Reflections About Persistency

A Very Technical Journey

What's here to learn?

- Criteria for choosing a persistence technology
- Reasons pro and contra technology abstraction
- Good judgement for using transaction semantics
- Good judgement for using caching

What's here to see?

- Various APIs
  - JPA, Criteria API,
  - JOOQ, QueryDSL,
  - Spring Data
- We'll discuss a case study
- and that's where the fun starts…

Our Technology Stack

- Java SDK 8
- Spring 4.0 with Spring-Data and Hibernate JPA 2.1
- MySQL
- MongoDB
- EhCache

If you'd have to program this...

- Swiss Trusted Poker Site
- Semi-private internet poker for closed groups
- Very high stakes, still thousands of players worldwide
- Extremely reliable AND blazing fast
- All bets must be perfectly consistent (it's loads of money)

Reflecting on the problem

- What would you choose?
- What technology?
  - SQL or NoSQL?
- What deployment model?
  - Master-Slave, other HA options
- What transaction semantics?
  - CAP or ACID?

Varying Requirements

- Different Use Cases may require different runtime characteristics
- Not always, one size fits it all
- Modern architectures employ different technologies
- Keep your architectural mind open
- Mostly ok to consider JPA, the standard, first

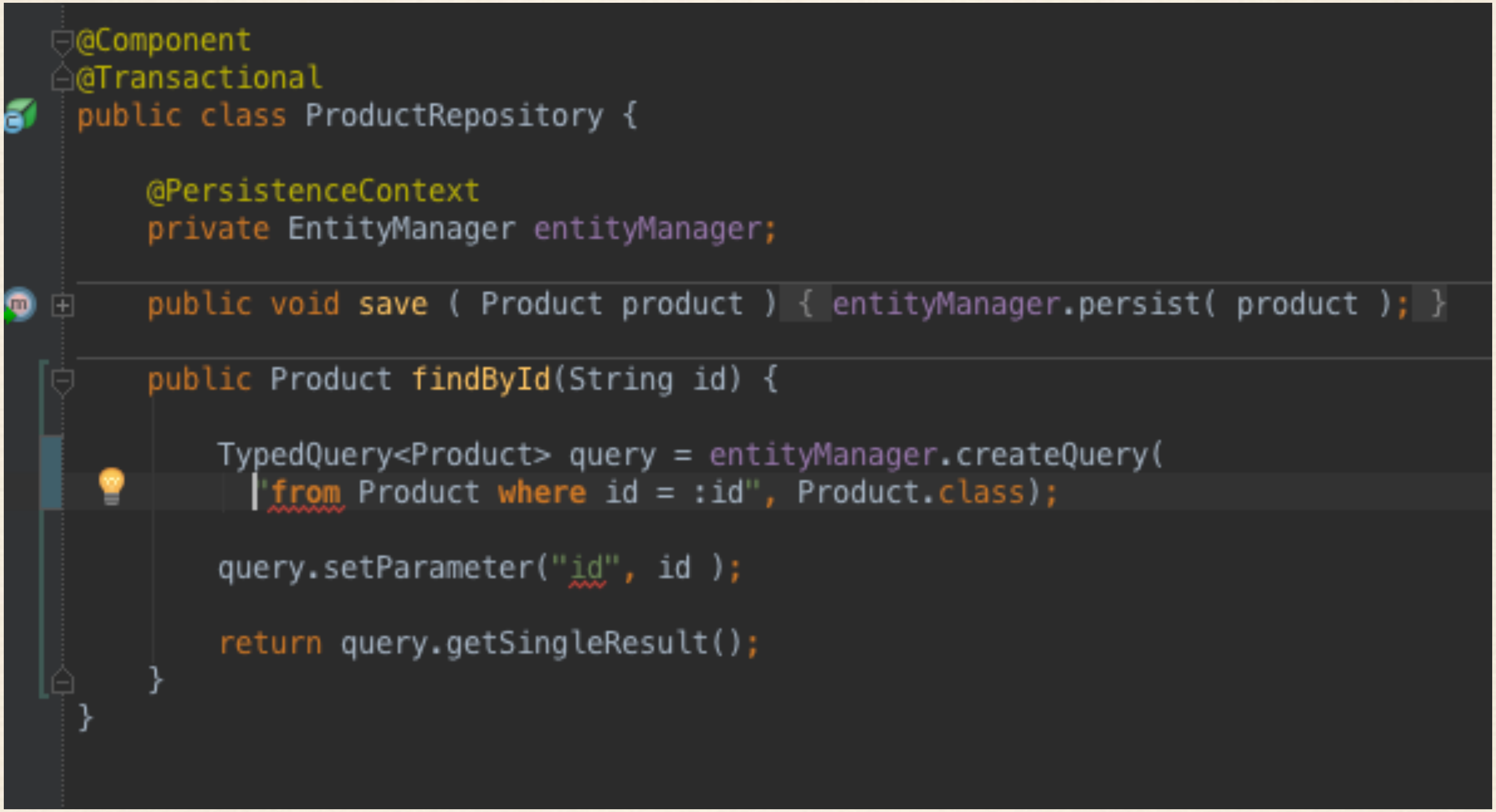
JPA is the Standard

- Java Persistence API 2.1 now the JEE Standard
- inspired by Toplink and Hibernate
- Replaced the EJB 2.1 Entity Beans long ago.
- Now, what is JPA - in a nutshell?

