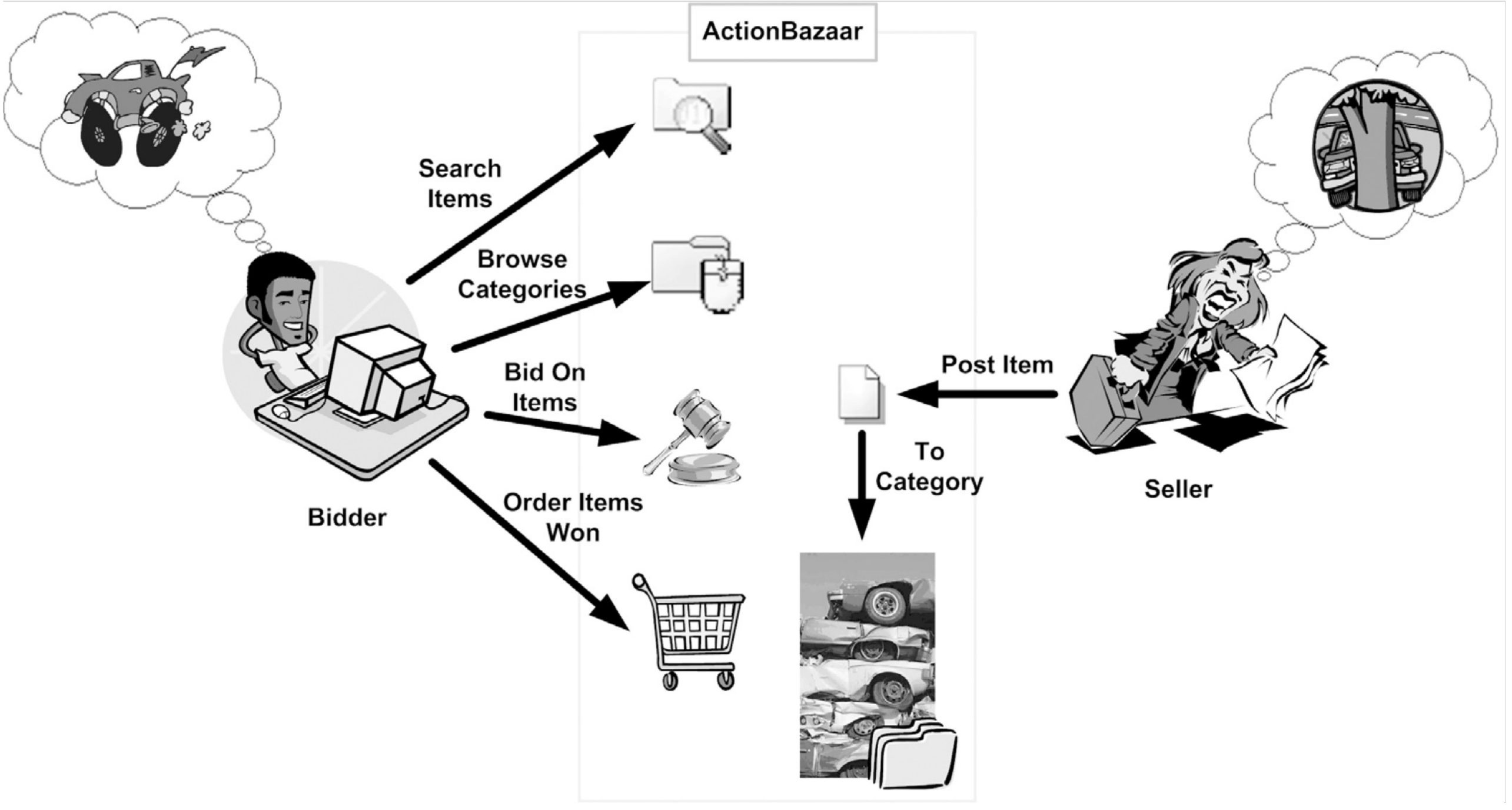
ORM & Persistence Basics

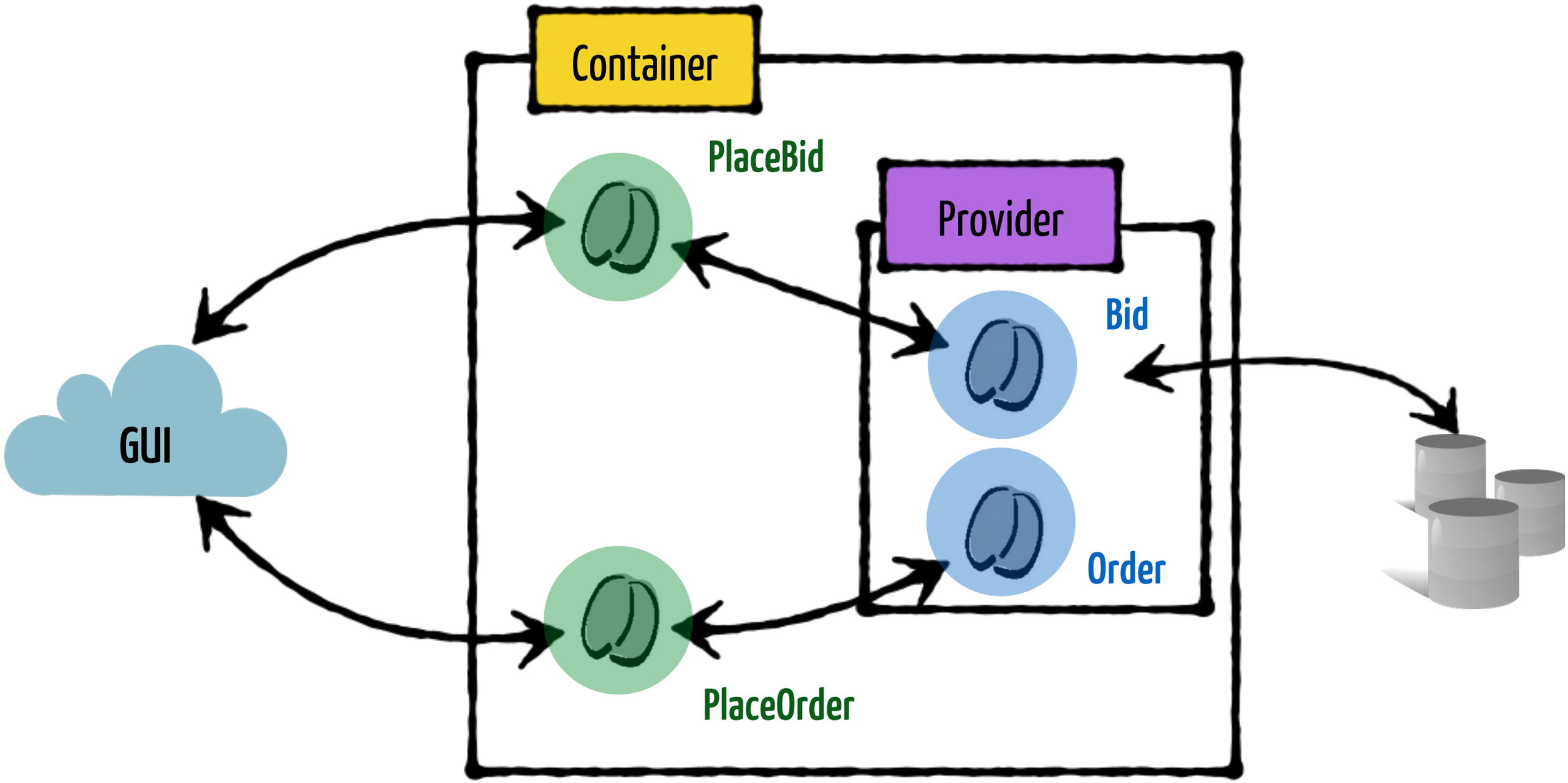
Philippe Collet
(with some slides from Sébastien Mosser)

pictures: sxc.hu, pixabay

Object-Relational Mapping 101


ActionBazaar
Search Items
Browse Categories
Bid On Items
Order Items Won
Bidder
Post Item
To Category
Seller

