListCrudRepository<T, ID> {

    /**
     * Flushes all pending changes to the database.
     */
    void flush();

    /**
     * Saves an entity and flushes changes instantly.
     *
     * @param entity entity to be saved. Must not be {@literal null}.
     * @return the saved entity
     */
    <S extends T> S saveAndFlush(S entity);

    /**
     * Returns a reference to the entity with the given identifier. Depending on how the JPA persistence provider is
     * implemented this is very likely to always return an instance and throw an
     * {@link jakarta.persistence.EntityNotFoundException} on first access. Some of them will reject invalid identifiers
     * immediately.
     *
     * @param id must not be {@literal null}.
     * @return a reference to the entity with the given identifier.
     * @see EntityManager#getReference(Class, Object) for details on when an exception is thrown.
     * @since 2.7
     */
    T getReferenceById(ID id);
}
```



Managed

Removed



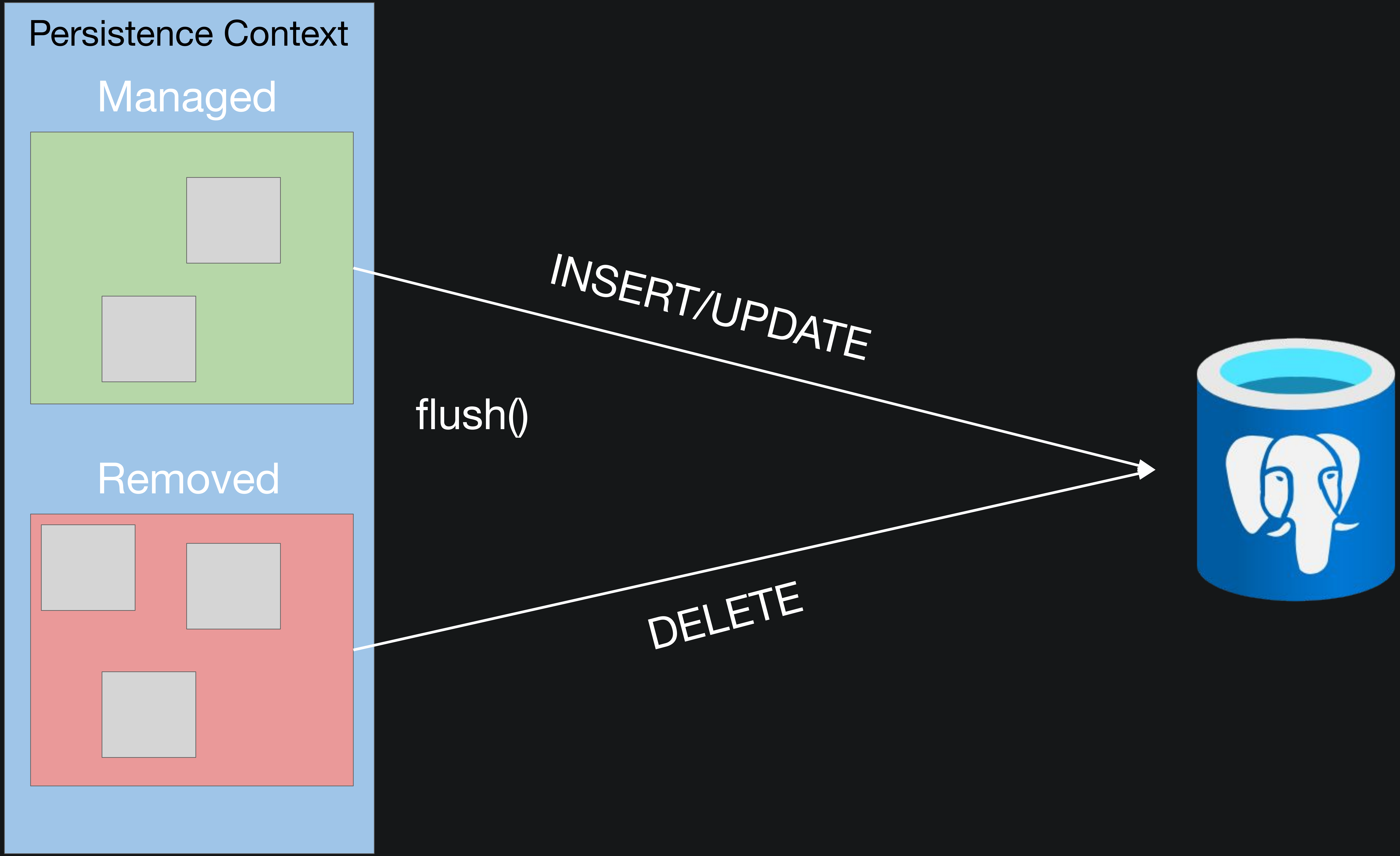
Managed

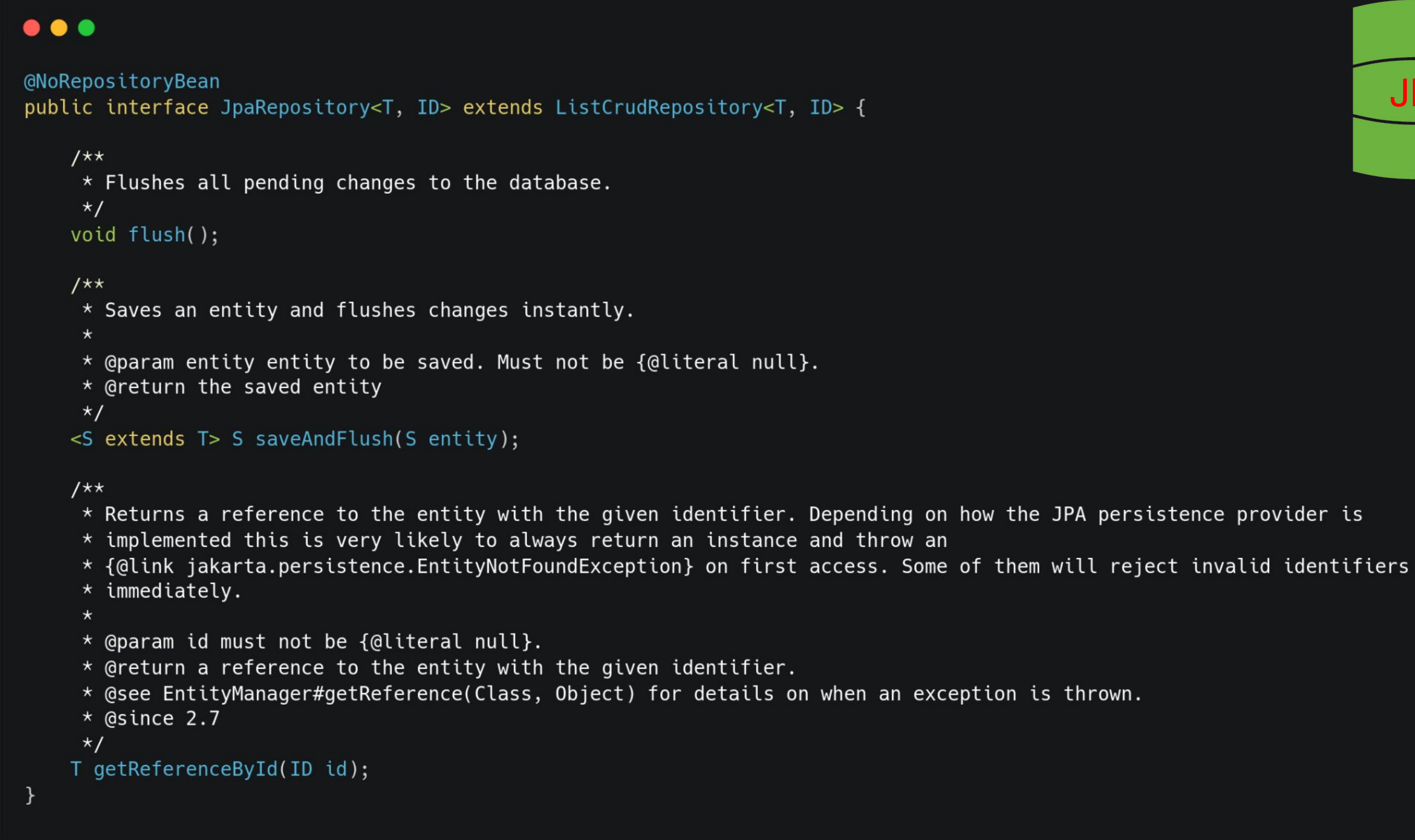
Removed

flush()

INSERT/UPDATE

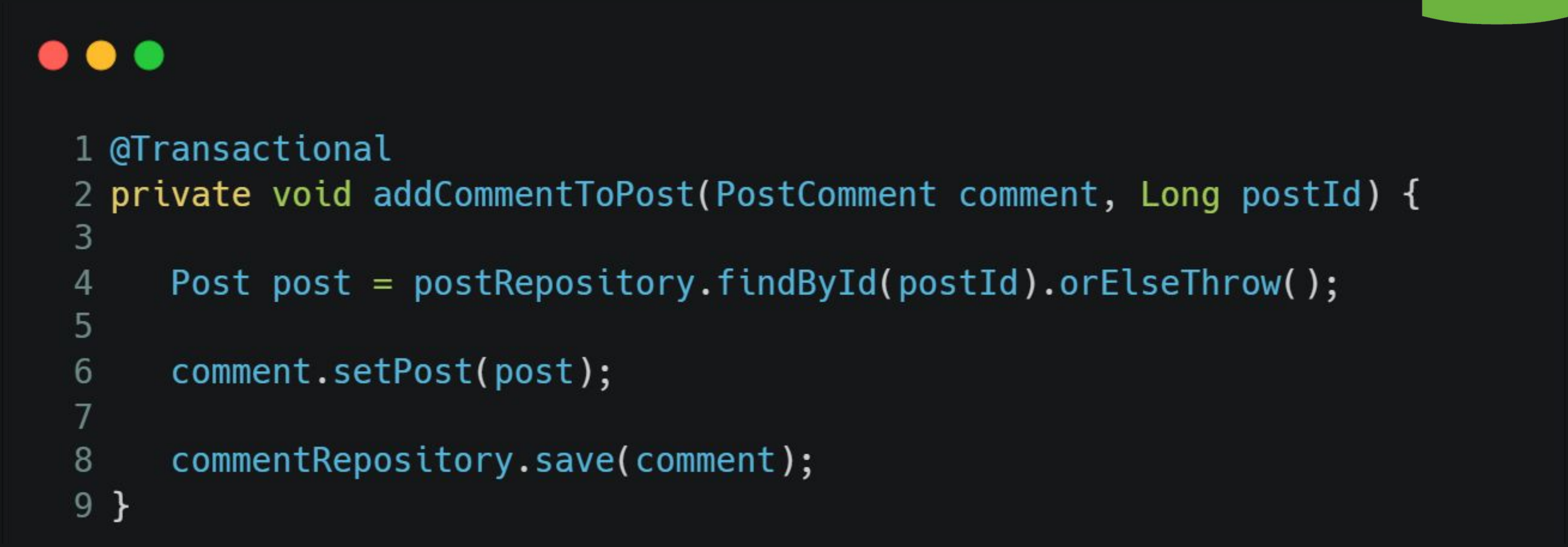
DELETE





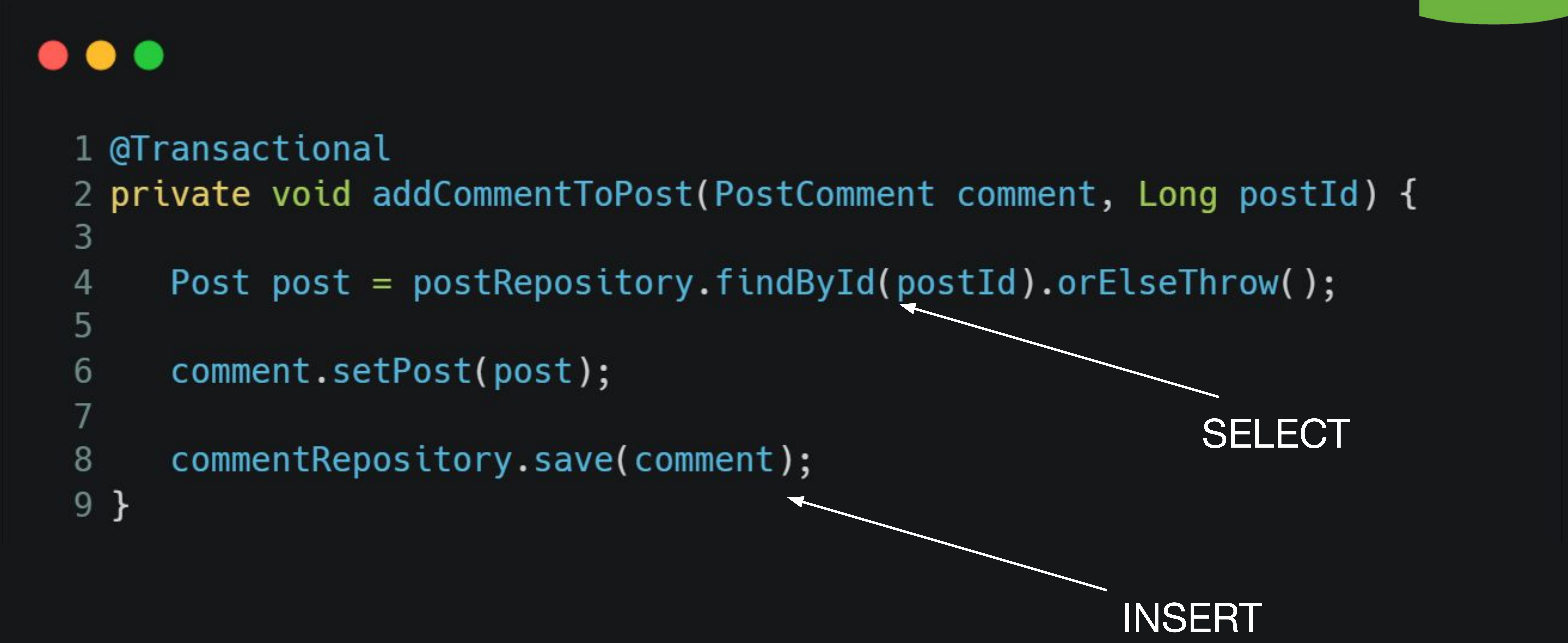
```
@NoRepositoryBean
public interface JpaRepository<T, ID> extends ListCrudRepository<T, ID> {