JPA in a Nutshell

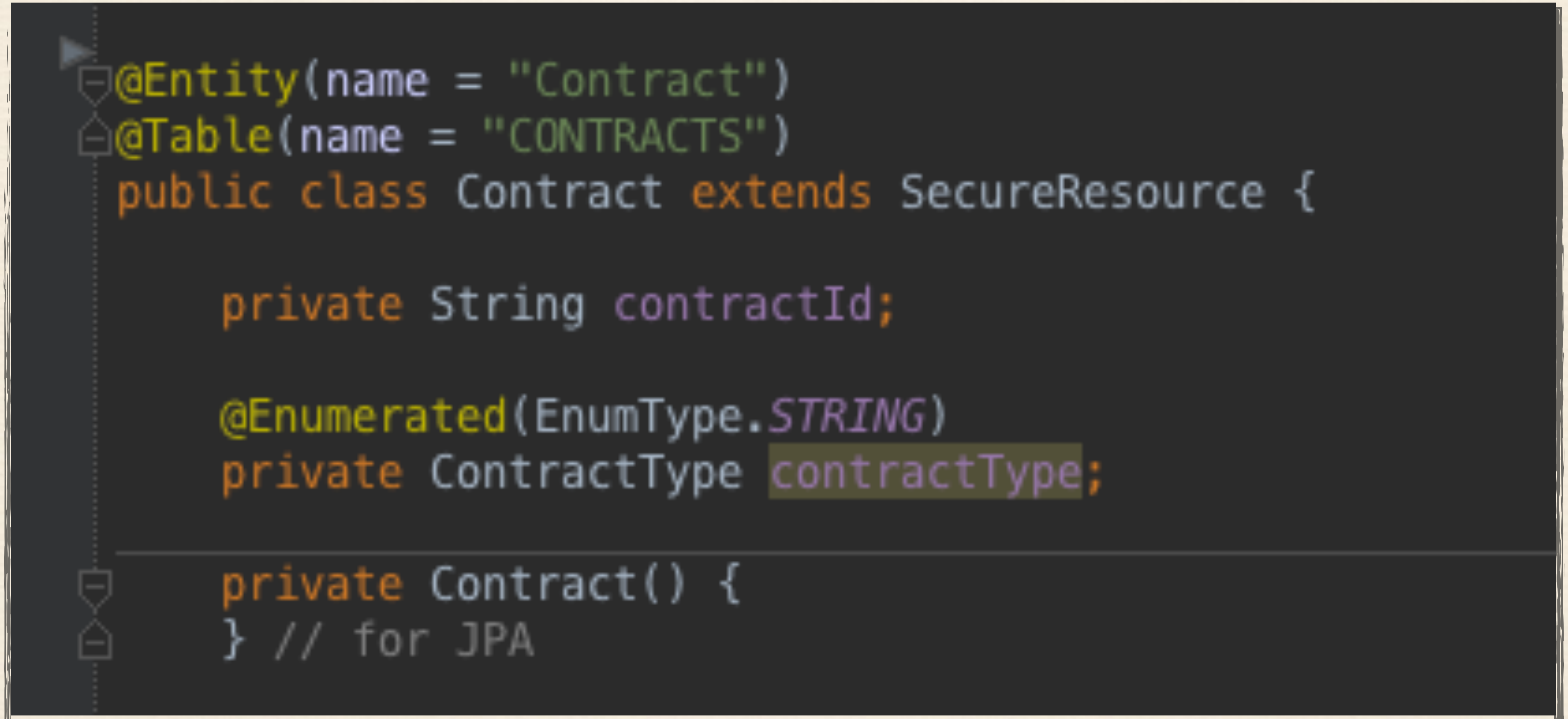
- ORM: Object-relational mapping
- Map Classes to Tables
- Map primitive fields to columns
- Map references to foreign keys
- Map inheritance in three different ways
- Map JPQL to SQL