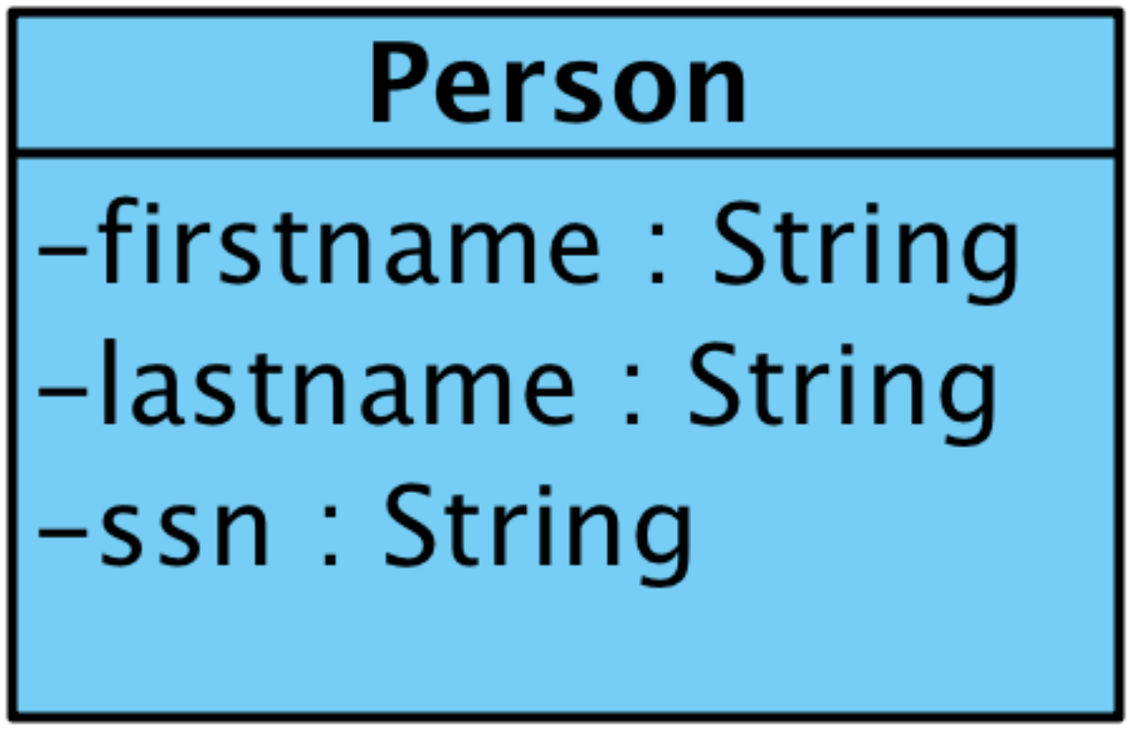
Components & Entities



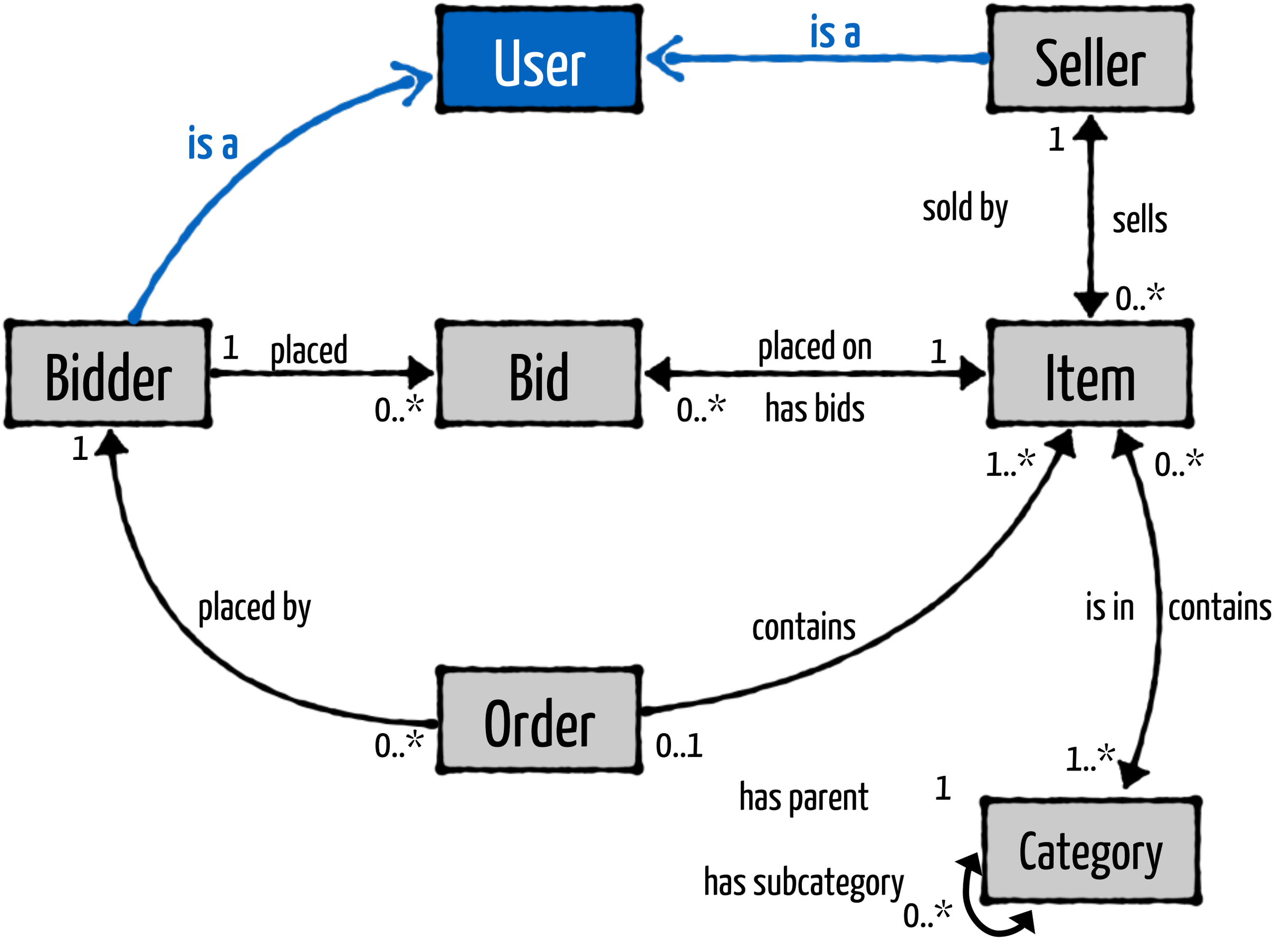
Impedance Mismatch

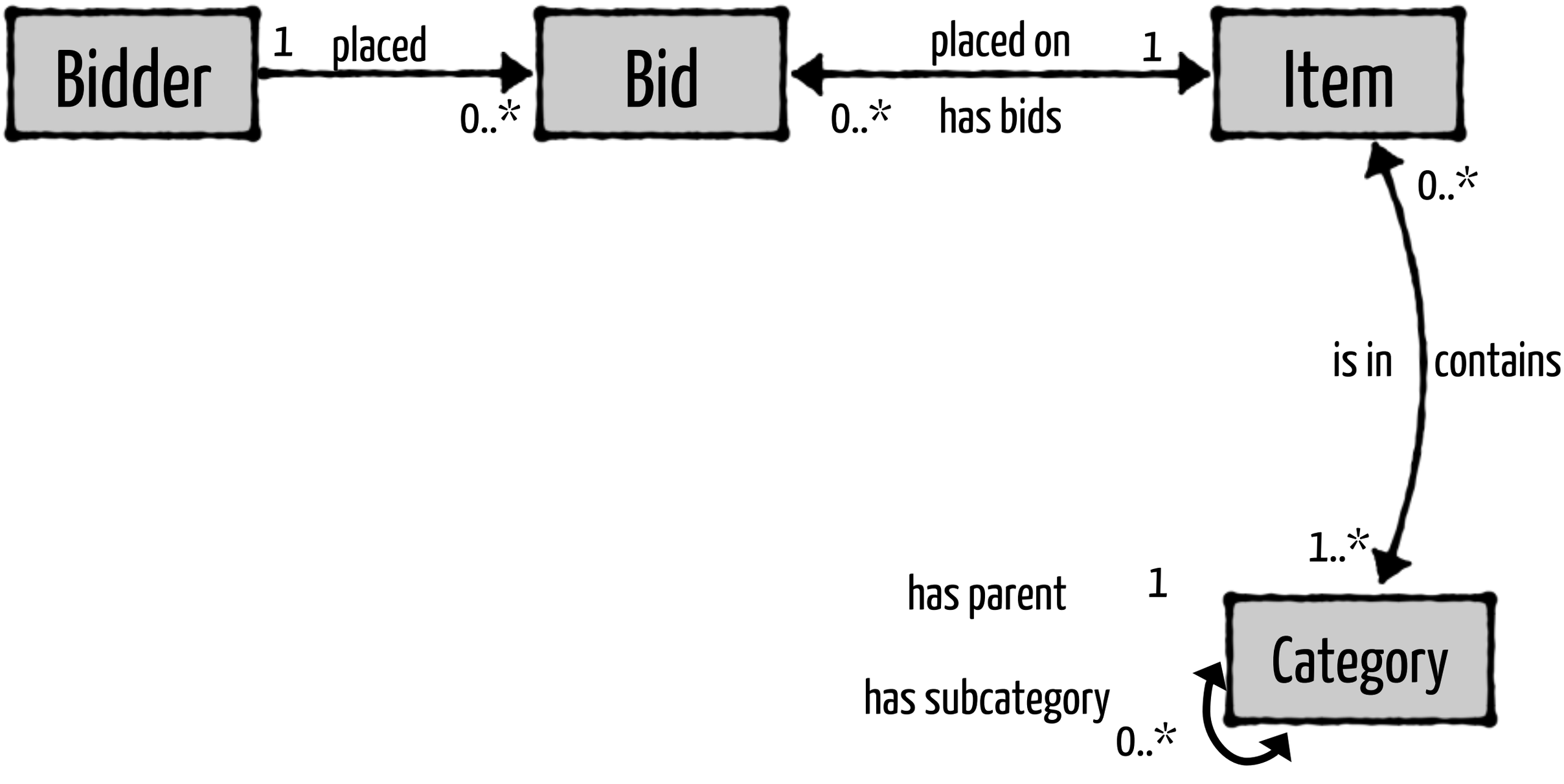
Object-Oriented	Relational
Classes	Relation (table)
Object	Tuple (row)
Attribute	Attribute (column)
Identity	Primary Key
Reference	Foreign Key
Inheritance	N/A
Methods	~ Stored Procedure

Example of Domain Model



first_name	last_name	ssn
Alex	CEPTION	16118325358
...		