    /**
     * Flushes all pending changes to the database.
     */
    void flush();

    /**
     * Saves an entity and flushes changes instantly.
     *
     * @param entity entity to be saved. Must not be {@literal null}.
     * @return the saved entity
     */
    <S extends T> S saveAndFlush(S entity);

    /**
     * Returns a reference to the entity with the given identifier. Depending on how the JPA persistence provider is
     * implemented this is very likely to always return an instance and throw an
     * {@link jakarta.persistence.EntityNotFoundException} on first access. Some of them will reject invalid identifiers
     * immediately.
     *
     * @param id must not be {@literal null}.
     * @return a reference to the entity with the given identifier.
     * @see EntityManager#getReference(Class, Object) for details on when an exception is thrown.
     * @since 2.7
     */
    T getReferenceById(ID id);
}
```

JPA



```java
@Transactional
private void addCommentToPost(PostComment comment, Long postId) {

    Post post = postRepository.findById(postId).orElseThrow();

    comment.setPost(post);

    commentRepository.save(comment);
}
```

JPA



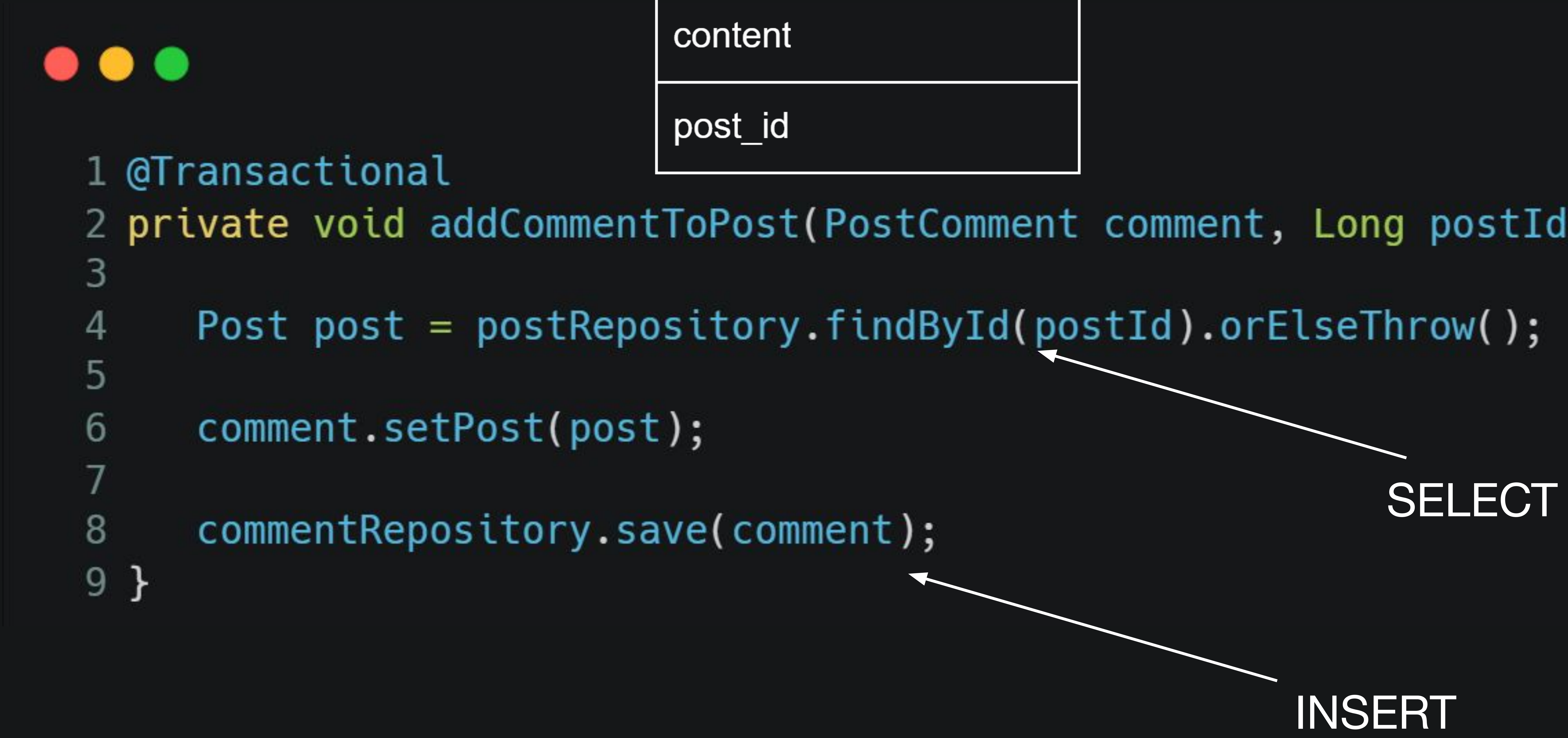
```java
@Transactional
private void addCommentToPost(PostComment comment, Long postId) {

    Post post = postRepository.findById(postId).orElseThrow();

    comment.setPost(post);

    commentRepository.save(comment);
}
```

SELECT

INSERT

post_comment
id
content
post_id

JPA



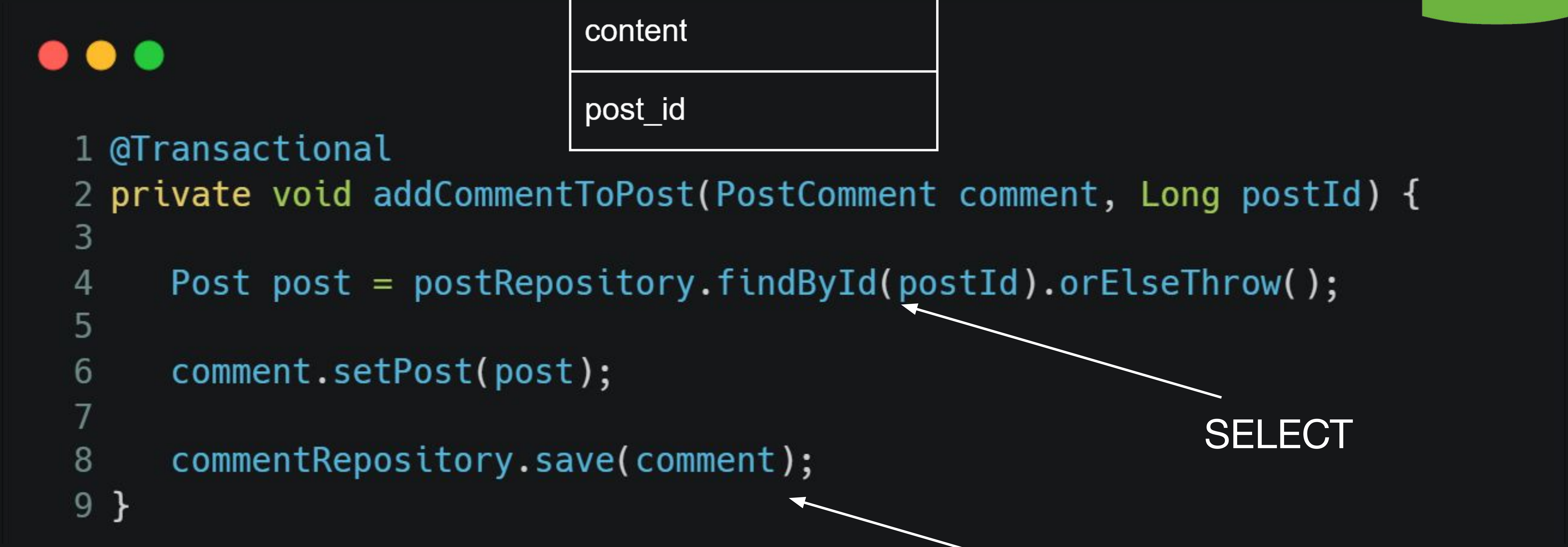
```java
@Transactional
private void addCommentToPost(PostComment comment, Long postId) {

    Post post = postRepository.findById(postId).orElseThrow();

    comment.setPost(post);

    commentRepository.save(comment);
}
```

SELECT

INSERT

JPA

post_comment
id
content
post_id



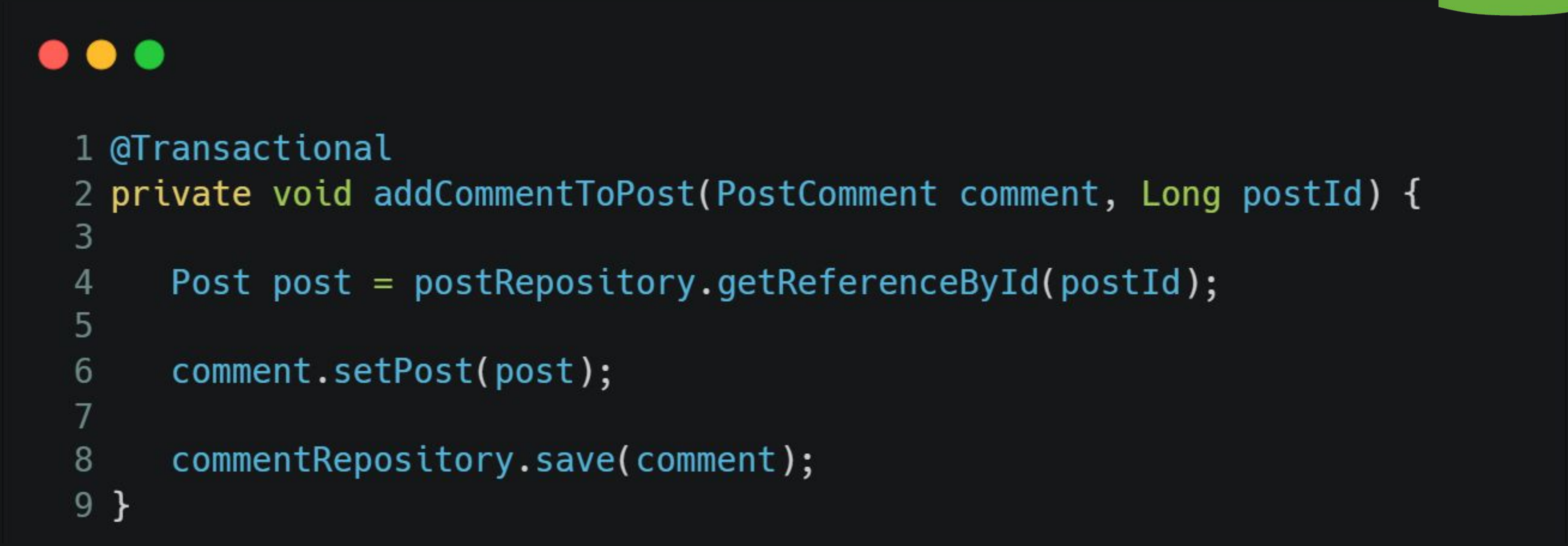
```java
@Transactional
private void addCommentToPost(PostComment comment, Long postId) {