API: the EntityManager



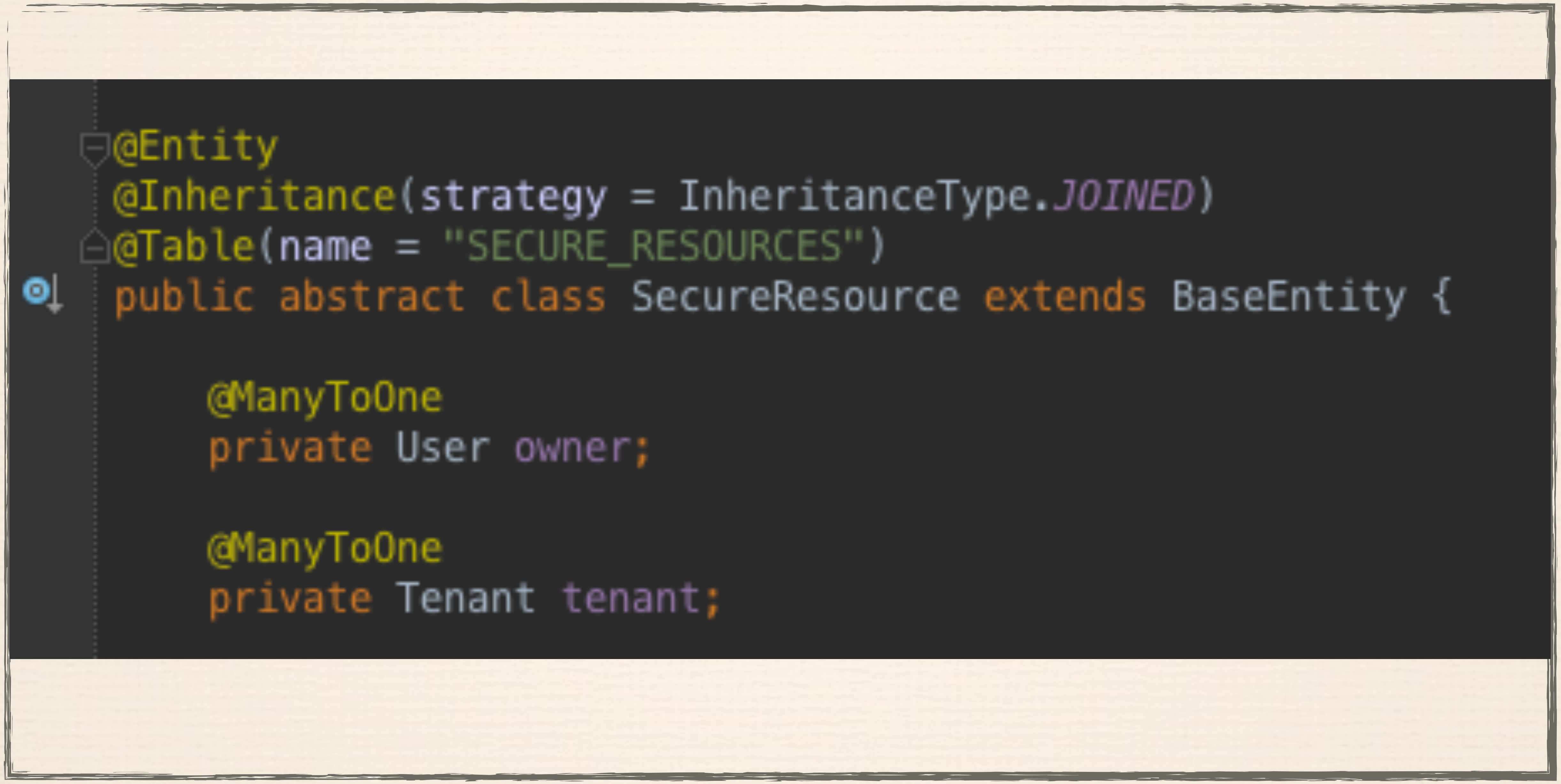
```
@Component
@Transactional
public class ProductRepository {

    @PersistenceContext
    private EntityManager entityManager;

    public void save ( Product product ) { entityManager.persist( product ); }

    public Product findById(String id) {

        TypedQuery<Product> query = entityManager.createQuery(
            "from Product where id = :id", Product.class);

        query.setParameter("id", id );

        return query.getSingleResult();
    }
}
```

Mapping fields



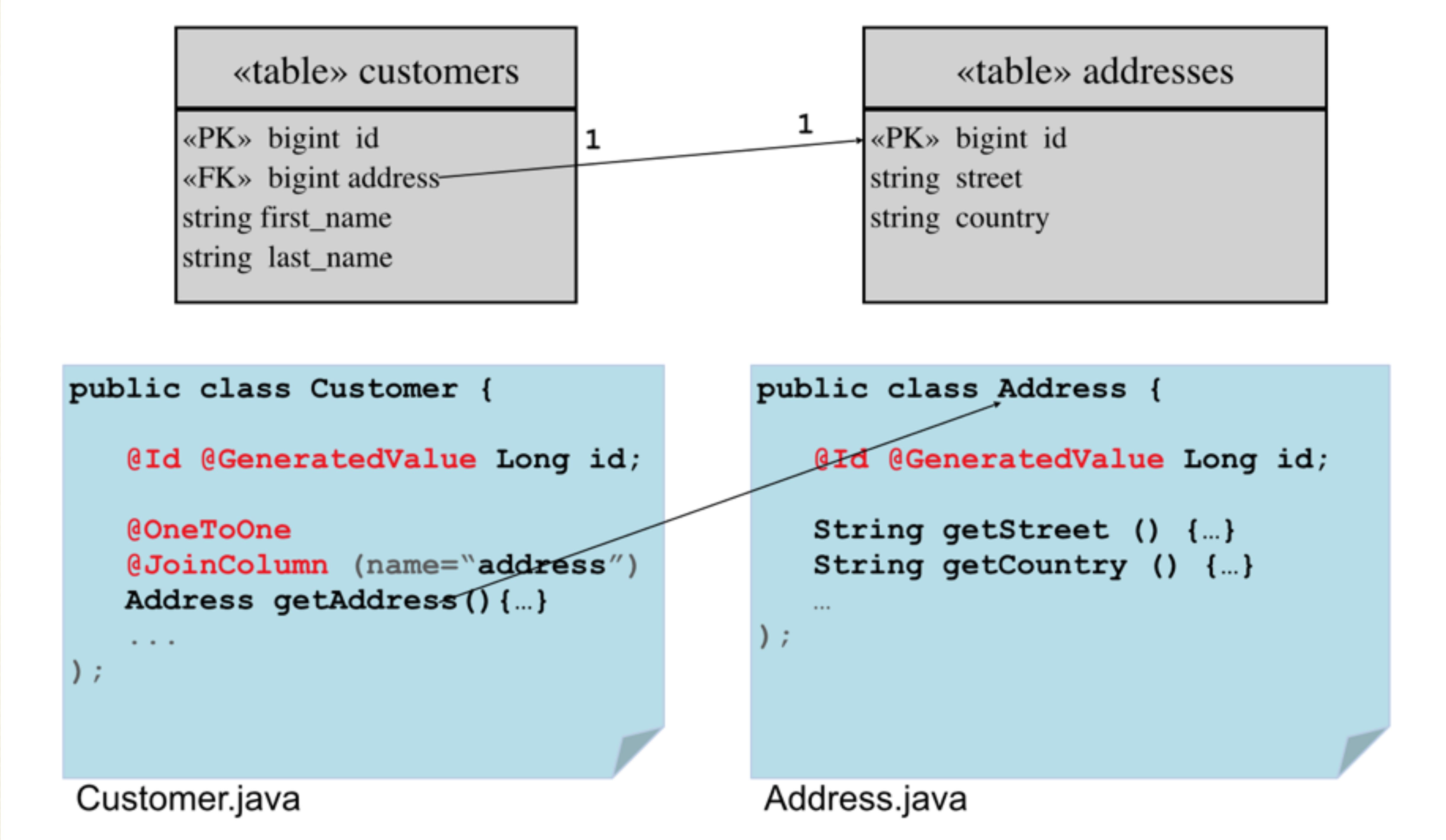
```java
@Entity(name = "Contract")
@Table(name = "CONTRACTS")
public class Contract extends SecureResource {

    private String contractId;

    @Enumerated(EnumType.STRING)
    private ContractType contractType;

    private Contract() {
    } // for JPA
```