User
is a
Seller
is a
1
sold by
sells
0..*
Bidder
1
placed
Bid
placed on
1
Item
0..*
has bids
0..*
1
1..*
0..*
placed by
contains
is in
contains
0..*
Order
0..1
1..*
has parent
1
Category
has subcategory
0..*


Bidder
1
placed
0..*
Bid
placed on
0..*
has bids
1
Item
0..*
is in
contains
1..*
has parent
1
has subcategory
0..*
Category

Category

```
@Entity
public class Category {

   public Category() { … }

   protected String name;

   public String getName() {
      return this.name;
   }

   public void setName(String n) {
      this.name = n.toUpperCase();
   }
}
```

property-based access

(JPA= Java Persistence API)

JPA Entities need more than simple annotations:

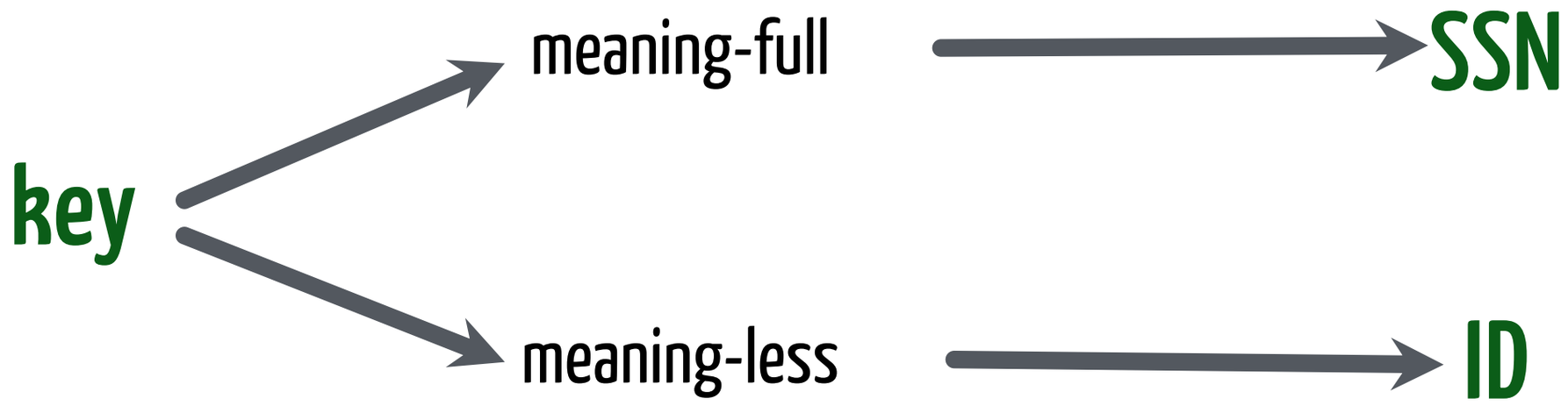
- An empty constructor
- A proper equals method that relies on business elements
- A proper hashCode method to support objects identification in caches

Problem: Domain Object Identity

```
Person p1 = new Person("X","Y","DDMMYYXXXXX")
Person p2 = new Person("X","Y","DDMMYYXXXXX")
```

How to support persons **uniqueness**?

Candidates for **Key** role



- Meaning-Full key:
  - **A posteriori uniqueness check**
- Meaning-Less key:
  - **Auto-generated number**

Necessary condition:

- **Key must be unique**

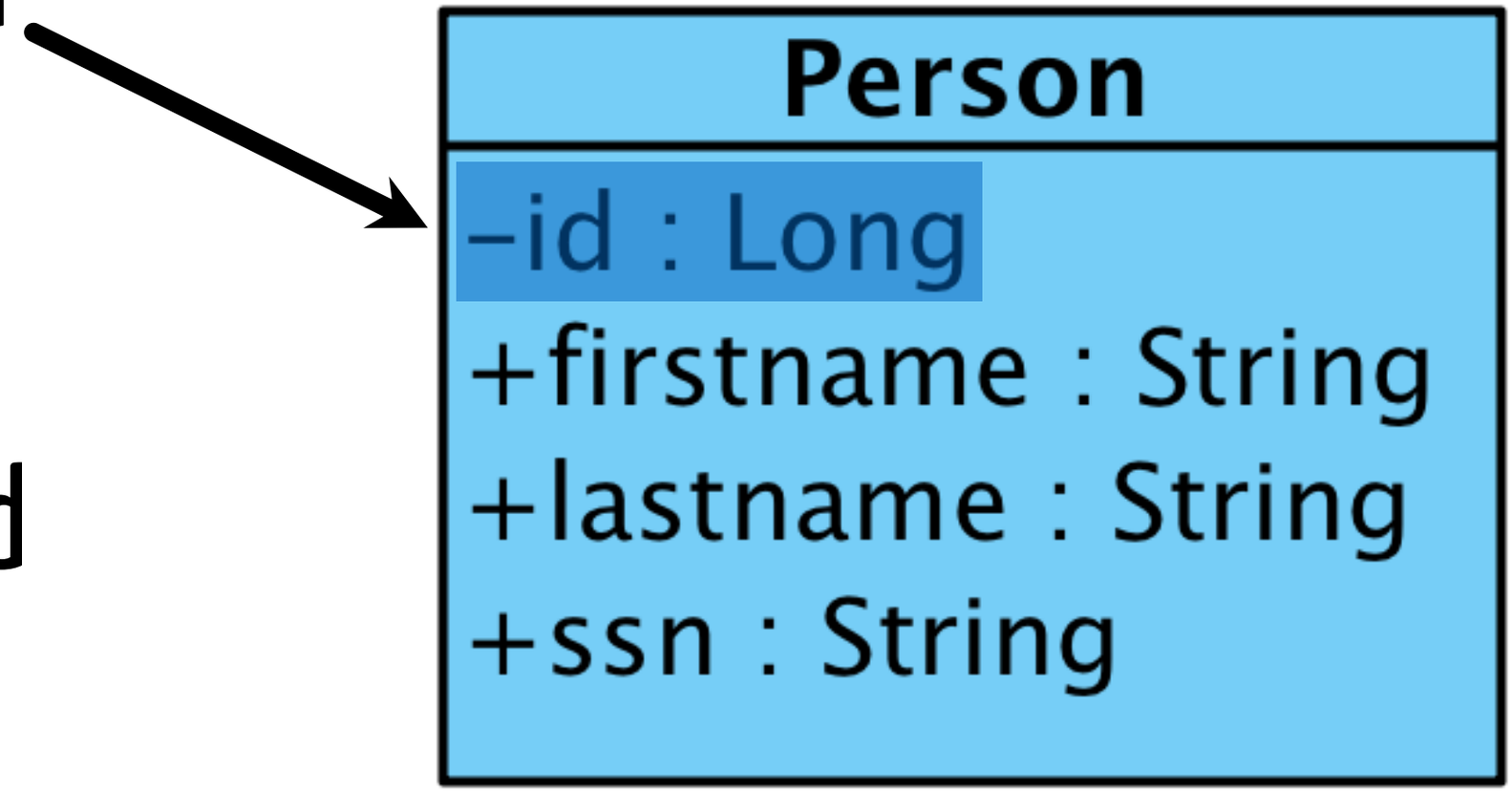
- Nice-to-have condition:
  - **Key should be immutable**
- **Compound** versus **Simple**

Pattern: Identity Field

Not visible at
the business level

auto-generated

GUID DIY



Simple Primary Key: @Id

```
@Entity
public class Category {
   // …

   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;

}
```

Identifiers must define an "equals" method

Equality Relation definition

- equals is reflexive
- equals is symmetric
- equals is transitive
- equals is consistent
- equals uses null as absorbing element

It's complicated!

Auto-generated
equals / hashcode

```
// Customer
public int hashCode() {
        int result = getName() != null ? getName().hashCode() : 0;
        result = 31 * result + (getCreditCard() != null ? getCreditCard().hashCode() : 0);
        result = 31 * result + (getOrders() != null ? getOrders().hashCode() : 0);
        return result;
}

// Order
public int hashCode() {
        int result = getCustomer() != null ? getCustomer().hashCode() : 0;
        result = 31 * result + (getItems() != null ? getItems().hashCode() : 0);
        result = 31 * result + (getStatus() != null ? getStatus().hashCode() : 0);
        return result;
}
```

Do not use Lombok or other similar frameworks

They could generate JPA incompatible equals or hashcode methods...

QUESTION

When are 2 customers equals?

Auto-generated equals / hashcode

IntelliJ – auto-generate – Objects.equals

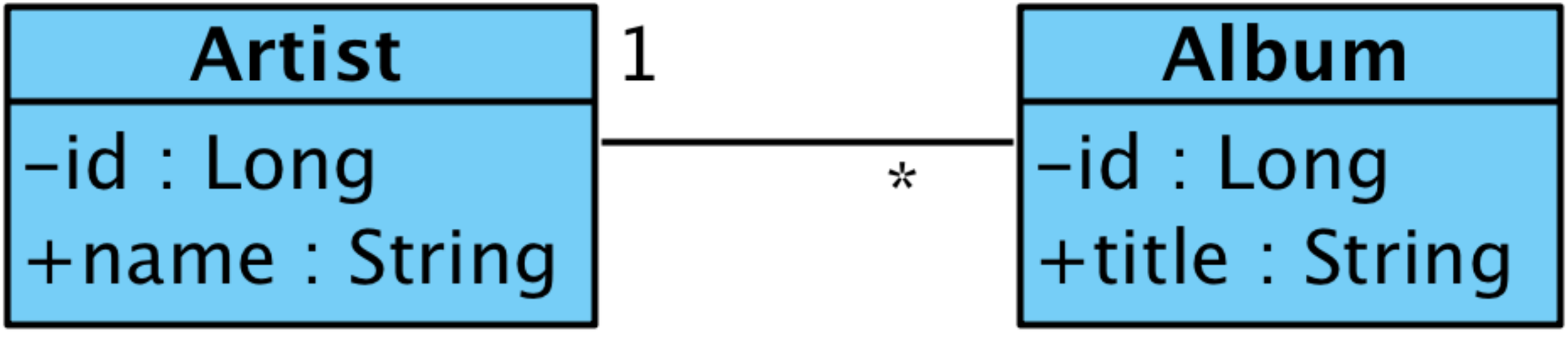
```java
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Customer customer)) return false;
    return Objects.equals(name, customer.name) && Objects.equals(creditCard, customer.creditCard);
}

@Override
public int hashCode() {
    return Objects.hash(name, creditCard);
}
```