    Post post = postRepository.findById(postId).orElseThrow();

    comment.setPost(post);

    commentRepository.save(comment);
}
```

SELECT

INSERT

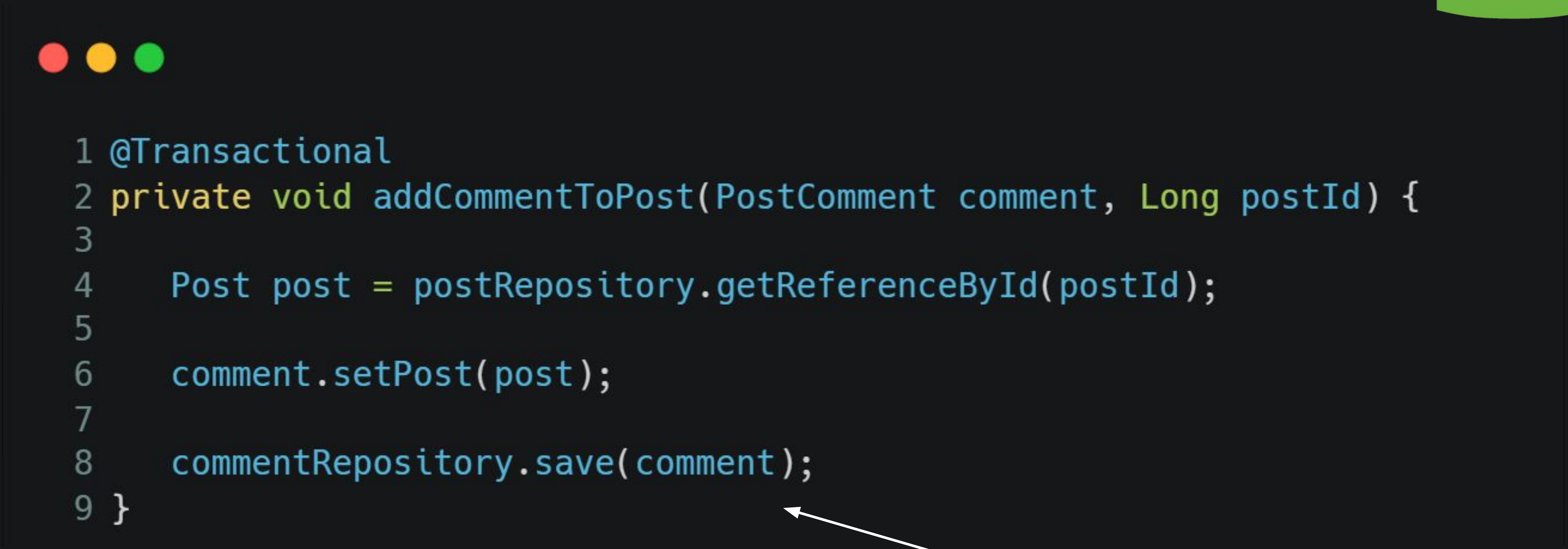
INSERT INTO post_comment VALUES(:id, :content, :post_id);

JPA



```java
@Transactional
private void addCommentToPost(PostComment comment, Long postId) {

   Post post = postRepository.getReferenceById(postId);

   comment.setPost(post);

   commentRepository.save(comment);
}
```

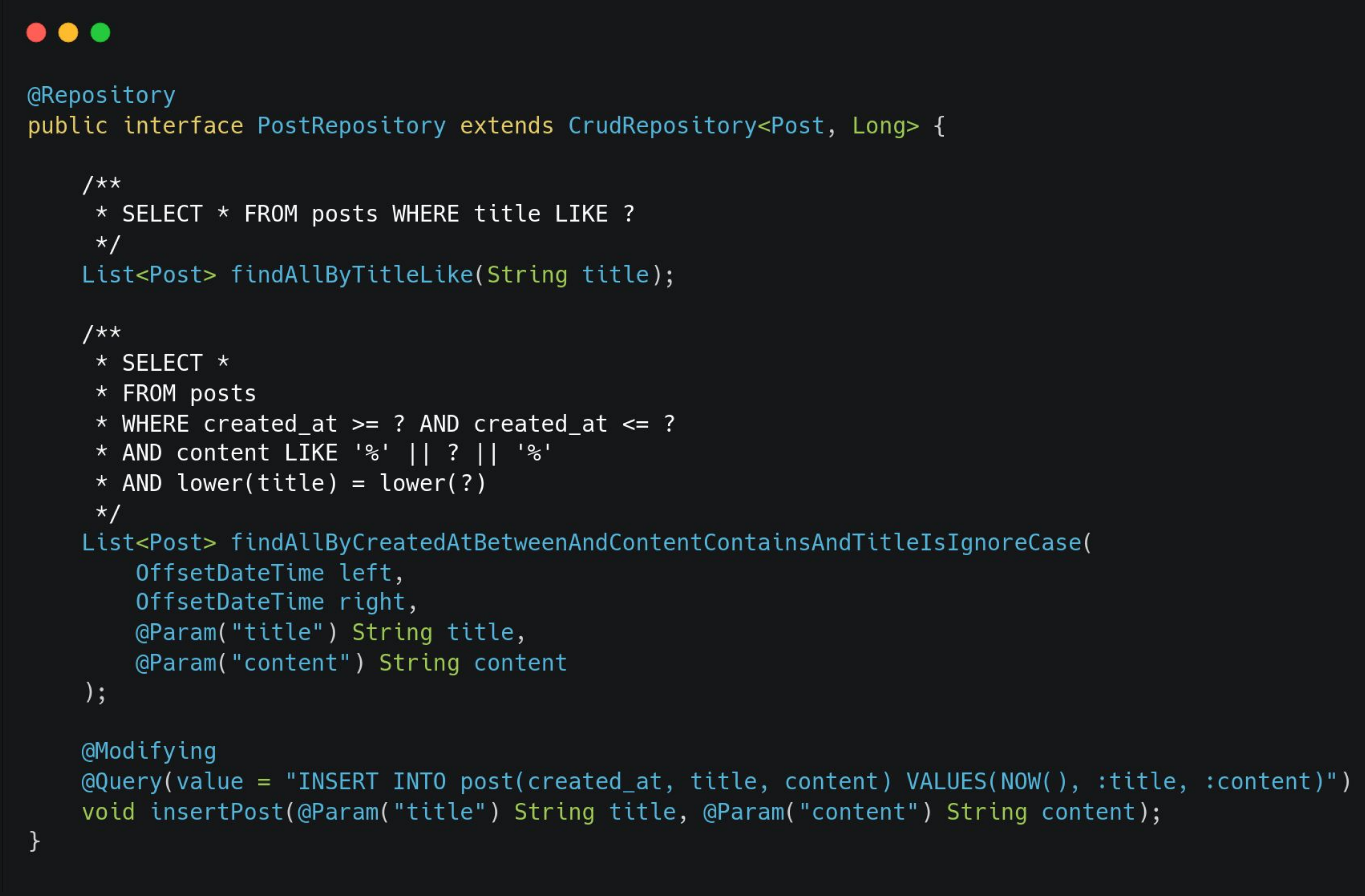
JPA



```java
@Transactional
private void addCommentToPost(PostComment comment, Long postId) {

    Post post = postRepository.getReferenceById(postId);

    comment.setPost(post);

    commentRepository.save(comment);
}
```

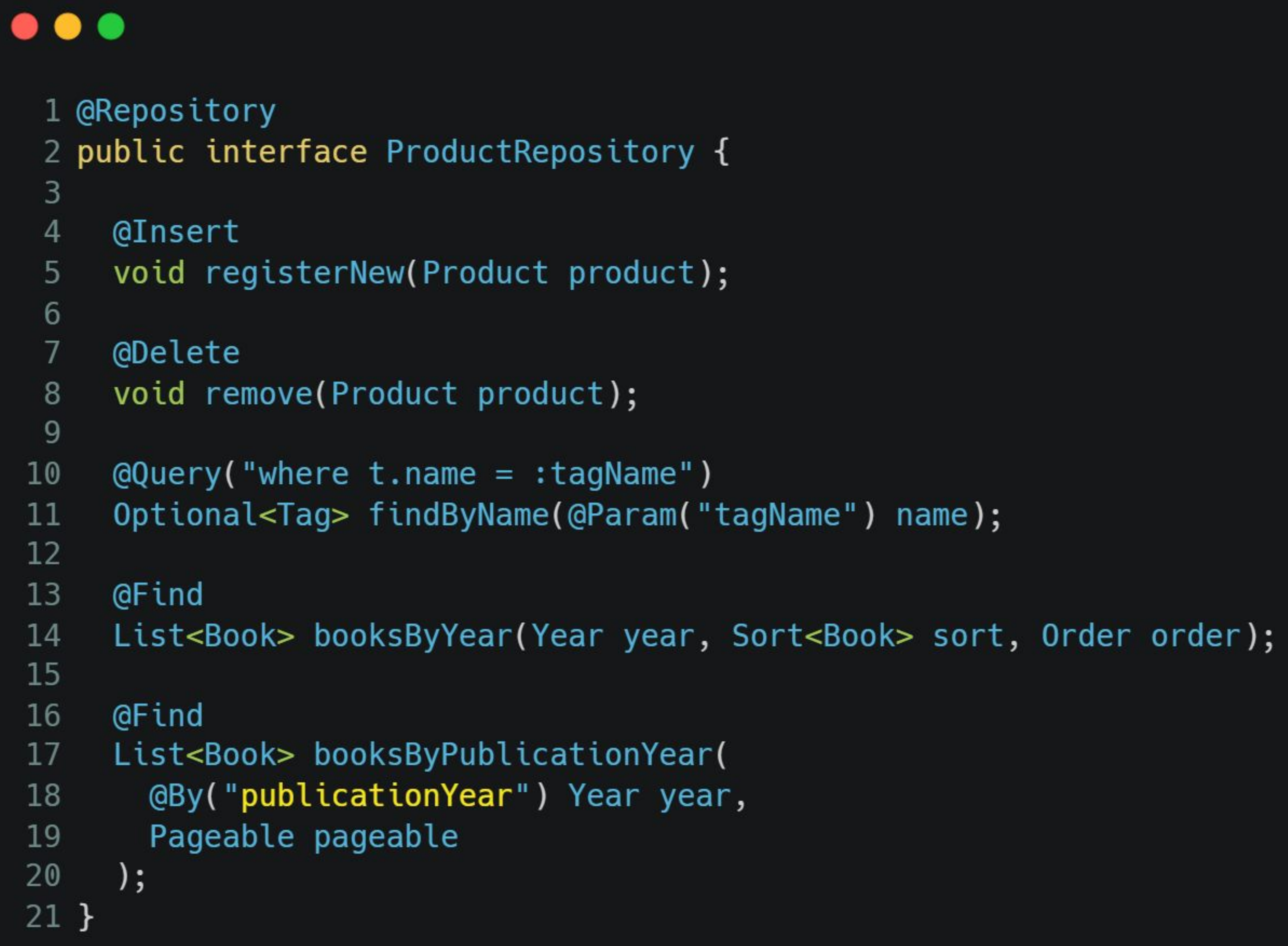
INSERT

Query By Method Name



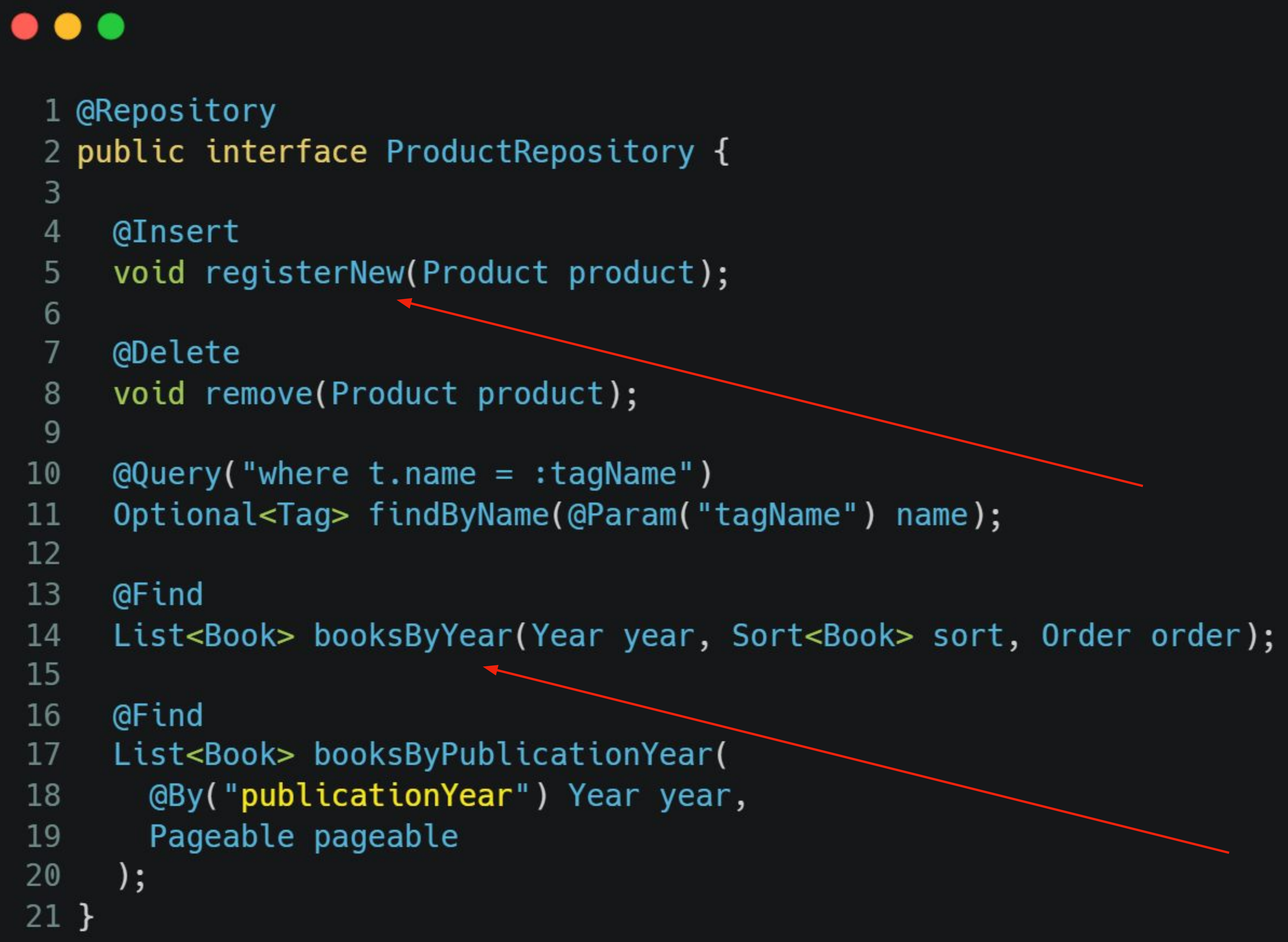
```java
@Repository
public interface PostRepository extends CrudRepository<Post, Long> {