Mapping references



```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@Table(name = "SECURE_RESOURCES")
public abstract class SecureResource extends BaseEntity {

    @ManyToOne
    private User owner;

    @ManyToOne
    private Tenant tenant;
```

The big picture



«table» customers

«PK» bigint id
«FK» bigint address
string first_name
string last_name

«table» addresses

«PK» bigint id
string street
string country

```
public class Customer {

   @Id @GeneratedValue Long id;

   @OneToOne
   @JoinColumn (name=“address”)
   Address getAddress(){…}
   ...
);
```

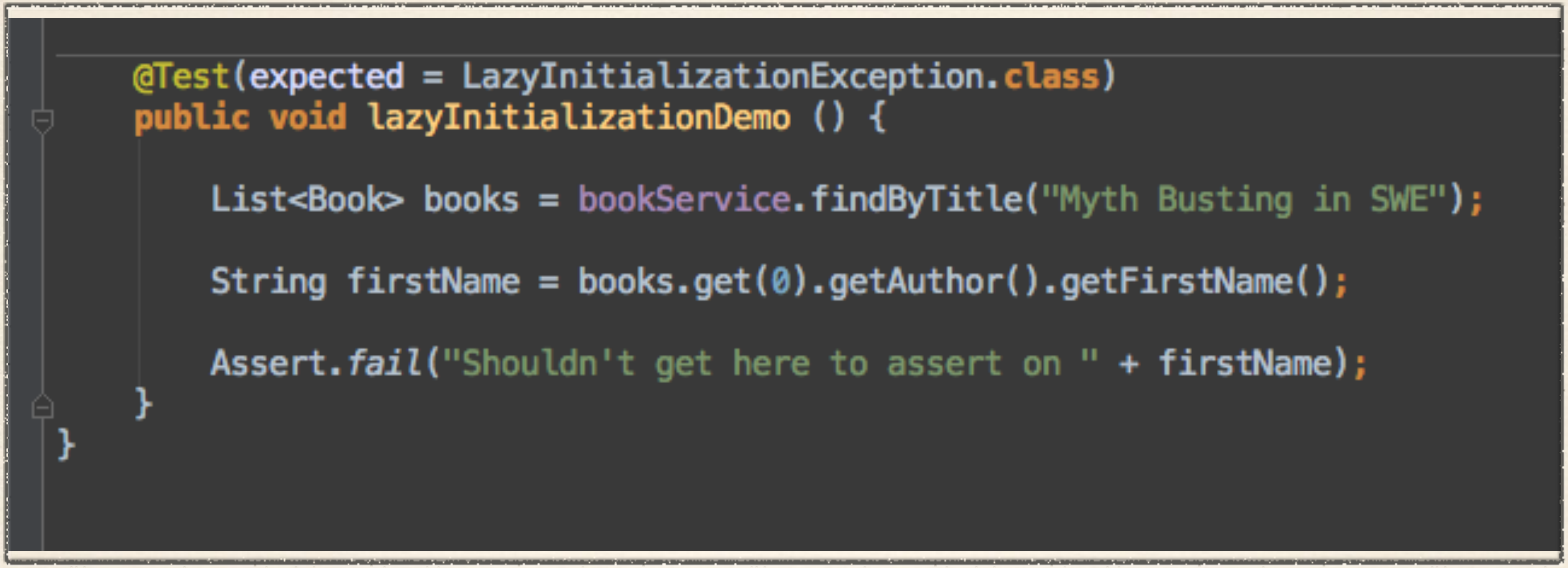
Customer.java

```
public class Address {

   @Id @GeneratedValue Long id;

   String getStreet () {…}
   String getCountry () {…}
   …
);
```