Never ever use a database primary key as part of your business object equality definition

Equals is used when:

- putting objects in Sets
- when reattaching entities to a new persistence context (e.g., in a transaction)

Problem: Representing associations



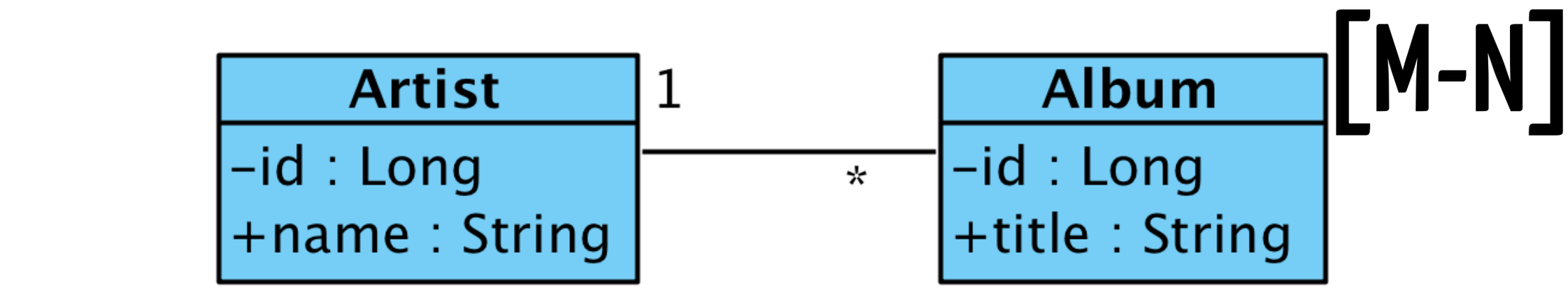
artists

id	name
1	Linkin Park
	…

albums

id	title
1	A Thousand Suns
2	Minutes to Midnight
	…

Solution #1: Association Table



artists

id	name
1	Linkin Park
…	

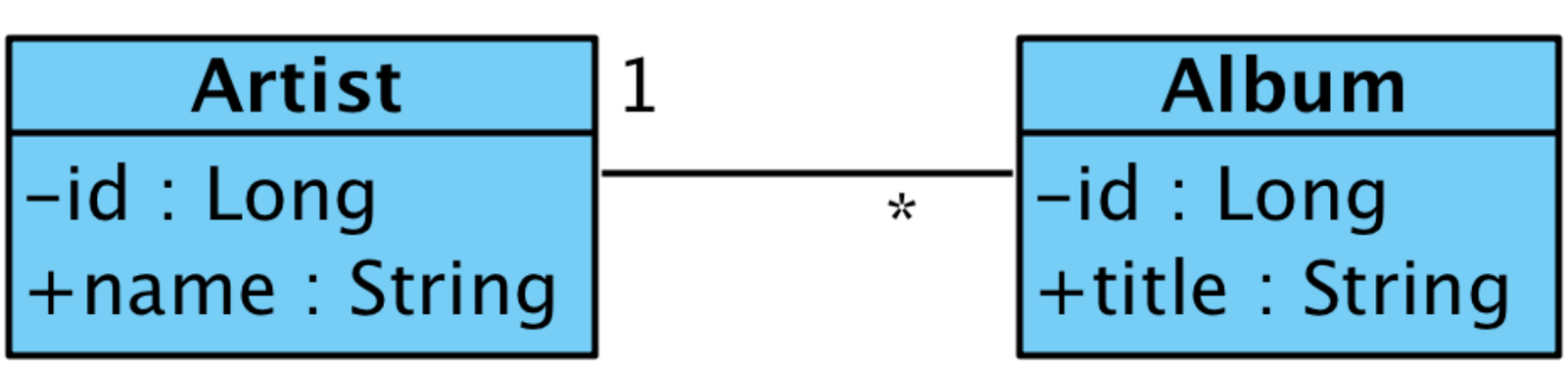
albums

id	title
1	A Thousand Suns
2	Minutes to Midnight
…	

artists_to_albums

artist_id	album_id
1	1
1	2
…	

Solution #2: Foreign Key



[1-N]

artists

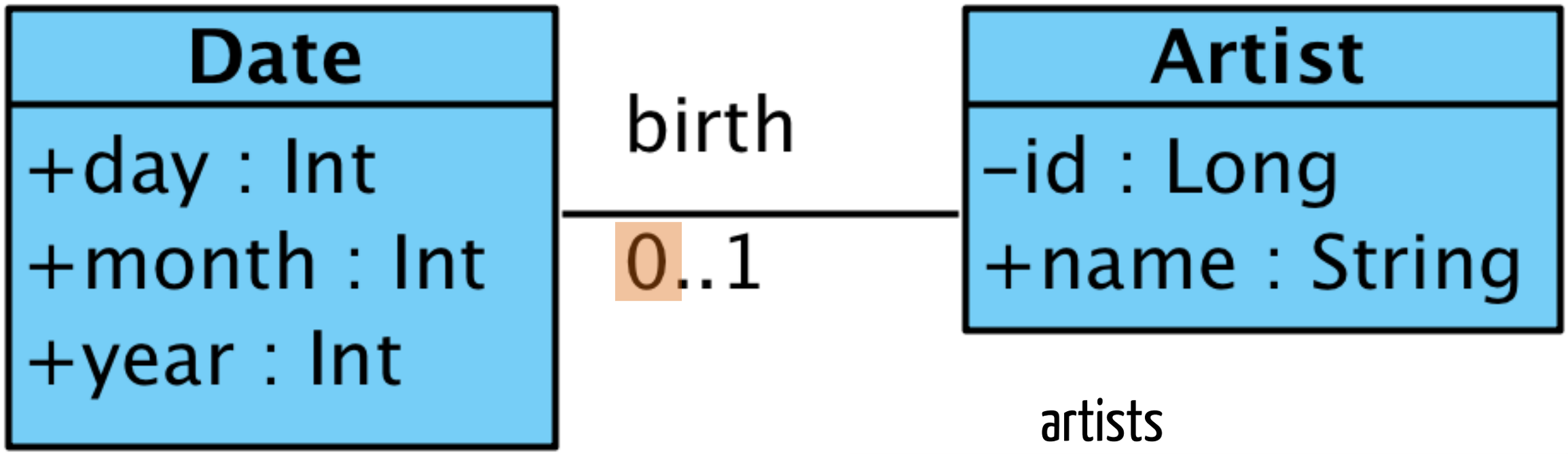
id	name
1	Linkin Park
...	

albums

id	title	artist_id
1	A Thousand Suns	1
2	Minutes to Midnight	1
...		

or [1-N] ≡ [M-N] when M =1

Solution #3: Relation Merge



or [1-1] ≡[1-N]
when N =1

artists

id	name	birth_day	birth_month	birth_year
1	Linkin Park	-1	-1	-1
…				

or [1-1] ≡[M-N] when M = 1 and N =1

Relationships

Type of Relationship	Annotation
1-1	`@OneToOne`
1-n	`@OneToMany`
n-1	`@ManyToOne`
n-m	`@ManyToMany`

Class/table mapping

```
@Entity
public class Bid {
   @Id
   protected String bidId;
   @ManytoOne
   protected Item item;
}

@Entity
public class Item {
   @Id
   protected String itemId;
   @OneToMany(mappedBy="item")
   protected Set<Bid> bids;
}
```

Id mapping

1-n mapping

QUESTION

How to bind customers to orders?

In TCF

```java
@Entity
public class Customer {

    @Id
    @GeneratedValue
    private Long id;

    @NotBlank
    @Column(unique = true)
    private String name;

    @Pattern(regexp = "\\d{10}+", message = "Invalid creditCardNumber")
    private String creditCard;

    @OneToMany(cascade = {CascadeType.REMOVE}, fetch = FetchType.LAZY, mappedBy = "customer")
    private Set<Order> orders = new HashSet<>();

    @ElementCollection
    private Set<Item> cart = new HashSet<>();
```

In TCF

```java
@Entity
@Table(name= "orders")
public class Order {

    @Id
    @GeneratedValue
    private Long id;

    @ManyToOne
    @NotNull
    private Customer customer;

    @ElementCollection
    private Set<Item> items;

    @Positive
    private double price;

    @NotBlank
    private String payReceiptId;

    @Enumerated(EnumType.STRING)
    @NotNull
    private OrderStatus status;
```

SQL Log (from a component test)

```
Hibernate:
    create table customer (
        id bigint not null,
        credit_card varchar(255),
        name varchar(255) unique,
        primary key (id)
    )
Hibernate:
    create table customer_cart (
        quantity integer not null,
        customer_id bigint not null,
        cookie varchar(255) check (cookie in ('CHOCOLALALA','DARK_TEMPTATION','SOO_CHOCOLATE'))
    )
Hibernate:
    create table order_items (
        quantity integer not null,
        order_id bigint not null,
        cookie varchar(255) check (cookie in ('CHOCOLALALA','DARK_TEMPTATION','SOO_CHOCOLATE'))
    )
Hibernate:
    create table orders (
        price float(53) not null,
        customer_id bigint,
        id bigint not null,
        pay_receipt_id varchar(255),
        status varchar(255) check (status in ('VALIDATED','IN_PROGRESS','READY')),
        primary key (id)
    )
```

```
Hibernate:
    alter table if exists customer_cart
       add constraint FKeaqo5asykq66fm0cfdrhoiyv7
       foreign key (customer_id)
       references customer
Hibernate:
    alter table if exists order_items
       add constraint FKbioxgbv59vetrxe0ejfubep1w
       foreign key (order_id)
       references orders
Hibernate:
    alter table if exists orders
       add constraint FK624gtjin3po807j3vix093tlf
       foreign key (customer_id)
       references customer
```

Make your entities persistent

101

The old DAO (Data Access Object) pattern

Pattern Context

- Access to data varies depending on the source of the data.