    /**
     * SELECT * FROM posts WHERE title LIKE ?
     */
    List<Post> findAllByTitleLike(String title);

    /**
     * SELECT *
     * FROM posts
     * WHERE created_at >= ? AND created_at <= ?
     * AND content LIKE '%' || ? || '%'
     * AND lower(title) = lower(?)
     */
    List<Post> findAllByCreatedAtBetweenAndContentContainsAndTitleIsIgnoreCase(
        OffsetDateTime left,
        OffsetDateTime right,
        @Param("title") String title,
        @Param("content") String content
    );

    @Modifying
    @Query(value = "INSERT INTO post(created_at, title, content) VALUES(NOW(), :title, :content)")
    void insertPost(@Param("title") String title, @Param("content") String content);
}
```

JAKARTA® EE



```java
@Repository
public interface ProductRepository {

  @Insert
  void registerNew(Product product);

  @Delete
  void remove(Product product);

  @Query("where t.name = :tagName")
  Optional<Tag> findByName(@Param("tagName") name);

  @Find
  List<Book> booksByYear(Year year, Sort<Book> sort, Order order);

  @Find
  List<Book> booksByPublicationYear(
    @By("publicationYear") Year year,
    Pageable pageable
  );
}
```

JAKARTA® EE



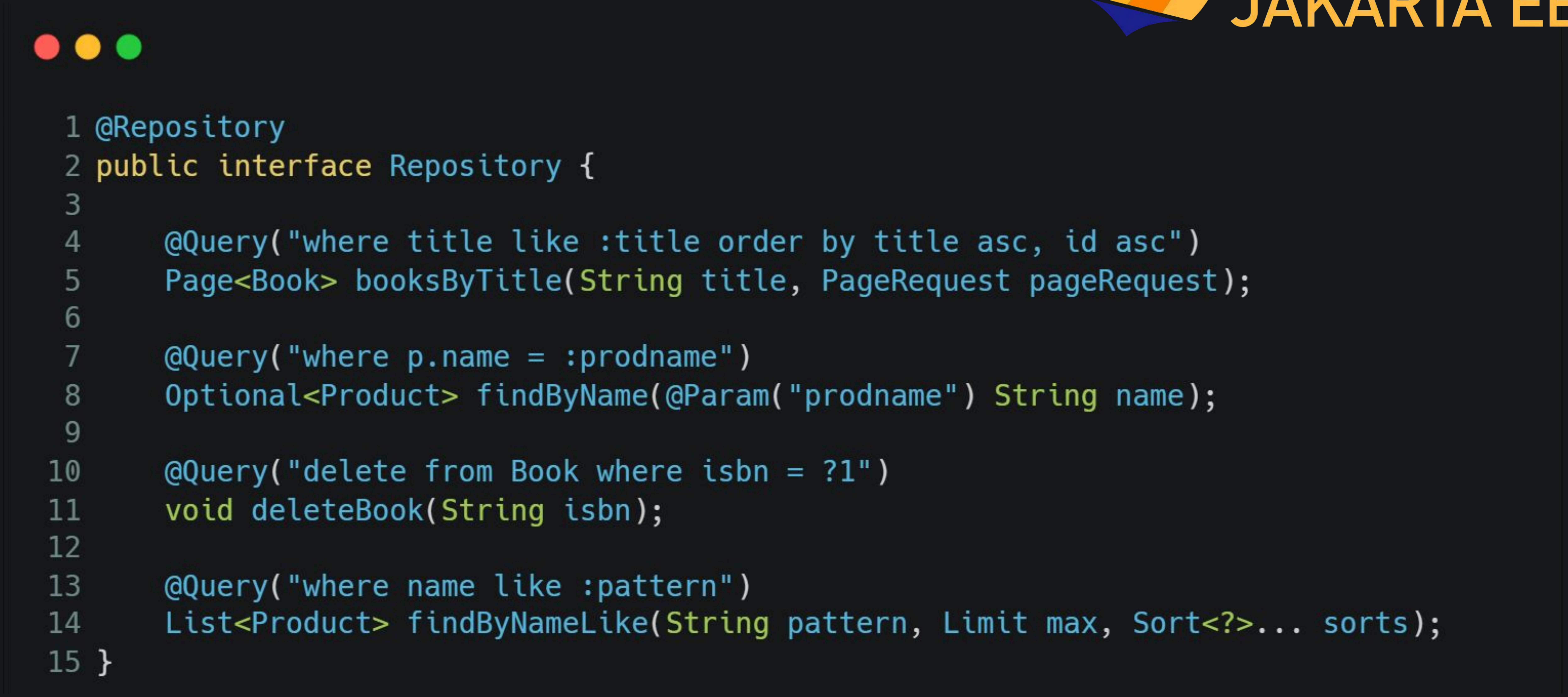
```java
@Repository
public interface ProductRepository {

  @Insert
  void registerNew(Product product);

  @Delete
  void remove(Product product);

  @Query("where t.name = :tagName")
  Optional<Tag> findByName(@Param("tagName") name);

  @Find
  List<Book> booksByYear(Year year, Sort<Book> sort, Order order);

  @Find
  List<Book> booksByPublicationYear(
    @By("publicationYear") Year year,
    Pageable pageable
  );
}
```

What if I want non-trivial search operators?

1. Or, which is preferable, use JDQL (Jakarta Data Query Language)

2. Dynamic query building (like Criteria API in JPA) is in discussion

JAKARTA® EE

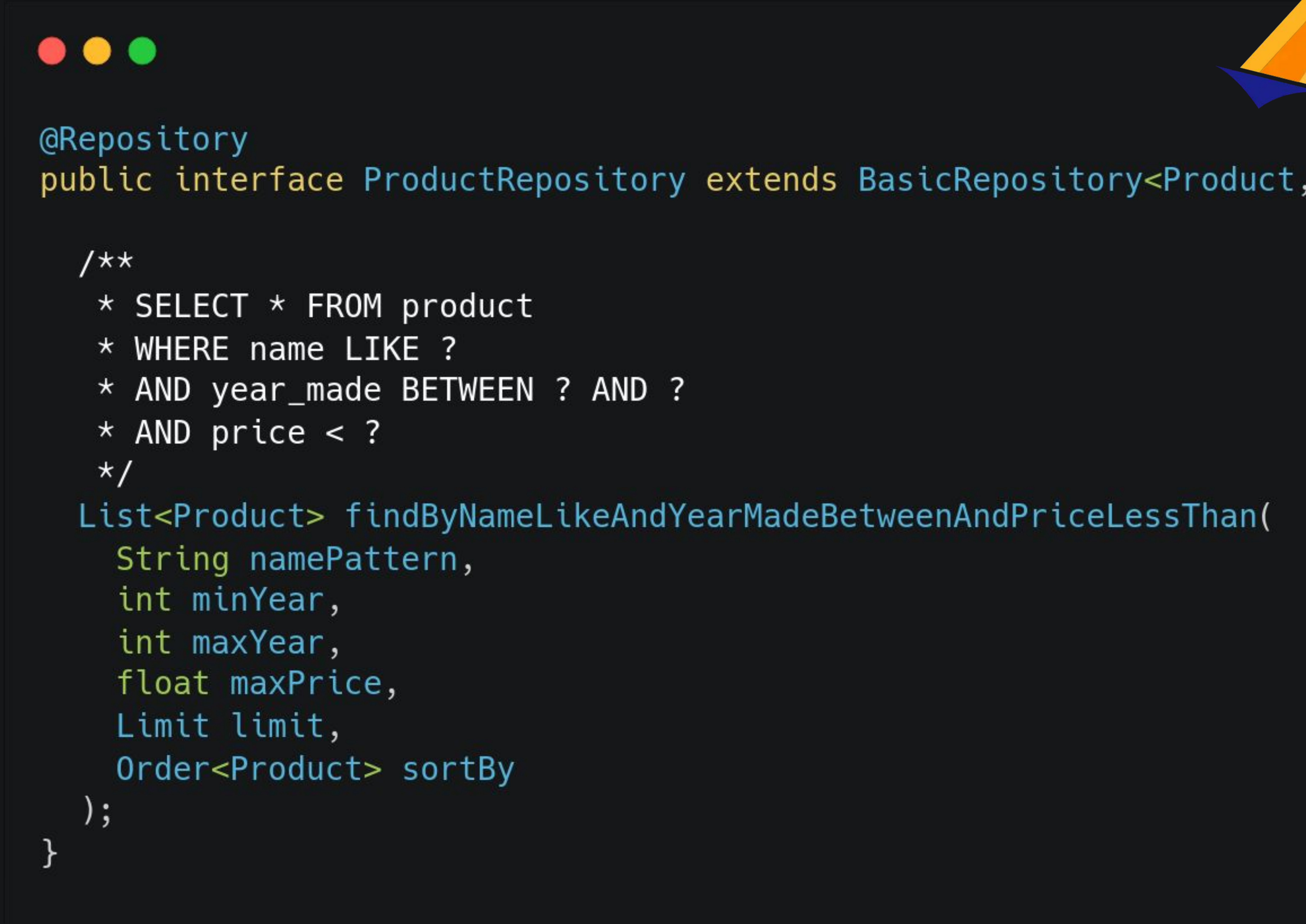


```java
@Repository
public interface Repository {

    @Query("where title like :title order by title asc, id asc")
    Page<Book> booksByTitle(String title, PageRequest pageRequest);

    @Query("where p.name = :prodname")
    Optional<Product> findByName(@Param("prodname") String name);

    @Query("delete from Book where isbn = ?1")
    void deleteBook(String isbn);