Address.java

Lazy vs Eager

- Not the entire graph will be read from DB
- References may be lazily loaded.
- So may be BLOBs
- Only accessing lazy fields will retrieve referenced entities
- LazyInitializationException if outside of transactional context.



```
    @Test(expected = LazyInitializationException.class)
    public void lazyInitializationDemo () {

        List<Book> books = bookService.findByTitle("Myth Busting in SWE");

        String firstName = books.get(0).getAuthor().getFirstName();

        Assert.fail("Shouldn't get here to assert on " + firstName);
    }
}
```

Gavin King's nightmare

Lazy-loaded references can't be accessed outside of the transaction

ACID Transactions

- Atomic: all or nothing to succeed
- Consistent: obey constraints
- Isolated: from each other
- Durable: on the disk
-

Transactions with Spring

- JPA can delegate Tx-Handling to the container
- Transactions are handled at the method entry and exit through AOP
- Exceptions may cause roll-back
- @Transactional Annotation defines attributes

Rollback Strategies

- Runtime Exceptions cause a roll-back
- Checked Exceptions don't
- Exceptions to the rule by
  - `@Transactional (rollbackOn=Class[])`
  - `@Transactional (dontRollbackOn=Class[])`

Propagation

- With Spring very similar to JEE
- REQUIRED: create tx if none available yet
- REQUIRES_NEW: create tx, suspend existing
- MANDATORY: fail if none available
- NOT_SUPPORTED: suspend, if existing
- SUPPORTS: execute in whatever is available
- NEVER: fail if tx available

How does a method know, whether there's a transaction?

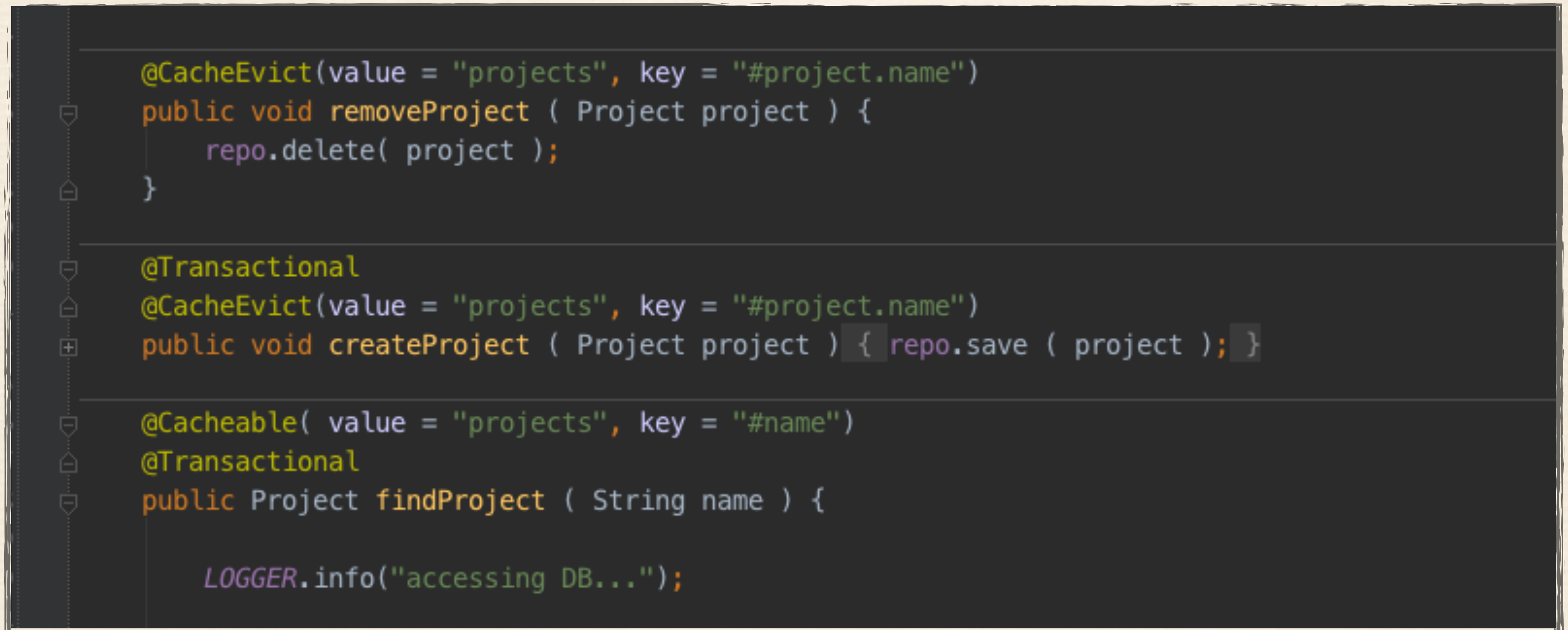
Saving Audit Logs

- Outer Method: REQUIRED
- Inner Method: REQUIRES_NEW
- Outer Method:
  - When exception occurs:
  - outer context rolls back
  - inner context still committed.