Pattern Motivation

- Persistent storage is implemented with different mechanisms
- Access data on separate systems
- Components need to be transparent to the actual persistent store or data source implementation

```
public interface IPersonDao {
    public Person find(String id);
    public List<Person> findAll();
    public void delete(String id);
    public void add(Person p);
}
```

DAO: basic internal JPA implementation

Advantages

- Enables Transparency
- Enables Easier Migration
- Reduces Code Complexity in Business Objects
- Centralizes All Data Access into a Separate Layer

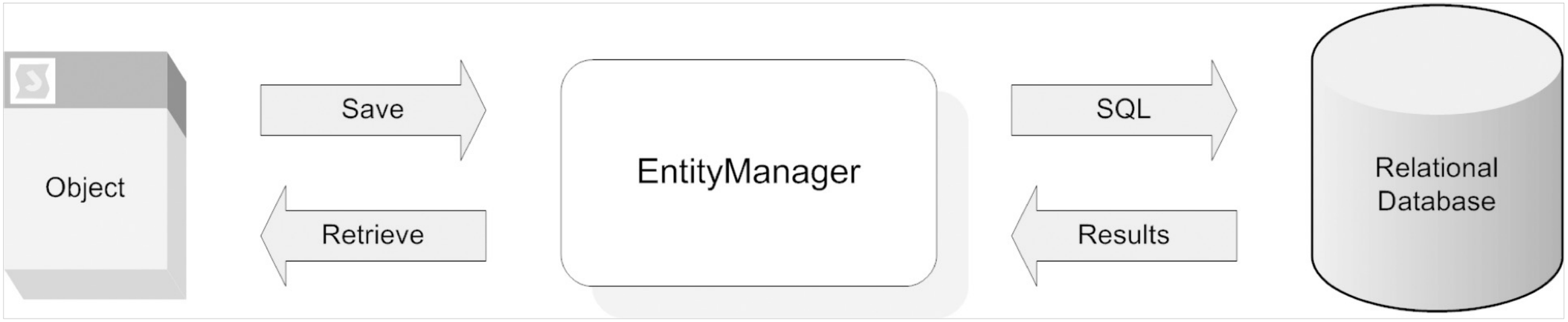
Drawbacks

- Adds Extra Layer

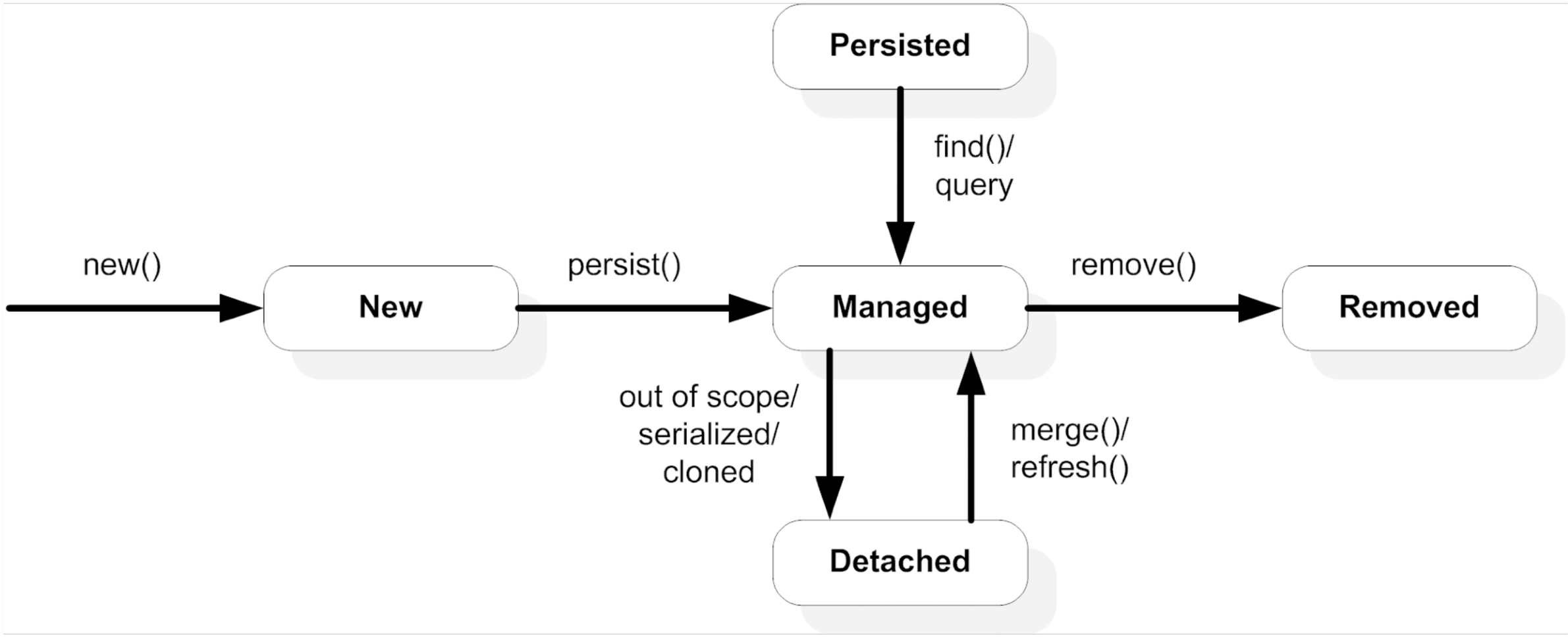
```java
@Repository
public class MeetingDao implements IMeetingDao {

    @PersistenceContext
    private EntityManager entityManager;

    @Transactional
    @Override
    public Meeting find(String id) {
        return entityManager.find(Meeting.class, id);
    }

    @Transactional
    @Override
    public void add(Meeting mr) {
        entityManager.persist(mr);
    }
```

will be hidden

JPA basic system
Entity Manager



JPA Entity Lifecycle



Handled by the provider

+ an entity manager in a persistent context

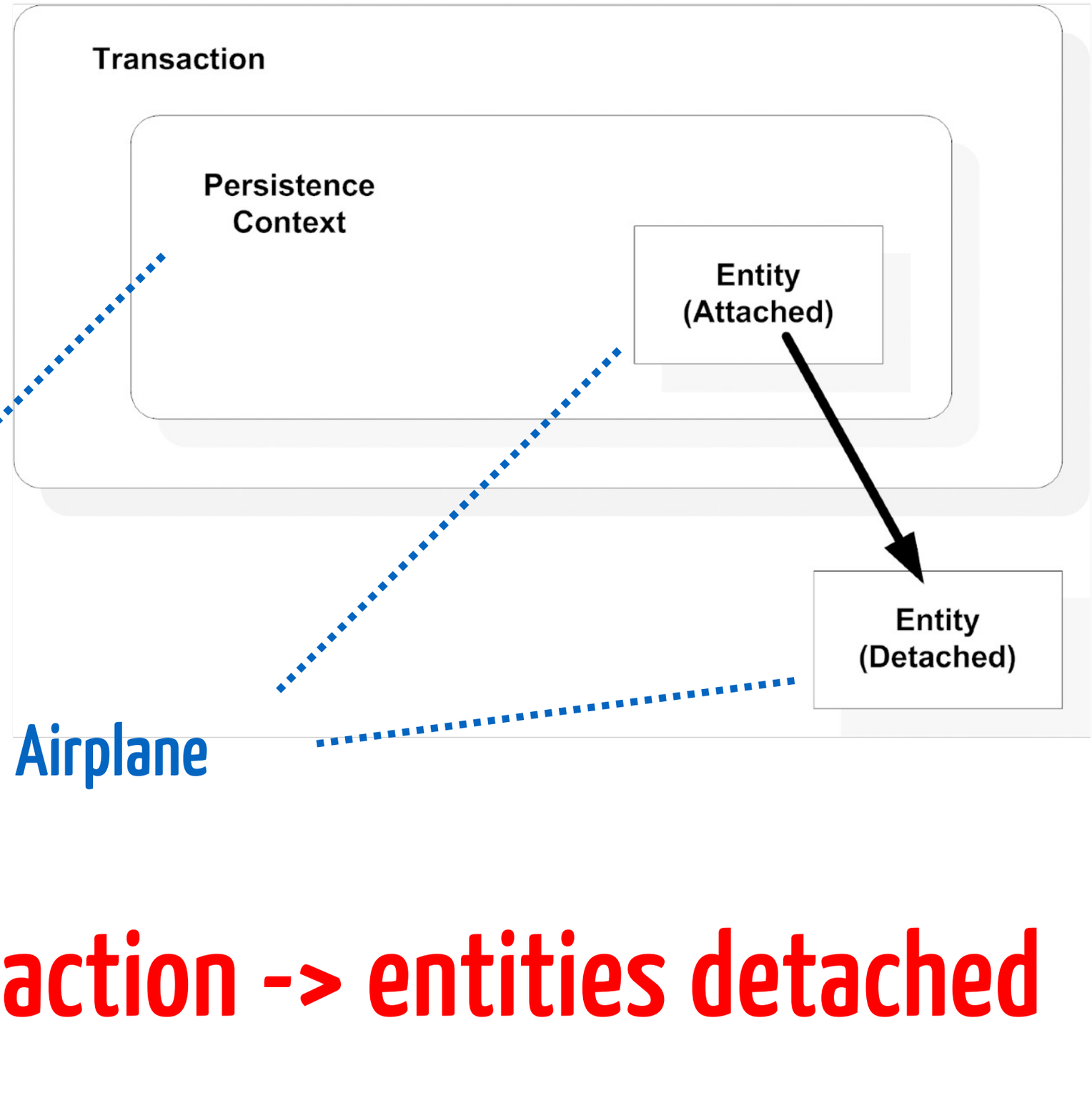

Transaction
Persistence
Context
Entity
(Attached)
Entity
(Detached)