    @Query("where name like :pattern")
    List<Product> findByNameLike(String pattern, Limit max, Sort<?>... sorts);
}
```

What if I want non-trivial search operators?

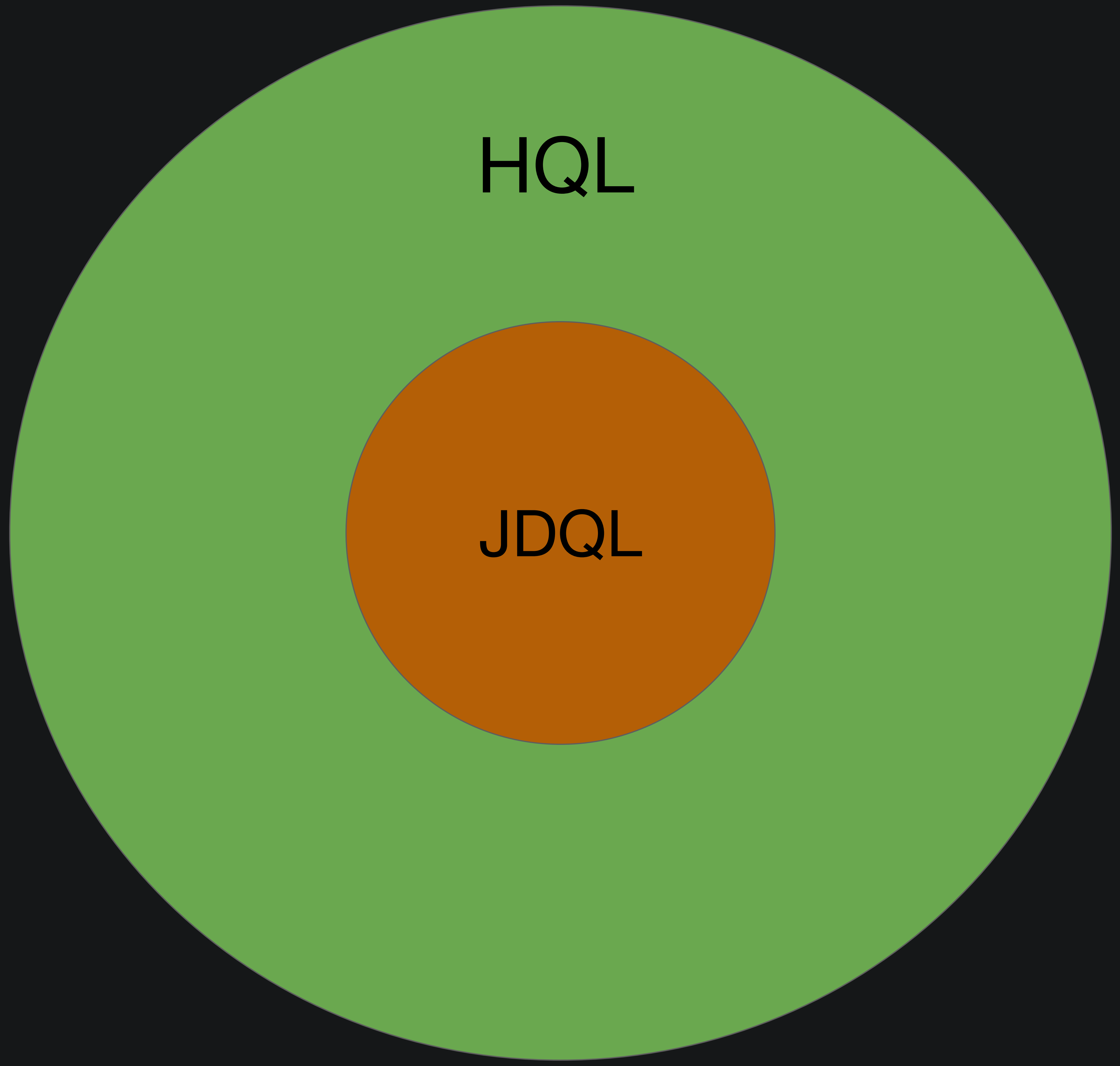
1. Or, which is preferable, use JDQL (Jakarta Data Query Language)

2. Dynamic query building (like Criteria API in JPA) is in discussion



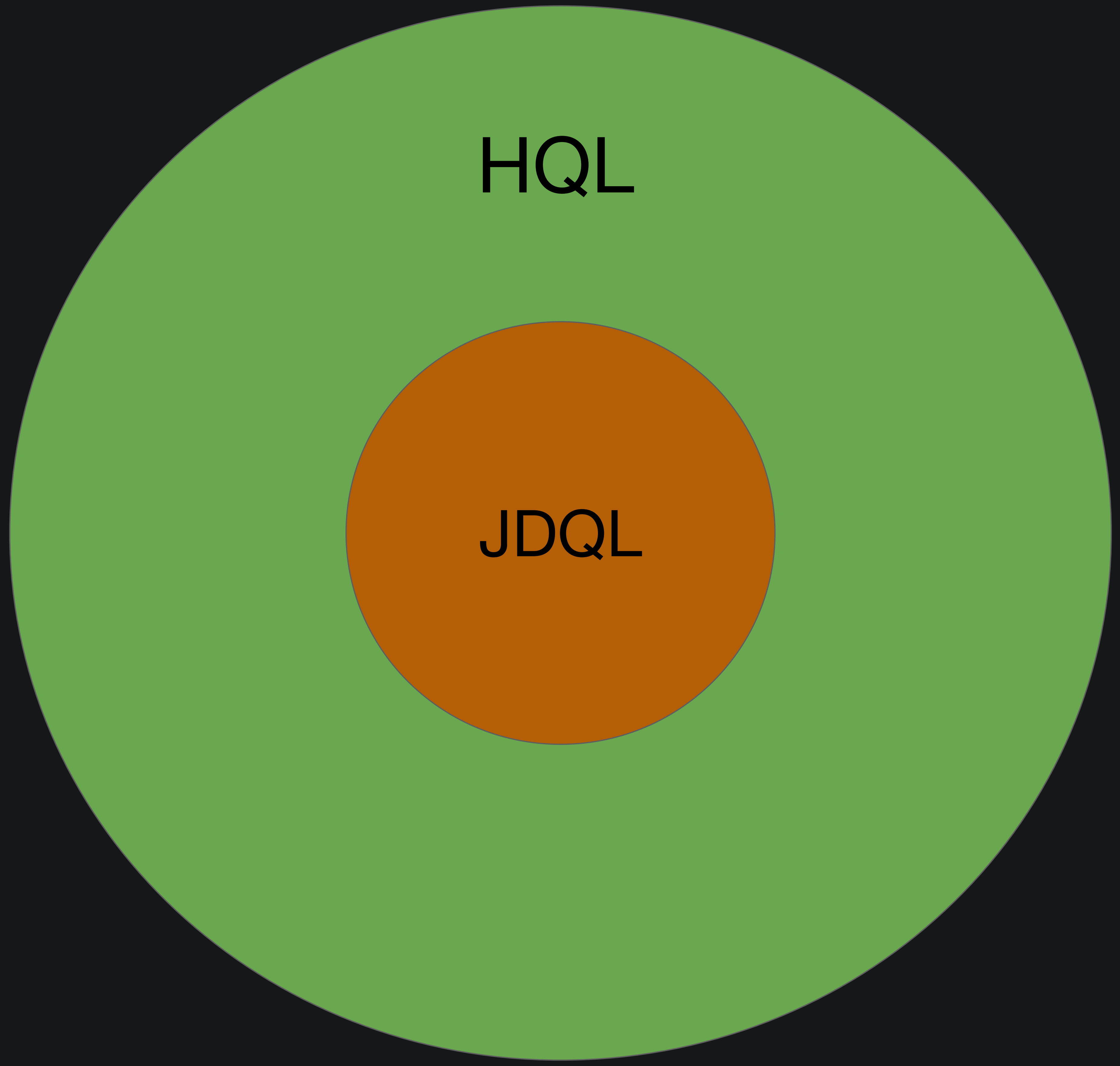
```
@Repository
public interface ProductRepository extends BasicRepository<Product, Long> {