Caching

- Client or server-side caching?
- Single point of access? Otherwise outdated?
- Evict entry when writing
- Evict to avoid OutOfMemoryException
- LRU (last recently used)
- Life time expiration
- May be difficult to test

The Caching Aspect



```
@CacheEvict(value = "projects", key = "#project.name")
public void removeProject ( Project project ) {
    repo.delete( project );
}

@Transactional
@CacheEvict(value = "projects", key = "#project.name")
public void createProject ( Project project ) { repo.save ( project ); }

@Cacheable( value = "projects", key = "#name")
@Transactional
public Project findProject ( String name ) {

    LOGGER.info("accessing DB...");
```

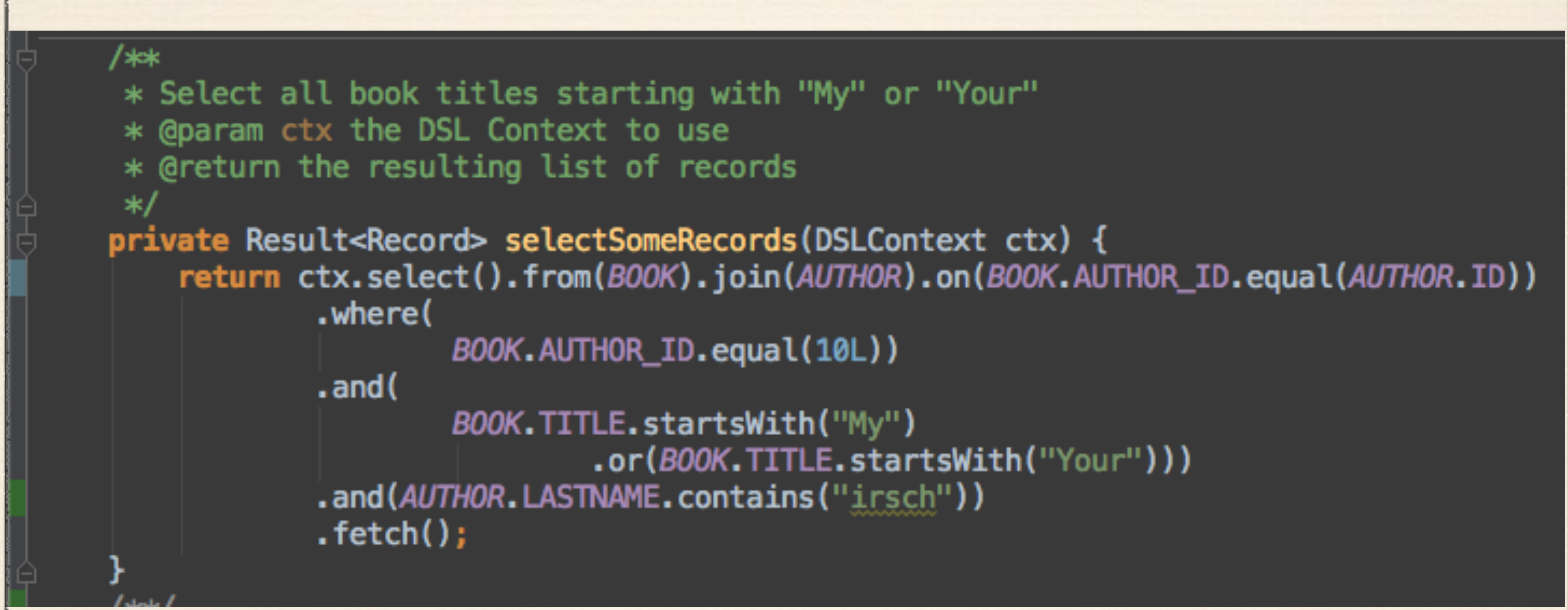
- run CacheTests.java and see the result

To cache or not to cache?

- Keep the results of expensive operations in memory, but where exactly?
- Consider eviction, single point of access
- Consider security
- Consider testability
- JCache in Java 9 (hopefully) and Spring 4.1

RDBMS: different APIs

- JPA with TypedQuery from Strings
- JPA with type-safe Criteria API
- QueryDSL with generated code
- JOOQ with generated Code
- See demo test classes in org.smurve.hsr2014.apis