Attach/Detach

EntityManager

=

Air Traffic Control



End of transaction -> entities detached

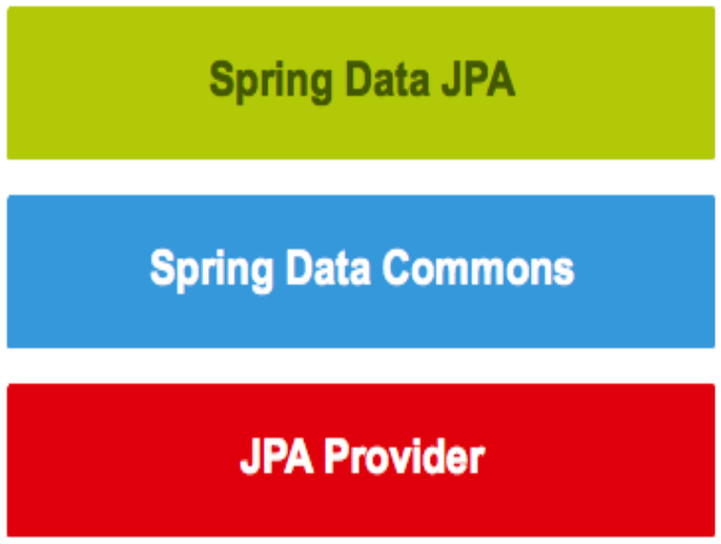
Spring Data: a framework on top of JPA

- Provides Boilerplate code for simple queries, pagination
- Hides persistent context and entity managers inside the Repository concept
- Spring Data is not a JPA Provider

Spring Data JPA provides support for creating JPA repositories by extending the Spring Data repository interfaces

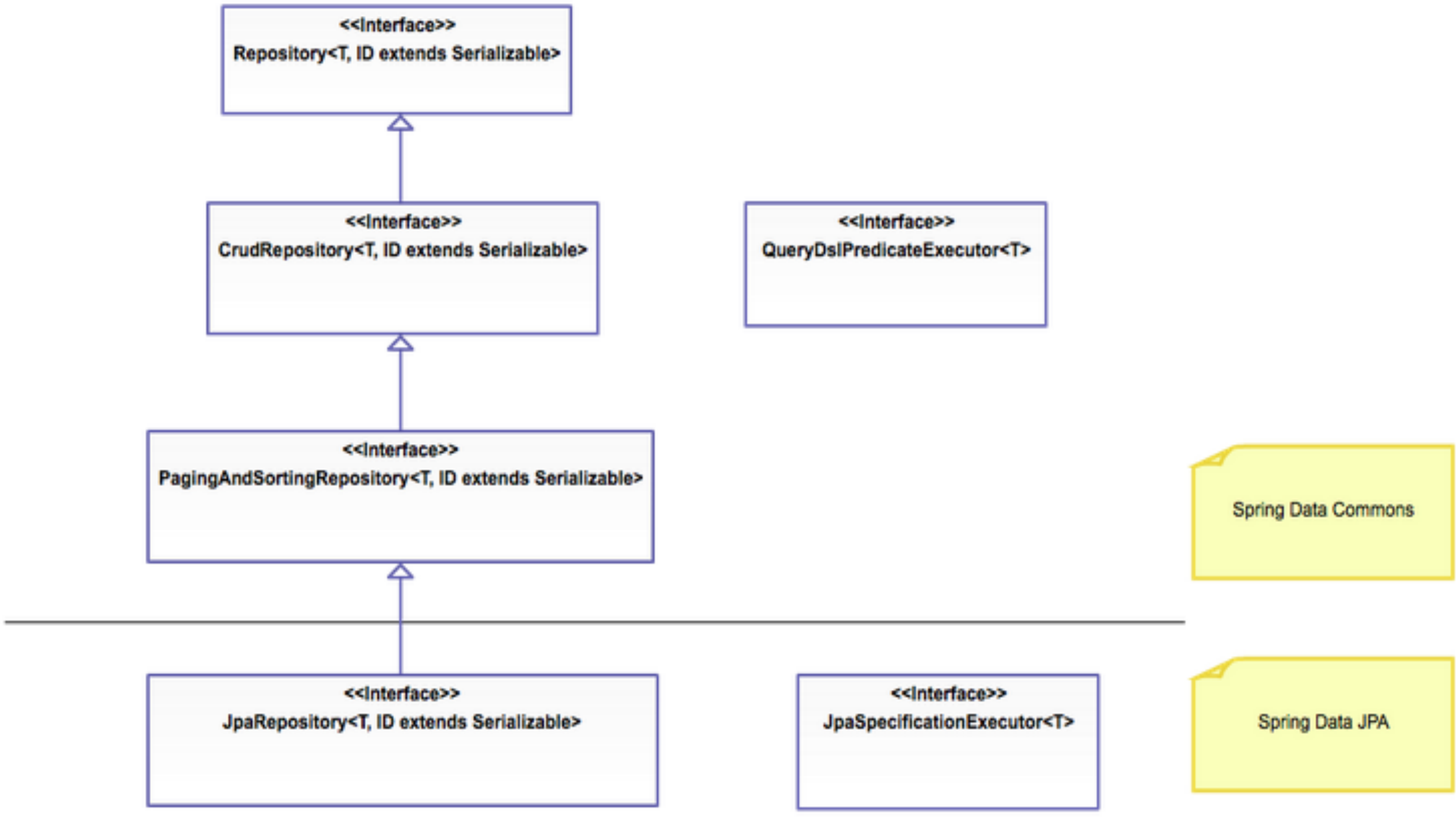
Spring Data Commons provides the infrastructure that is shared by the datastore specific Spring Data projects

The JPA Provider (Hibernate by default) implements the Java Persistence API



Repositories

```java
@Repository
public interface CustomerRepository extends JpaRepository<Customer, Long> {
    Optional<Customer> findCustomerByName(String name);
}
```



Generated repository

```java
@Entity
@Table(name= "orders")
public class Order {

    @Id
    @GeneratedValue
    private Long id;
```

```java
@Repository
public interface OrderRepository extends JpaRepository<Order, Long> {
}
```

deleteAllByIdInBatch(Iterable<ID> ids)
Deletes the entities identified by the given ids using a single query.

deleteAllInBatch()
Deletes all entities in a batch call.

deleteAllInBatch(Iterable<T> entities)
Deletes the given entities in a batch which means it will create a single query.

deleteInBatch(Iterable<T> entities)
Deprecated.
Use **deleteAllInBatch(Iterable)** instead.

findAll()

findAll(Example<S> example)

findAll(Example<S> example, Sort sort)

findAll(Sort sort)

findAllById(Iterable<ID> ids)

flush()
Flushes all pending changes to the database.

getById(ID id)
Returns a reference to the entity with the given identifier.

getOne(ID id)
Deprecated.
use `JpaRepository#getById(ID)` instead.

saveAll(Iterable<S> entities)

saveAllAndFlush(Iterable<S> entities)
Saves all entities and flushes changes instantly.

saveAndFlush(S entity)
Saves an entity and flushes changes instantly.

Repository usage

```
@Service
public class CustomerRegistry implements CustomerRegistration, CustomerFinder {

    private final CustomerRepository customerRepository;

    @Autowired // annotation is optional since Spring 4.3 if component has only one constructor
    public CustomerRegistry(CustomerRepository customerRepository) { this.customerRepository = customerRepository; }

    @Override
    @Transactional
    public Customer register(String name, String creditCard)
            throws AlreadyExistingCustomerException {
        if (findByName(name).isPresent())
            throw new AlreadyExistingCustomerException(name);
        Customer newcustomer = new Customer(name, creditCard);
        return customerRepository.save(newcustomer);
    }

    @Override
    @Transactional(readOnly = true)
    public Optional<Customer> findByName(String name) { return customerRepository.findCustomerByName(name); }

    @Override
    @Transactional(readOnly = true)
    public Optional<Customer> findById(Long id) { return customerRepository.findById(id); }
```