  /**
   * SELECT * FROM product
   * WHERE name LIKE ?
   * AND year_made BETWEEN ? AND ?
   * AND price < ?
   */
  List<Product> findByNameLikeAndYearMadeBetweenAndPriceLessThan(
    String namePattern,
    int minYear,
    int maxYear,
    float maxPrice,
    Limit limit,
    Order<Product> sortBy
  );
}
```

What Spring Data module can implement Jakarta Data?



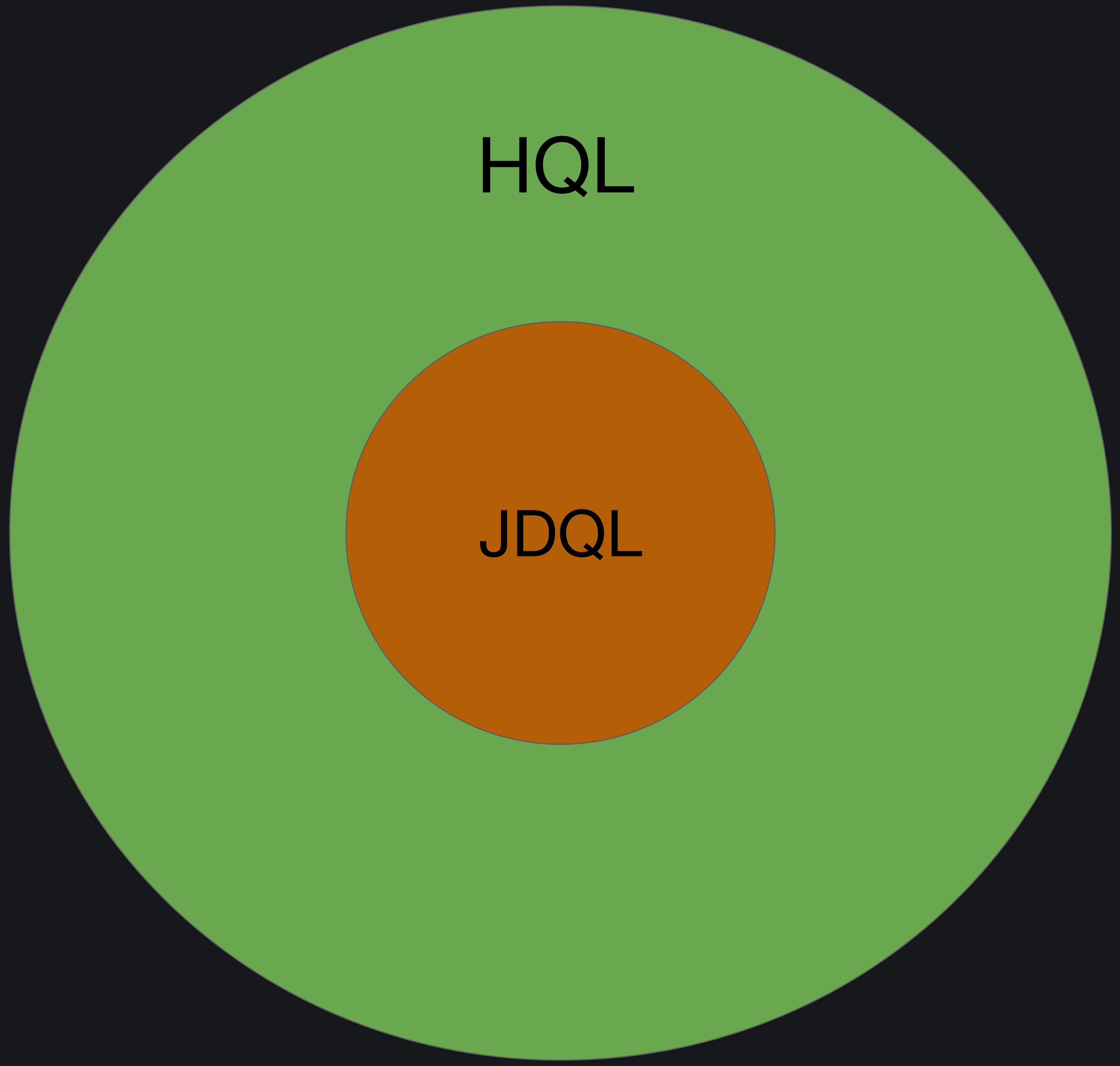
- Spring Data JPA
- Spring Data JDBC
- Other Spring Data Modules?

What Spring Data module can implement Jakarta Data?



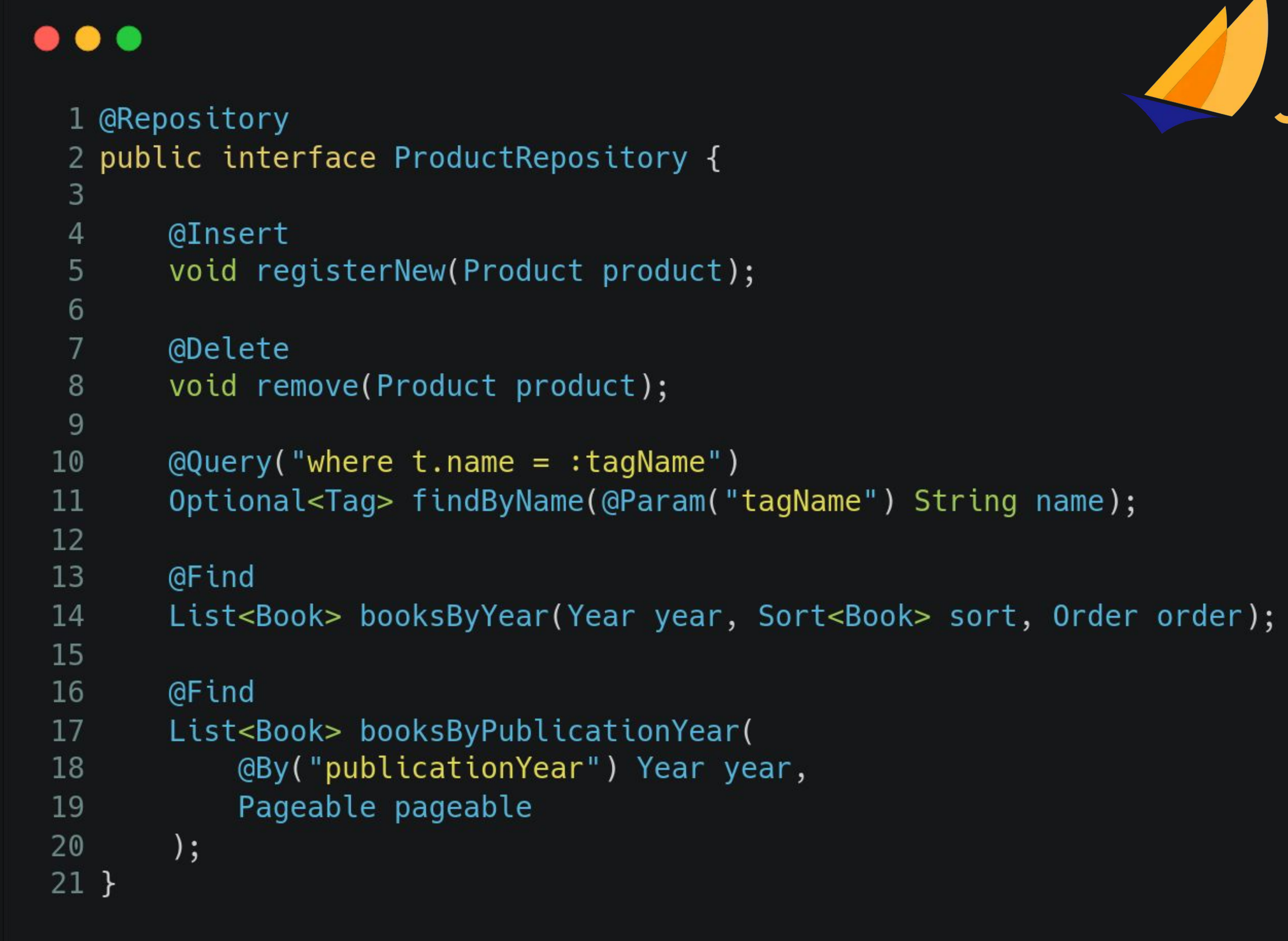
- ~~Spring Data JPA~~ Stateless Repos!
- Spring Data JDBC
- Other Spring Data Modules

What Spring Data module can implement Jakarta Data?



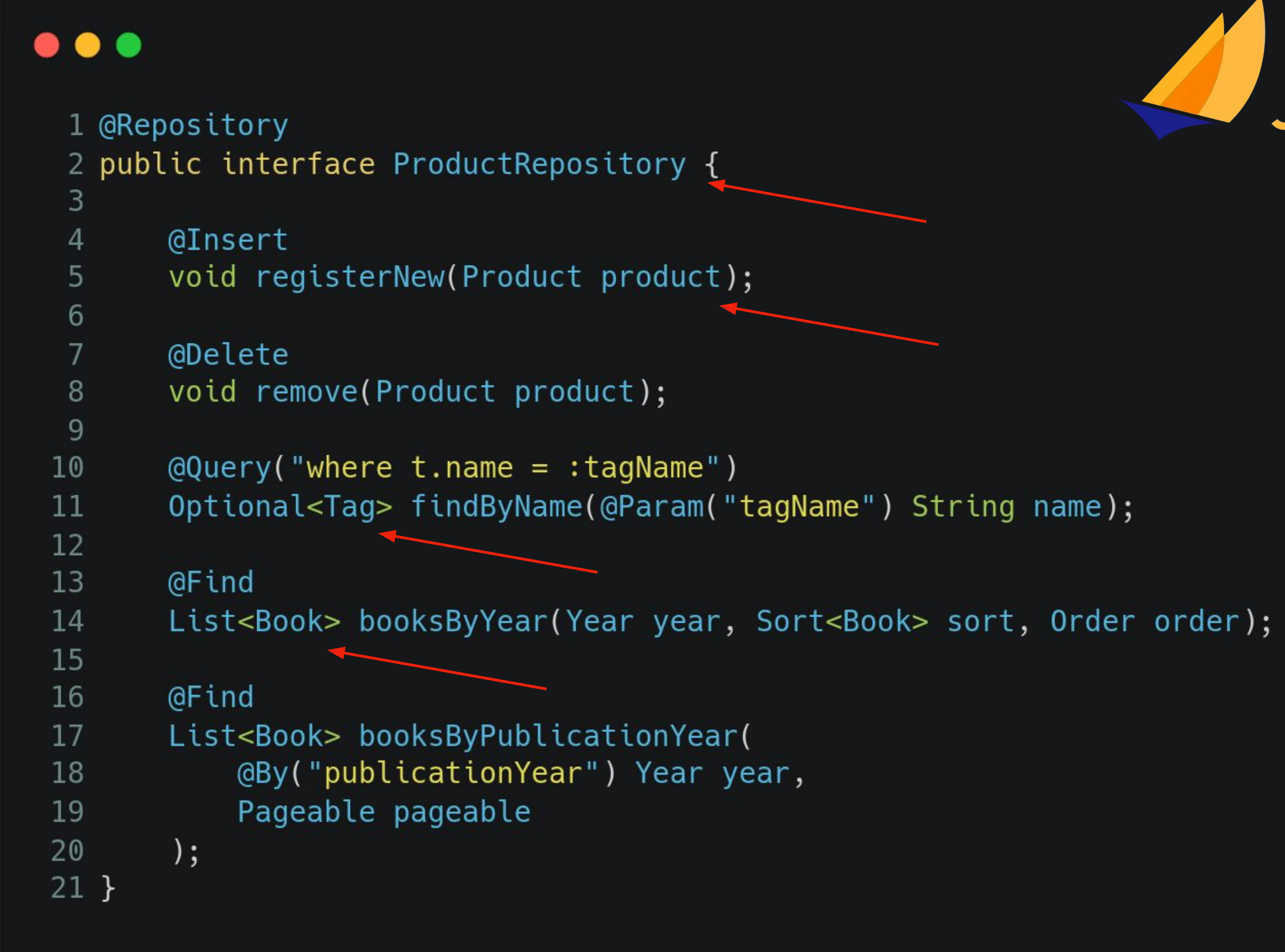
- ~~Spring Data JPA~~ Stateless Repos!
- Spring Data JDBC
- Other Spring Data Modules

Repository Concept



```java
@Repository
public interface ProductRepository {

    @Insert
    void registerNew(Product product);

    @Delete
    void remove(Product product);

    @Query("where t.name = :tagName")
    Optional<Tag> findByName(@Param("tagName") String name);

    @Find
    List<Book> booksByYear(Year year, Sort<Book> sort, Order order);

    @Find
    List<Book> booksByPublicationYear(
        @By("publicationYear") Year year,
        Pageable pageable
    );
}
```

JAKARTA® EE



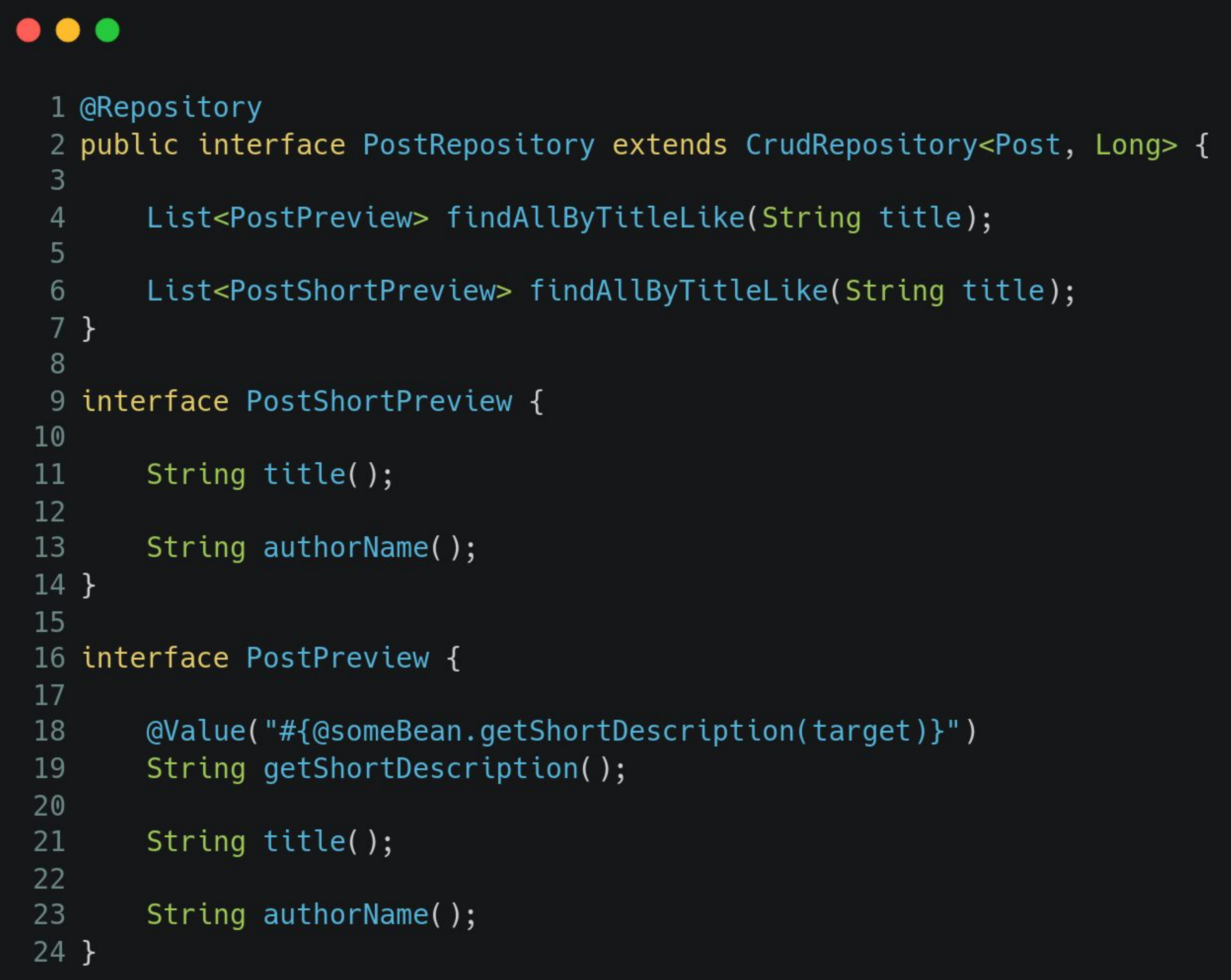
```java
@Repository
public interface ProductRepository {

    @Insert
    void registerNew(Product product);

    @Delete
    void remove(Product product);

    @Query("where t.name = :tagName")
    Optional<Tag> findByName(@Param("tagName") String name);

    @Find
    List<Book> booksByYear(Year year, Sort<Book> sort, Order order);