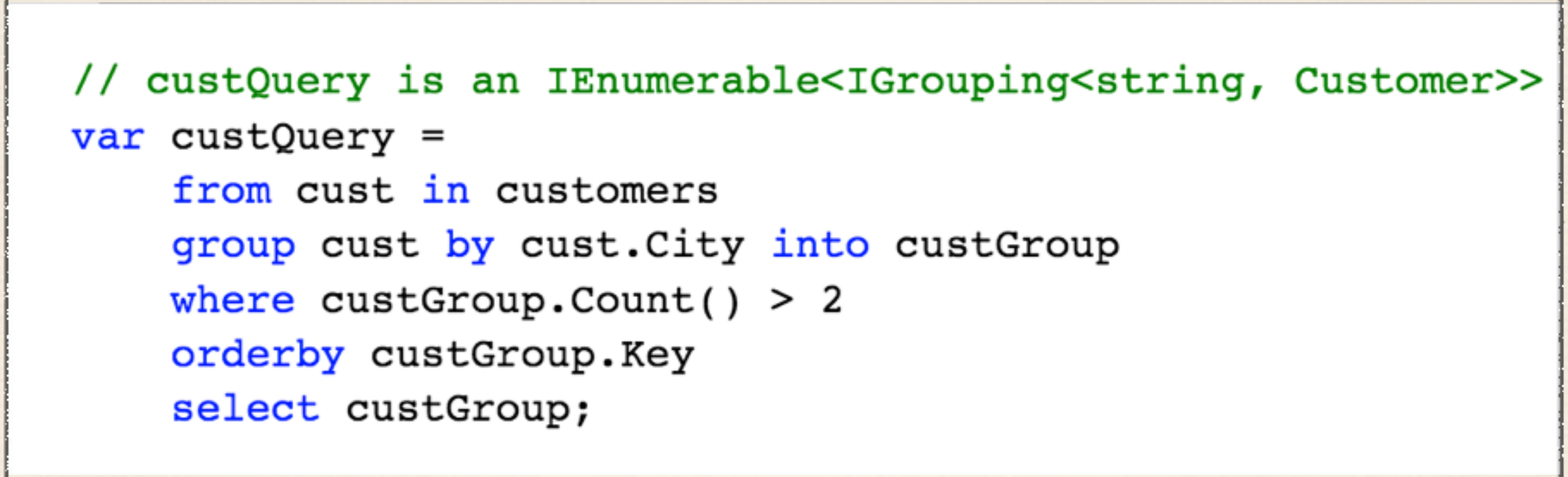
```
/**
 * Select all book titles starting with "My" or "Your"
 * @param ctx the DSL Context to use
 * @return the resulting list of records
 */
private Result<Record> selectSomeRecords(DSLContext ctx) {
    return ctx.select().from(BOOK).join(AUTHOR).on(BOOK.AUTHOR_ID.equal(AUTHOR.ID))
            .where(
                    BOOK.AUTHOR_ID.equal(10L))
            .and(
                    BOOK.TITLE.startsWith("My")
                            .or(BOOK.TITLE.startsWith("Your")))
            .and(AUTHOR.LASTNAME.contains("irsch"))
            .fetch();
}
```

JOOQ is QOOL

Use for complex reporting queries

Low level APIs

- Low-level APIs typically need be integrated to have transactions managed by a larger framework
- In the exercises: JooqSpringDemo



```
// custQuery is an IEnumerable<IGrouping<string, Customer>>
var custQuery =
    from cust in customers
    group cust by cust.City into custGroup
    where custGroup.Count() > 2
    orderby custGroup.Key
    select custGroup;
```

OK, ok, LINQ is cooler

Choosing Persistency

…is not so easy!

RDBMS/ORM/JPA

- Very well established technology
- Transactions: ACID
- JPA: Standardised abstraction from underlying product
- A relational System is good for things that are significantly related and form small graphs.

Non-Trivial Tx

- Exceptions and Rollbacks
- Propagation Semantics for Audit Requirements
- One example in the exercises (please find out how it works):

```
MultiTenantSecurityTest.
  test_audit_records_with_Exceptions()
```

ORM: The Abstraction Promise

- Hide the details from the developer
- Everything is OO
- loose coupling
- make DB replaceable
- and what's the result?

ORM Gone Bad: The Abstraction Problem

- Start easily without thinking, think later and thus a lot harder
- Many non-trivial JPA projects now recognise Hibernate behind the abstraction
- Many non-trivial Hibernate projects now recognise SQL behind the abstraction
- And by the way: Most JSF projects now recognise JS and HTTP (REST)