You need to think transactional and stateless

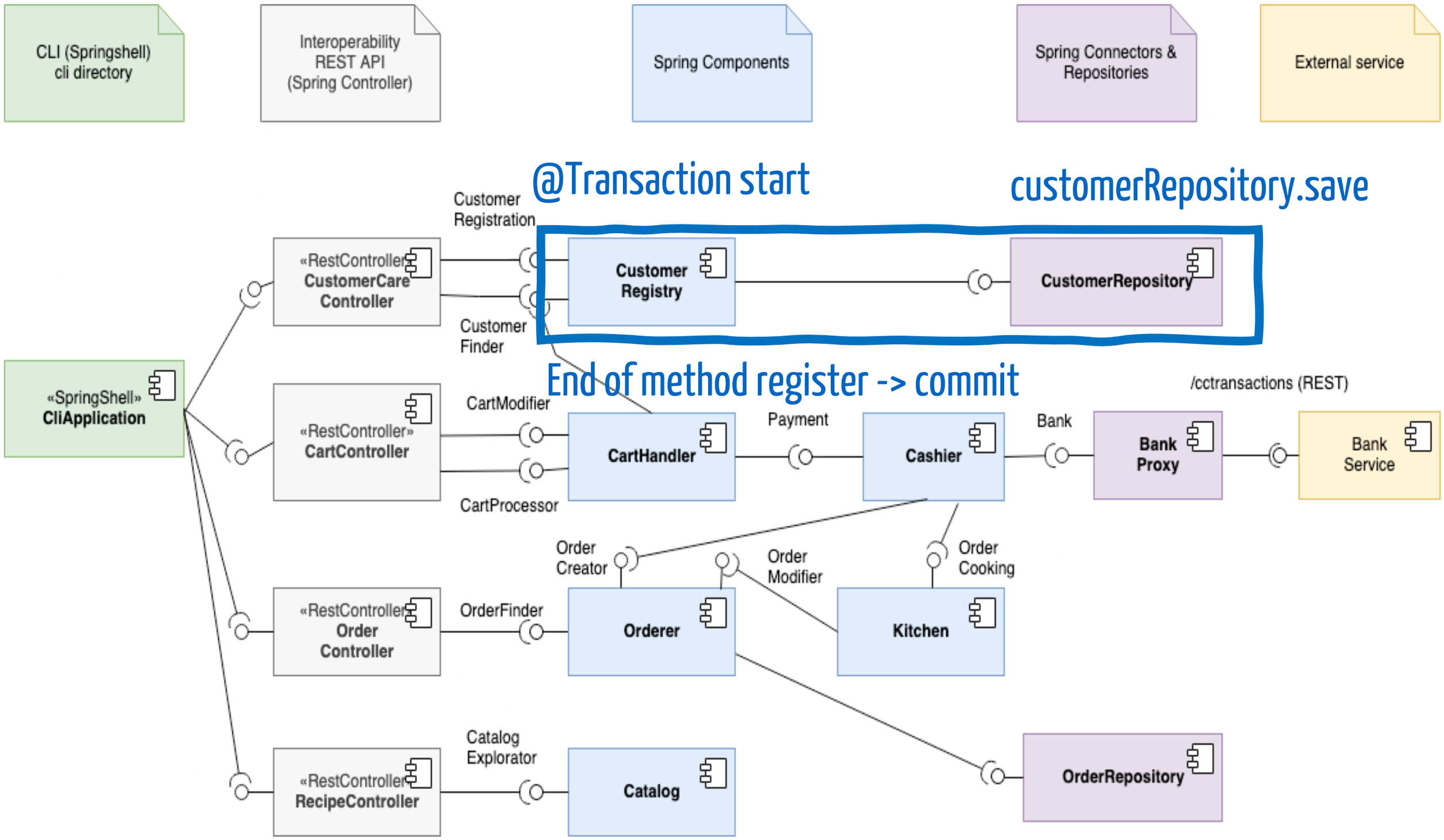
@Transactional basics

```java
@Service
public class CustomerRegistry implements CustomerRegistration, CustomerFinder {

    private final CustomerRepository customerRepository;

    @Autowired // annotation is optional since Spring 4.3 if component has only one constructor
    public CustomerRegistry(CustomerRepository customerRepository) { this.customerRepository = customerRepository; }

    @Override
    @Transactional
    public Customer register(String name, String creditCard)
            throws AlreadyExistingCustomerException {
        if (findByName(name).isPresent())
            throw new AlreadyExistingCustomerException(name);
        Customer newcustomer = new Customer(name, creditCard);
        return customerRepository.save(newcustomer);
    }

    @Override
    @Transactional(readOnly = true)
    public Optional<Customer> findByName(String name) { return customerRepository.findCustomerByName(name); }

    @Override
    @Transactional(readOnly = true)
    public Optional<Customer> findById(Long id) { return customerRepository.findById(id); }
```

Read-write transactions with explicit NEW object being saved

Read-only (no write lock)

@Transaction on CustomerRegistry::register



SQL Log (from the registry test)

```
2024-01-17T10:15:46.199+01:00  INFO 20421 --- [           main] f.u.s.components.CustomerRegistryTest    : Started
Hibernate:
    select
        c1_0.id,
        c1_0.credit_card,
        c1_0.name
    from
        customer c1_0
    where
        c1_0.name=?
Hibernate:
    select
        next value for customer_seq
Hibernate:
    insert
    into
        customer
        (credit_card, name, id)
    values
        (?, ?, ?)
```

```java
@Override
@Transactional
public Customer register(String name, String creditCard)
        throws AlreadyExistingCustomerException {
    if (findByName(name).isPresent())
        throw new AlreadyExistingCustomerException(name);
    Customer newcustomer = new Customer(name, creditCard);
    return customerRepository.save(newcustomer);
}
```

```java
@Override
@Transactional(readOnly = true)
public Optional<Customer> findByName(String name) {
    return customerRepository.findCustomerByName(name);
}
```

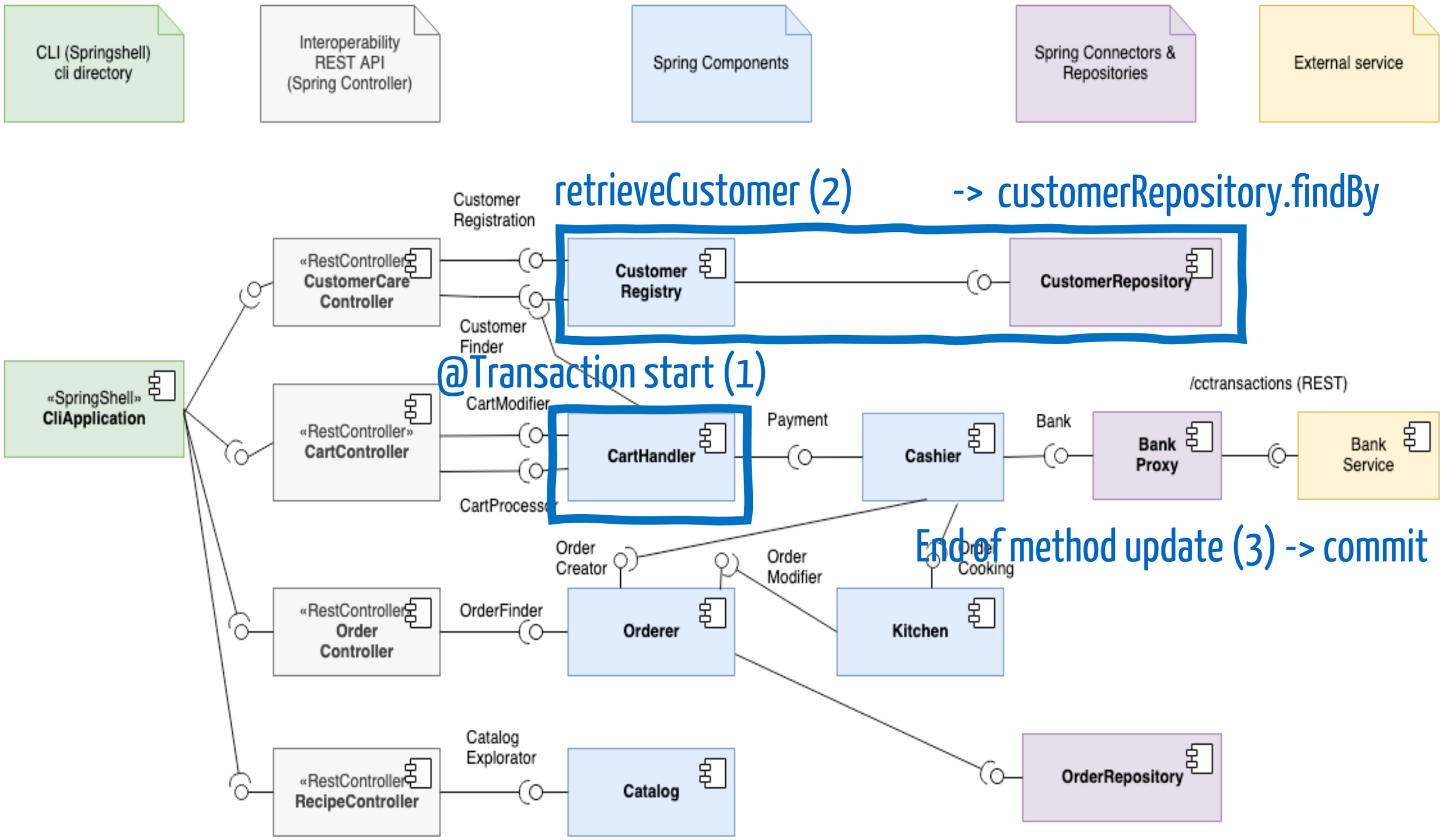
Transactional CartHandler

```java
@Service
public class CartHandler implements CartModifier, CartProcessor {

    private static final Logger LOG = LoggerFactory.getLogger(CartHandler.class);

    private final Payment payment;

    private final CustomerFinder customerFinder;

    @Autowired
    public CartHandler(Payment payment, CustomerFinder customerFinder) {
        this.payment = payment;
        this.customerFinder = customerFinder;
    }

    @Override
    @Transactional
    public Item update(Long customerId, Item item) throws NegativeQuantityException, CustomerIdNotFoundException {
        Customer customer = customerFinder.retrieveCustomer(customerId);
        // some very basic logging (see the AOP way for a more powerful approach, in class ControllerLogger)
        LOG.info("TCFS:Cart-Component: Updating cart of {} with {}", customer.getName(), item);
        int newQuantity = item.getQuantity();
        Set<Item> items = customer.getCart();
        Optional<Item> existing = items.stream().filter(e -> e.getCookie().equals(item.getCookie())).findFirst();
        if (existing.isPresent()) {
            newQuantity += existing.get().getQuantity();
        }
        if (newQuantity < 0) {
            throw new NegativeQuantityException(customer.getName(), item.getCookie(), newQuantity);
        } else {
            existing.ifPresent(items::remove);
            if (newQuantity > 0) {
                items.add(new Item(item.getCookie(), newQuantity));
            }
        }
        return new Item(item.getCookie(), newQuantity);
    }
```

No repository, only other components (accessing repos)

But at the end of the method, the scope is computed and objects are updated in the DB by the internal entity manager