    @Find
    List<Book> booksByPublicationYear(
        @By("publicationYear") Year year,
        Pageable pageable
    );
}
```

Repositories Design Principles

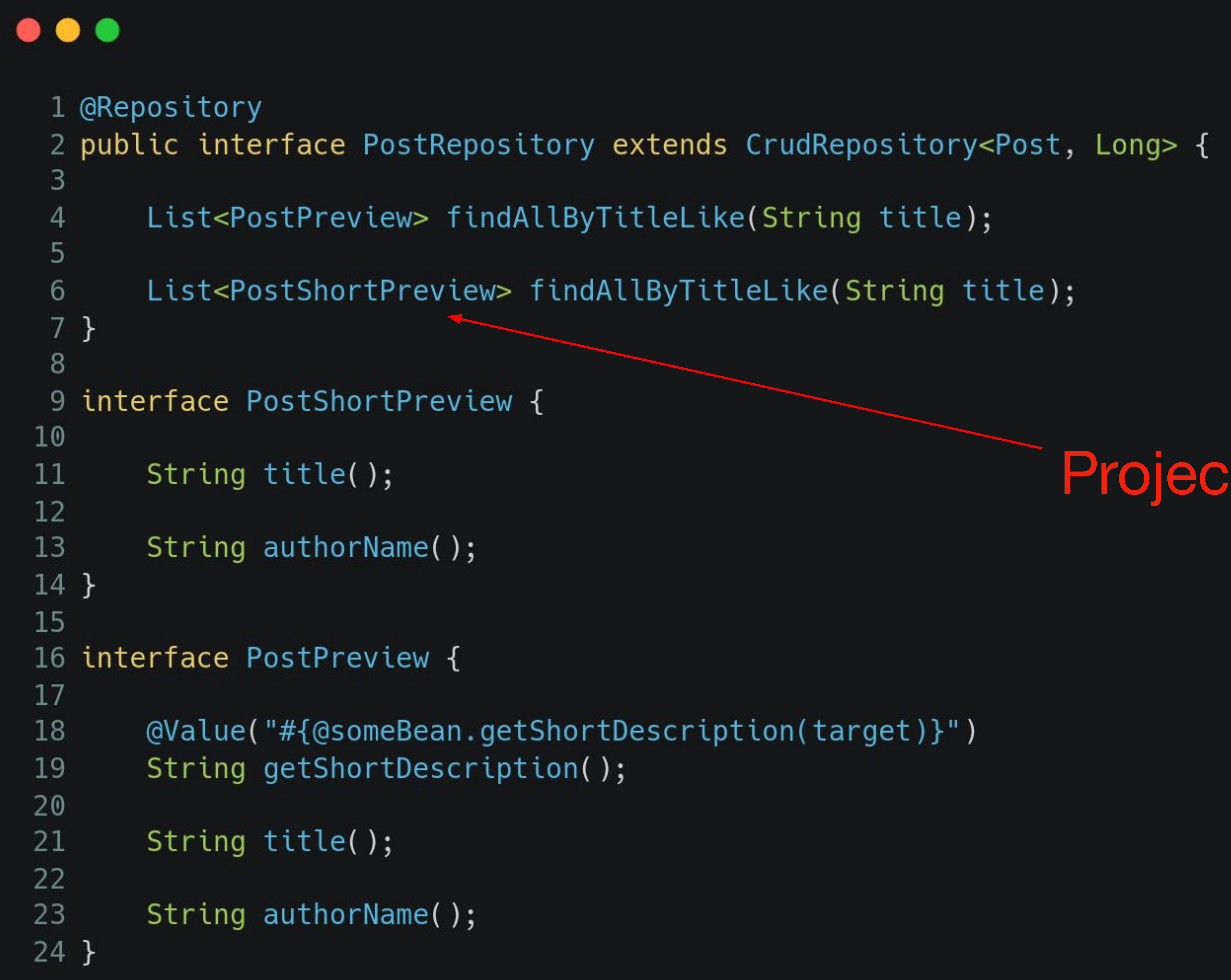
1. Jakarta Data does not require a repository to be bound to a single entity type.

2. Jakarta Data assumes, that it can resolve the entity type for given operation by using either:
   1. returned entity type
   2. or parameter type

```java
@NoRepositoryBean
public interface CrudRepository<T, ID> extends Repository<T, ID> {

    boolean existsById(ID id);

    long count();

    void deleteAll();

    void deleteById(ID id);

    void deleteAllById(Iterable<? extends ID> ids);
}
```

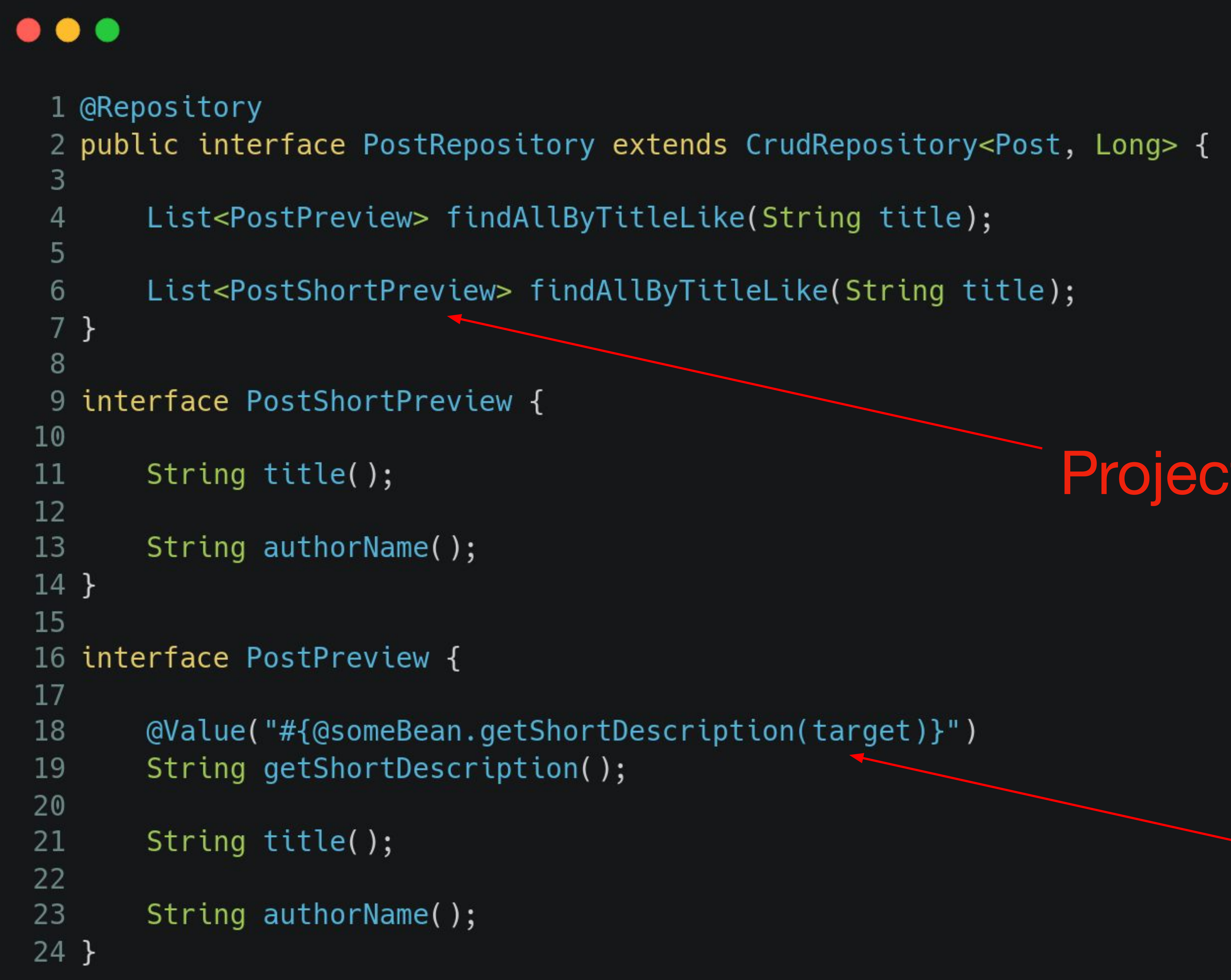


```java
@Repository
public interface PostRepository extends CrudRepository<Post, Long> {

    List<PostPreview> findAllByTitleLike(String title);

    List<PostShortPreview> findAllByTitleLike(String title);
}

interface PostShortPreview {

    String title();

    String authorName();
}

interface PostPreview {

    @Value("#{@someBean.getShortDescription(target)}")
    String getShortDescription();

    String title();

    String authorName();
}
```



```java
@Repository
public interface PostRepository extends CrudRepository<Post, Long> {

    List<PostPreview> findAllByTitleLike(String title);

    List<PostShortPreview> findAllByTitleLike(String title);
}

interface PostShortPreview {

    String title();

    String authorName();
}

interface PostPreview {

    @Value("#{@someBean.getShortDescription(target)}")
    String getShortDescription();

    String title();

    String authorName();
}
```

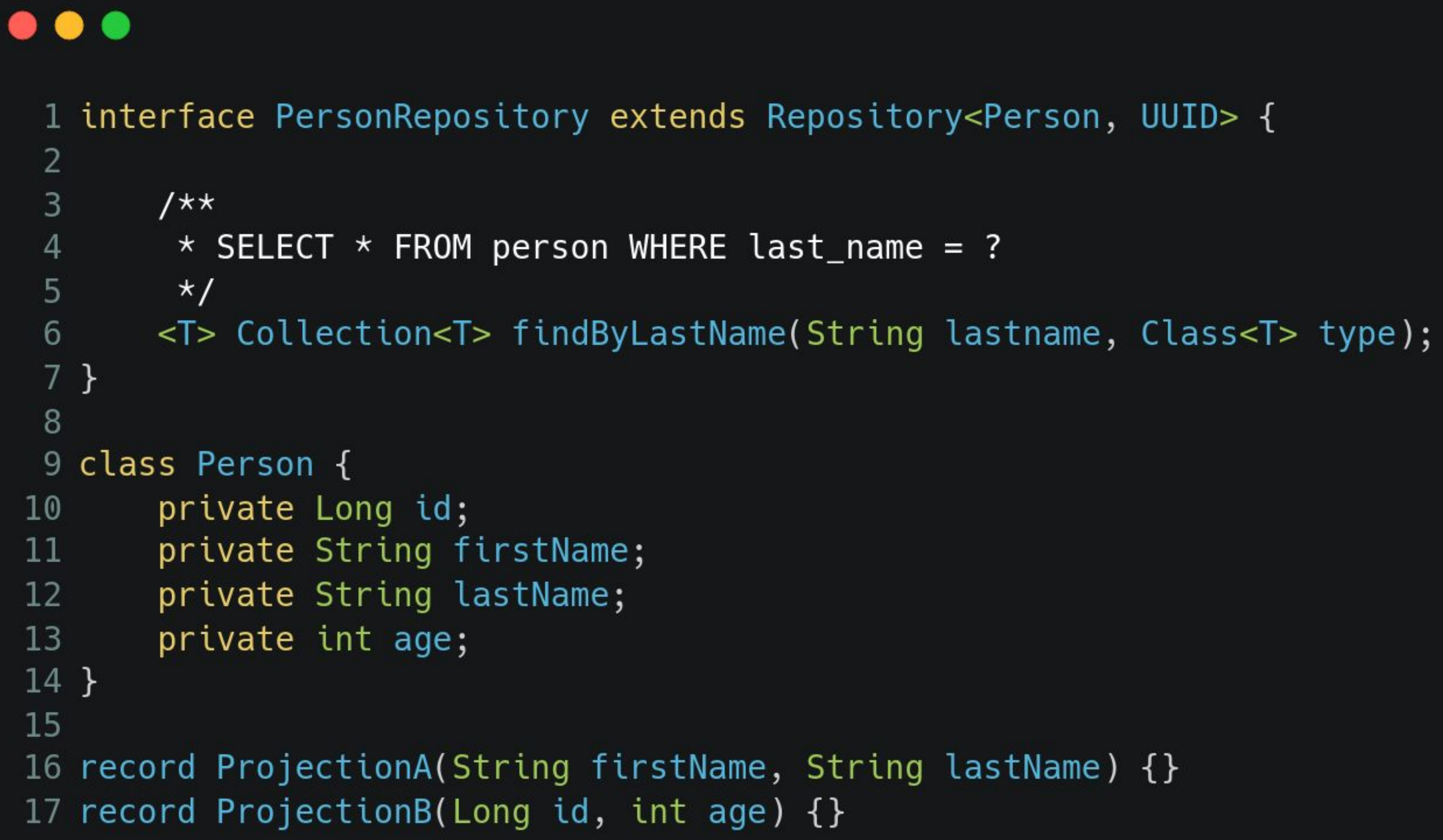
Projections



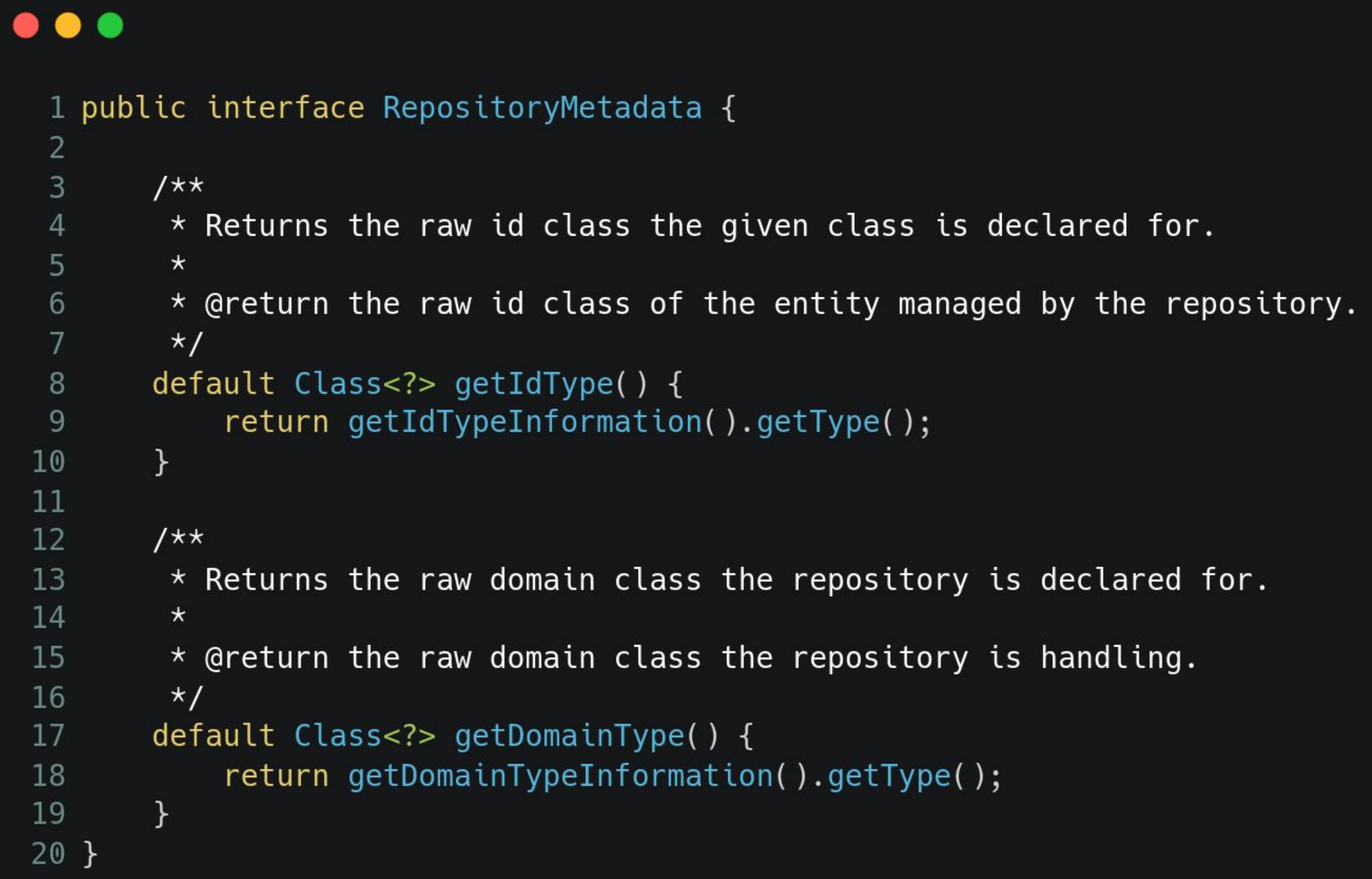
```java
@Repository
public interface PostRepository extends CrudRepository<Post, Long> {