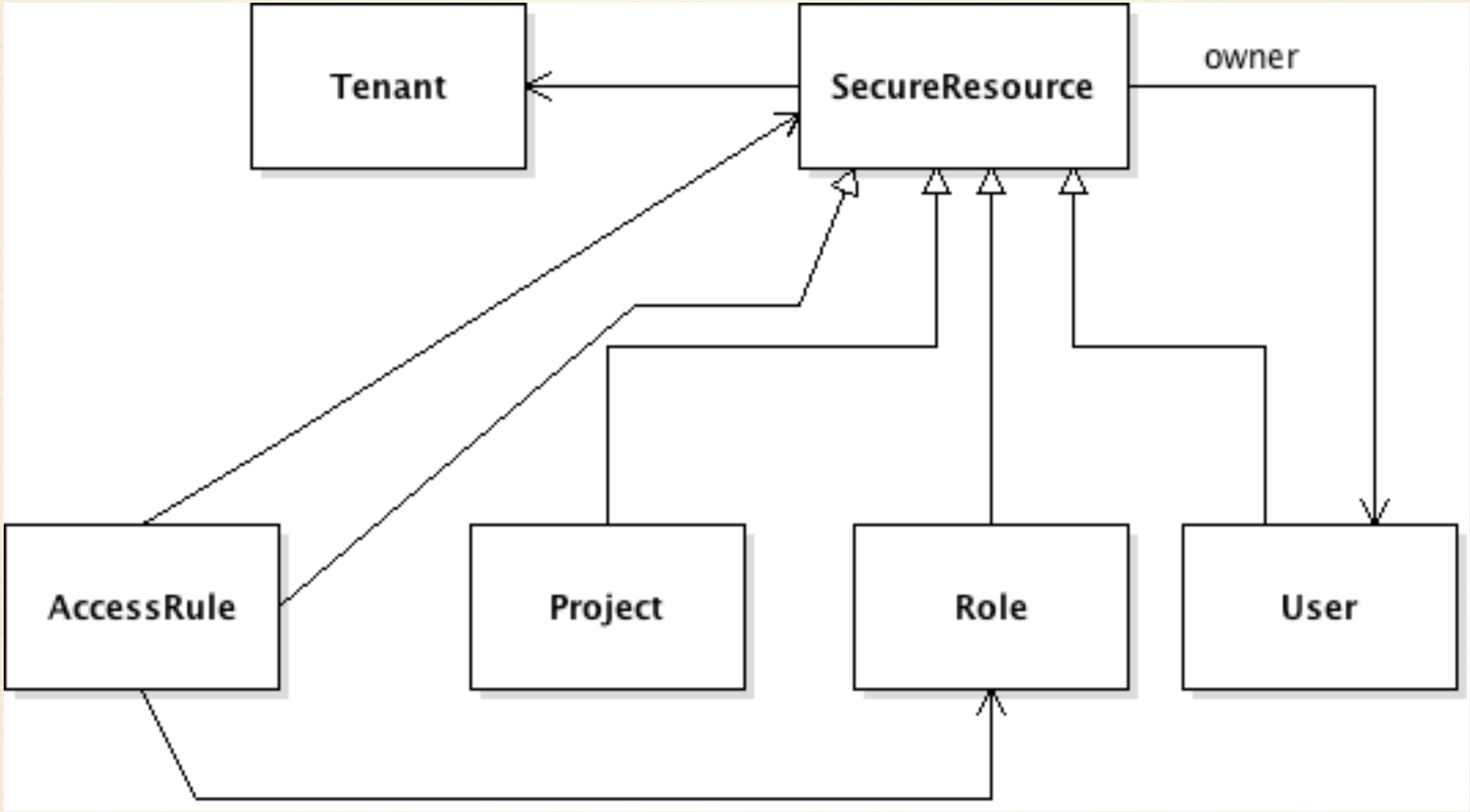
Non-trivial ORM

- Do I need ORM mapping, query abstraction?
- How smart do I need ORM to be?
- "There's already one perfect DSL to access an RDBMS, it's called SQL and it's truly well-established - so why invent another?"
- Some current projects avoid ORM

ORM Gone Bad: An example

- A SecureResource "knows" its owner and tenant
- A Project, a Role, a User and an AccessRule are all SecureResources
- An AccessRule associates Roles with SecureResources

Simple domain model



In the exercises

- Hibernate produces massive SQL Statements
- Not all Inheritance strategies work
-

If the price is high,...

- Do I need referential Integrity via DB?
- Do I need to map inheritance?
- Is the domain model too tightly coupled?
- Do I need ACID Transactions across entities and multiple statements?
- Cut your domain model into reasonable parts

Careful Design your data model

- The domain model MUST consider the implementation details.
- Decouple aspects from the domain model
- Favour reference over inheritance
- consider "Embedded" mapping to avoid inheritance

More Data Model Design Aspects

- Do we really need normalisation? Disks are cheap.
- Identify clusters in your domain model, clusters may then become documents
- Loose coupling may improve performance
- Lazy-load hell can be avoided.

New kid on the block: CQRS

- Command-Query-Responsibility Segregation
- Separate reading to and writing from storage
- Optimise reading and writing independently
- Consider for very large and fast data requirements
- see: http://martinfowler.com/bliki/CQRS.html

Not only SQL

- What else?
- MongoDB, Redis, Cassandra, Hazelcast, Neo4J,…
- No more ACID, but CAP
- Remember: Architecture decisions are trade-off decisions - you may not get it all
- Cost of failure vs. cost of prevention

NoSQL? No, SQL!

- Consider pros and cons, features and acceptance
- What exactly is it that you need?
- Why not both?

Selection Criteria

- What part of ACID/CAP do I need? Is it provided?
- Which performance requirements do I expect mid-term? Size, latency in R/W?
- Is the product well-established and well-supported?
- Can the operations dept support the runtime characteristics?

Design to cost

- Why use transactions?
- What's the price of an "accident"?
- What's the likelihood of an "accident" to happen
- What's the price of preventing the accident?

One Alternative: MongoDB

- High performance db journaling
- Flexible access patterns
- Document-style database
- Secondary indexes, huge tables, simple queries, extremely fast and eventually persistent.

Approaching Swiss Trusted Poker

- No typical session state, thus RESTful.
- Multiple players share state: Tables and Games
  - Consider Actor-Based implementation or alike.
- Infrastructure like a poker table is held in MySQL
- Game state is held in Cache representing the game
- All messages are written to a persistent journal (MongoDB)
- The messages can recreate the state in case of failure.

In The Exercises: Exploring Spring Data MongoDB

- Needs mongod installed on localhost
- Cool Abstraction
- Hiding the Persistence layer behind the repository interface.
- class: MongoPersonRepoTests
- Cool: PersonRepo is technology-agnostic

In the Exercises: Exploring ORM Monster statements

- Some domain models are not easy to map
- Class InheritanceTests
- Creates massive statements
- Compare to class InheritanceTests2
- Domain model simplification solves the problem.

In the Exercises: Exploring Caching Aspects

- Caching is a typical cross-cutting concern
- Should be implemented as an aspect
- Class: CacheTests
- Pay attention to the eviction strategies
- Hint: search for and uncomment “show_sql”

Tonight's main course:

- Write a class that takes any public Author findBy<Field> (String name) interface and returns a JOOQ-based implementation of it.