@Transaction on CartHandler::update



SQL Log (from the update test - beforeEach)

```
2024-01-17T10:42:13.386+01:00  INFO 20766 --- [           main] f.u.s.components.CartHandlerTest
Hibernate:
    select
        c1_0.id,
        c1_0.credit_card,
        c1_0.name
    from
        customer c1_0
    where
        c1_0.name=?
Hibernate:
    select
        next value for customer_seq
Hibernate:
    insert
    into
        customer
        (credit_card, name, id)
    values
        (?, ?, ?)
```

```java
@BeforeEach
void setUp() throws AlreadyExistingCustomerException {
    johnId = customerRegistration.register( name: "John",  creditCard: "1234567890").getId();
}
```

```java
@Override
@Transactional
public Customer register(String name, String creditCard)
        throws AlreadyExistingCustomerException {
    if (findByName(name).isPresent())
        throw new AlreadyExistingCustomerException(name);
    Customer newcustomer = new Customer(name, creditCard);
    return customerRepository.save(newcustomer);
}
```

```java
@Override
@Transactional(readOnly = true)
public Optional<Customer> findByName(String name) {
    return customerRepository.findCustomerByName(name);
}
```

SQL Log (from the update test - addItems)

```
@Test
void addItems() throws NegativeQuantityException, CustomerIdNotFoundException {
    Item itemResult = cartModifier.update(johnId, new Item(Cookies.CHOCOLALALA,  quantity: 2));
```

```
@Transactional
public Item update(Long customerId, Item item) throws NegativeQuantityException, CustomerIdNotFoundException {
    Customer customer = customerFinder.retrieveCustomer(customerId);
    // some very basic [illegible] (see the AOP [illegible] for a more powerful approach, in class ControllerLogger)
    LOG.info("TCFS:Cart-Component: Updating cart of {} with {}", customer.getName(), item);
```

```
@Transactional(readOnly = true)
public Customer retrieveCustomer(Long customerId) throws CustomerIdNotFoundException {
    return findById(customerId).orElseThrow(() -> new CustomerIdNotFoundException(customerId));
}
```

```
@Transactional(readOnly = true)
public Optional<Customer> findById(Long id) { return customerRepository.findById(id) }
```

```
Hibernate:
    select
        c1_0.id,
        c2_0.customer_id,
        c2_0.cookie,
        c2_0.quantity,
        c1_0.credit_card,
        c1_0.name
    from
        customer c1_0
    left join
        customer_cart c2_0
            on c1_0.id=c2_0.customer_id
    where
        c1_0.id=?
2024-01-17T10:42:13.789+01:00  INFO 20766 --- [           main] f.u.simpletcfs.components.CartHandler   : TCFS:Cart-Component: Updating cart of John with 2xCHOCOLALALA
Hibernate:
    insert
    into
        customer_cart
        (customer_id, cookie, quantity)
    values
        (?, ?, ?)
```

SQL Log (from the update test - addItems)

```java
@Test
void addItems() throws NegativeQuantityException, CustomerIdNotFoundException {
    Item itemResult = cartModifier.update(johnId, new Item(Cookies.CHOCOLALALA, quantity: 2));
```

```java
@Transactional
public Item update(Long customerId, Item item) throws NegativeQuantityException, CustomerIdNotFoundException {
    Customer customer = customerFinder.retrieveCustomer(customerId);
    // some very basic logging (see the AOP way for a more powerful approach, in class ControllerLogger)
    LOG.info("TCFS:Cart-Component: Updating cart of {} with {}", customer.getName(), item);
    int newQuantity = item.getQuantity();
    Set<Item> items = customer.getCart();
    Optional<Item> existing = items.stream().filter(e -> e.getCookie().equals(item.getCookie())).findFirst();
    if (existing.isPresent()) {
        newQuantity += existing.get().getQuantity();
    }
    if (newQuantity < 0) {
        throw new NegativeQuantityException(customer.getName(), item.getCookie(), newQuantity);
    } else {
        existing.ifPresent(items::remove);
        if (newQuantity > 0) {
            items.add(new Item(item.getCookie(), newQuantity));
        }
    }
    return new Item(item.getCookie(), newQuantity);
}
```

Reference on the set of items

Change on the set of items

End of method -> commit

```
Hibernate:
    select
        c1_0.id,
        c2_0.customer_id,
        c2_0.cookie,
        c2_0.quantity,
        c1_0.credit_card,
        c1_0.name
    from
        customer c1_0
    left join
        customer_cart c2_0
            on c1_0.id=c2_0.customer_id
    where
        c1_0.id=?
2024-01-17T10:42:13.789+01:00  INFO 20766 --- [           main] f.u.simpletcfs.components.CartHandler    : TCFS:Cart-Component: Updating cart of John with 2xCHOCOLALALA
Hibernate:
    insert
    into
        customer_cart
        (customer_id, cookie, quantity)
    values
        (?, ?, ?)
```

Insert only in the customer_cart table

Set up with 2 databases (dev, test)

src/main/resources/application.yaml

```yaml
# postgres DB # IN DOCKER COMPOSE SHOULD BE OVERRIDEN BY ENV VARIABLES

spring:
  datasource:
    # POSTGRES_USER
    username: postgresuser
    # POSTGRES_PASSWORD
    password: postgrespass
    url: jdbc:postgresql://${POSTGRES_HOST}/tcf-db
    driver-class-name: org.postgresql.Driver

jpa:
  properties:
    hibernate:
      dialect: org.hibernate.dialect.PostgreSQLDialect
      format_sql: true
  show-sql: true
  generate-ddl: true
  open-in-view: false
```

Ensures Controllers do not start a default transaction
(bad conceptually and bad for performance)

src/test/resources/application.yaml

```yaml
spring:
  main:
    banner-mode: off
  jpa:
    properties:
      hibernate:
        format_sql: true
    show-sql: true
    open-in-view: false

logging:
  level:
    org:
      springframework: ERROR
```

Test DB setup omitted
(default is H2, a in-memory java DB)

Only configuration changes are done (e.g.
display generated SQL statements)

Postgres in docker's compose.yaml

```yaml
# the postgres DB to be connected to the backend (watch out: no volume specified, everything can be lost)
postgres:
  image: postgres:18.1
  container_name: db
  environment:
    - POSTGRES_PASSWORD=postgrespass
    - POSTGRES_USER=postgresuser
    - POSTGRES_DB=tcf-db
  restart: always
  healthcheck:
    test: [ "CMD-SHELL", "pg_isready -d tcf-db -U $$POSTGRES_USER" ]
    interval: 5s
    timeout: 3s
    retries: 3
    start_period: 5s
```

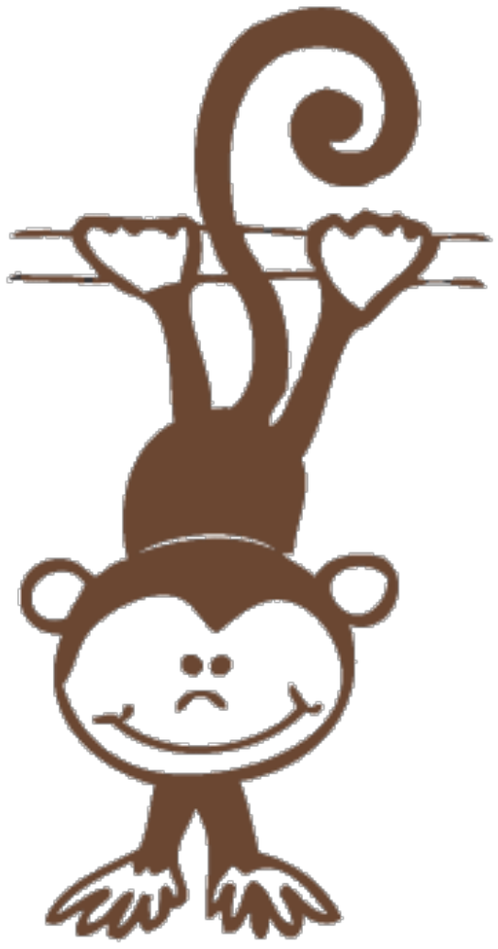
```yaml
# The Cookie Factory backend in SpringBoot
tcf-server:
  image: pcollet/tcf-spring-backend
  container_name: backend
  ports:
    - "8080:8080"
  environment:
    - BANK_WITH_PORT=bank-system:9090
    - POSTGRES_HOST=postgres:5432
  restart: always
  depends_on:
    bank-system:
      condition: service_healthy
    postgres:
      condition: service_healthy
  healthcheck:
    test: "curl --silent --fail localhost:8080/actuator/health | grep UP || exit 1"
    interval: 5s
    timeout: 3s
    retries: 3
    start_period: 10s
```

Psql within the postgres docker image

```
) docker exec -it db psql -U postgresuser -W -d tcf-db
Password:
psql (16.1 (Debian 16.1-1.pgdg120+1))
Type "help" for help.

tcf-db=# \dt+
                                    List of relations
 Schema |     Name      | Type  |    Owner     | Persistence | Access method |    Size    | Description
--------+---------------+-------+--------------+-------------+---------------+------------+-------------
 public | customer      | table | postgresuser | permanent   | heap          | 16 kB      |
 public | customer_cart | table | postgresuser | permanent   | heap          | 8192 bytes |
 public | order_items   | table | postgresuser | permanent   | heap          | 8192 bytes |
 public | orders        | table | postgresuser | permanent   | heap          | 16 kB      |
(4 rows)

tcf-db=# SELECT * FROM customer;
 id | credit_card | name
----+-------------+-------
  1 | 5251896983  | kadoc
  2 | 1234567890  | tatie
(2 rows)

tcf-db=# SELECT * FROM orders;
 id |                pay_receipt_id                | price |   status    | customer_id
----+----------------------------------------------+-------+-------------+-------------
  1 | RECEIPT:74d81b1f-5b39-4e8f-bea7-0a055b1ad714 |  18.1 | IN_PROGRESS |           1
  2 | RECEIPT:1dd4e455-fcf1-4b93-9d12-7474da3db0e7 |   2.5 | IN_PROGRESS |           1
(2 rows)

tcf-db=#
```

https://github.com/CookieFactoryInSpring/simpleTCFS

monkey see



monkey do