    List<PostPreview> findAllByTitleLike(String title);

    List<PostShortPreview> findAllByTitleLike(String title);
}

interface PostShortPreview {

    String title();

    String authorName();
}

interface PostPreview {

    @Value("#{@someBean.getShortDescription(target)}")
    String getShortDescription();

    String title();

    String authorName();
}
```

Projections

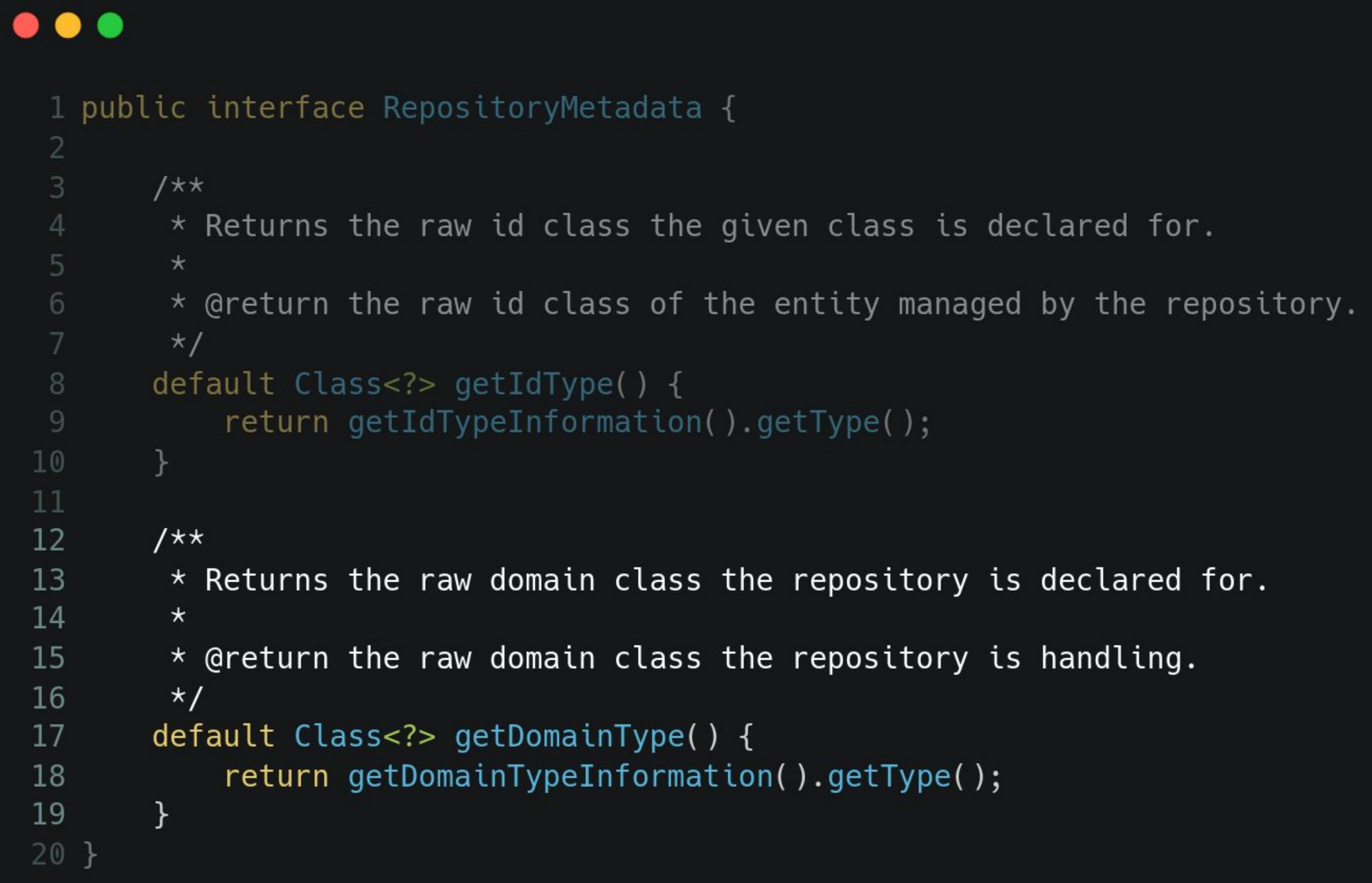
Open Proj.



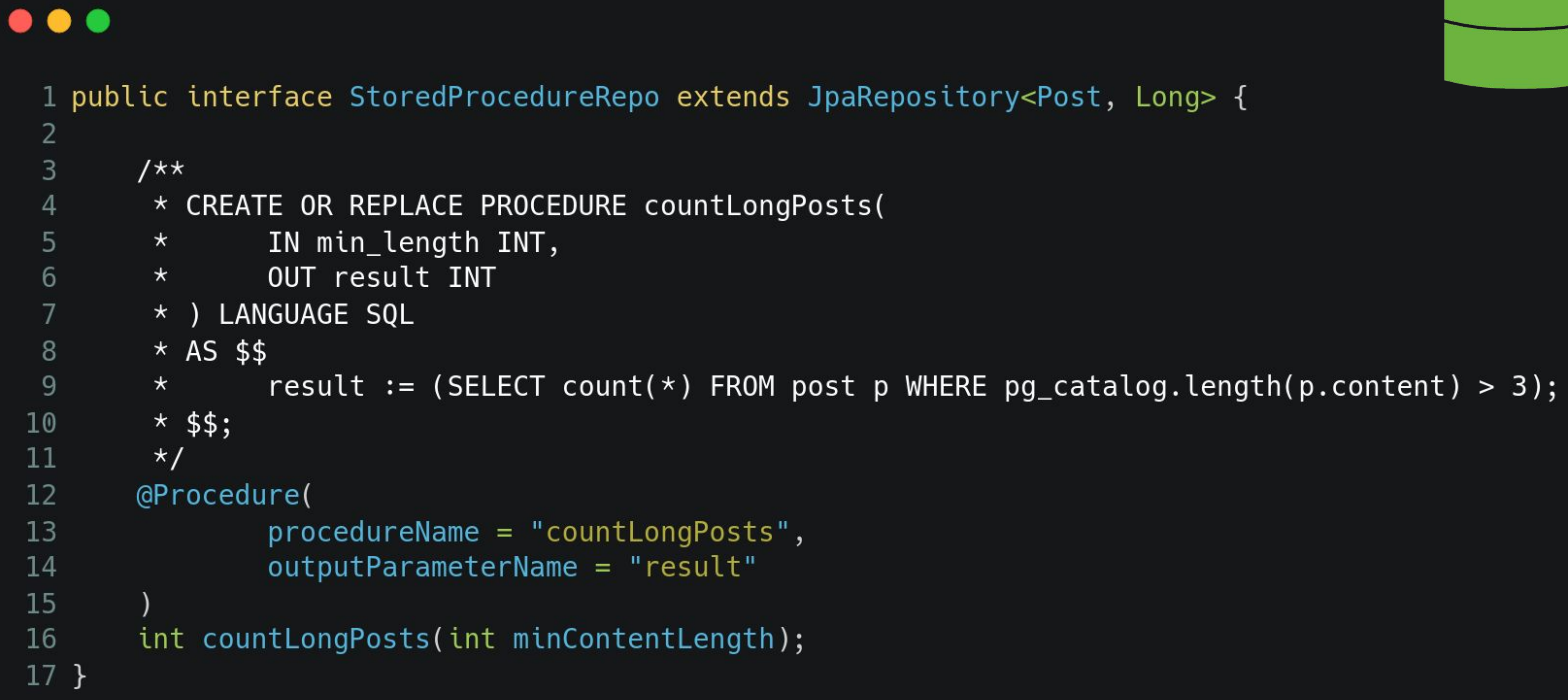
```java
interface PersonRepository extends Repository<Person, UUID> {

    /**
     * SELECT * FROM person WHERE last_name = ?
     */
    <T> Collection<T> findByLastName(String lastname, Class<T> type);
}

class Person {
    private Long id;
    private String firstName;
    private String lastName;
    private int age;
}

record ProjectionA(String firstName, String lastName) {}
record ProjectionB(Long id, int age) {}
```



```java
public interface RepositoryMetadata {

    /**
     * Returns the raw id class the given class is declared for.
     *
     * @return the raw id class of the entity managed by the repository.
     */
    default Class<?> getIdType() {
        return getIdTypeInformation().getType();
    }

    /**
     * Returns the raw domain class the repository is declared for.
     *
     * @return the raw domain class the repository is handling.
     */
    default Class<?> getDomainType() {
        return getDomainTypeInformation().getType();
    }
}
```



```java
public interface RepositoryMetadata {

    /**
     * Returns the raw id class the given class is declared for.
     *
     * @return the raw id class of the entity managed by the repository.
     */
    default Class<?> getIdType() {
        return getIdTypeInformation().getType();
    }

    /**
     * Returns the raw domain class the repository is declared for.
     *
     * @return the raw domain class the repository is handling.
     */
    default Class<?> getDomainType() {
        return getDomainTypeInformation().getType();
    }
}
```



```java
public interface StoredProcedureRepo extends JpaRepository<Post, Long> {

    /**
     * CREATE OR REPLACE PROCEDURE countLongPosts(
     *      IN min_length INT,
     *      OUT result INT
     * ) LANGUAGE SQL
     * AS $$
     *      result := (SELECT count(*) FROM post p WHERE pg_catalog.length(p.content) > 3);
     * $$;
     */
    @Procedure(
            procedureName = "countLongPosts",
            outputParameterName = "result"
    )
    int countLongPosts(int minContentLength);
}
```

Repositories Design Principles

1. Jakarta Data does not require a repository to be bound to a single entity type.

2. Jakarta Data assumes, that it can resolve the entity type for given operation by using either:
   a. returned entity type
   b. or parameter type

3. Spring Data heavily relies on the fact that Repository is bound to a single Aggregate Root

4. Spring Data does not require a given method to be anyhow bound to an entity. It might be a simple storage function call.

Conclusions

1. Spring Data currently incapable to implement Jakarta Data in any way.

2. On the other hand, RedHat projects, such as Panache work well with Jakarta Data

3. As a consequence, the industry would have two competing Repository implementations - Jakarta Data with Quarkus infrastructure, and Spring Data with Spring framework

Thank you!

@MIPO256