Tópicos em Frameworks

Cássio Seffrin

O que é um Framework?

- Abstração de código entre vários projetos.
- Frameworks VS Bibliotecas (classes)
- exemplos de bibliotecas: jquery, lodash, underscore …

Framework...

- Um framework em desenvolvimento de software, é uma abstração que une códigos comuns entre vários projetos de software provendo uma funcionalidade genérica. Um framework pode atingir uma funcionalidade específica, por configuração, durante a programação de uma aplicação. Ao contrário das bibliotecas, é o framework quem dita o fluxo de controle da aplicação, chamado de Inversão de Controle.
  Fonte: wiki

- Todo o fluxo de controle já está lá, existem brechas predefinidas onde devemos preencher com nosso código.

- Um framework é normalmente mais complexo. Ele vai controlar o todo fluxo, seu código será chamado pela estrutura quando apropriado.

- O benefício é que os desenvolvedores não precisam se preocupar com o design, mas apenas sobre a implementação de funções específicas.

- Estes fatores nos fazem pensar muito antes de escolher o framework mais adequado a nosso projeto.

Exemplos

Vert.X
Spring MVC – Model View Controller
Spring (POA)
Spring Data (https://spring.io/projects/spring-data)
GWT – Google Web Toolkit
Grails Web Framework
Struts
JSF – JavaServer Faces
Cake PHP
Zend Framework
Laravel
Hibernate
Doctrine
Flyway - database migration / liquibase
Eureka - Service Discovery Framework
JUnit

Principais Vantagens

- As Principais vantagens para o uso de framework são:

- Utilidade: O objetivo primeiro dos frameworks é auxiliar no desenvolvimento de aplicações e softwares. Para tal, eles têm funcionalidades nativas das mais variadas, que ajudam você a resolver as questões sobre programação do dia-a-dia com muito mais qualidade e eficiência.

- Segurança: Os bons frameworks são projetados de modo a garantir a segurança de quem programa e, principalmente, de quem usa o que foi feito a partir dele. Não se preocupe mais com aquelas intermináveis linhas de código para evitar um SQL Injection, por exemplo; com frameworks, a parte de segurança já “vem de fábrica”.

- Extensibilidade: Os frameworks permitem que você entenda suas funcionalidades nativas. Se aquela biblioteca de envio de e-mails por SMTP não contempla todas as possibilidades que você gostaria, simplesmente entenda suas funcionalidades e as use como se fossem parte do framework (na verdade, elas serão).

- Economia de tempo: O que você demoraria algumas horas ou alguns dias para fazer, você encontra pronto em um framework. Pense no quão são trabalhosas aquelas funções de manipulação de imagens são; usando um framework que tenha isso, você só usa, e pronto.

um pouco de prática

- https://spring.io/tools
- mysql
- git

Spring Boot

O Spring Boot facilita a criação de aplicativos baseados em Spring autônomos e de produção que você pode "executar".

Adotamos uma visão opinativa da plataforma Spring e de bibliotecas de terceiros para que você possa começar com o mínimo de stress. A maioria dos aplicativos Spring Boot precisa de uma configuração de Spring muito pequena.

Características

- Crie aplicativos Spring independentes
- Incorpore Tomcat, Jetty ou Undertow diretamente (não é necessário implantar arquivos WAR)
- Forneça dependências 'iniciais' com opinião para simplificar sua configuração de criação
- Configurar automaticamente bibliotecas Spring e de terceiros sempre que possível
- Fornecer recursos prontos para produção, como métricas, verificações de integridade e configuração externalizada
- Absolutamente nenhuma geração de código e nenhum requisito para configuração XML

Projeto Escola

- https://start.spring.io/

- Spring Data JPA

- Spring Web Starter

- Mysql Driver

- Lombok
  https://projectlombok.org/downloads/lombok.jar

Spring Framework

Spring Framework

Origem: Wikipédia, a enciclopédia livre.

- O Spring é um framework open source para a plataforma Java criado por Rod Johnson e descrito em seu livro "Expert One-on-One: JEE Design e Development". Trata-se de um framework não intrusivo, baseado nos padrões de projeto inversão de controle (IoC) e injeção de dependência.

- No Spring o container se encarrega de "instanciar" classes de uma aplicação Java e definir as dependências entre elas através de um arquivo de configuração em formato XML, inferências do framework, o que é chamado de auto-wiring ou ainda anotações nas classes, métodos e propriedades. Dessa forma o Spring permite o baixo acoplamento entre classes de uma aplicação orientada a objetos.

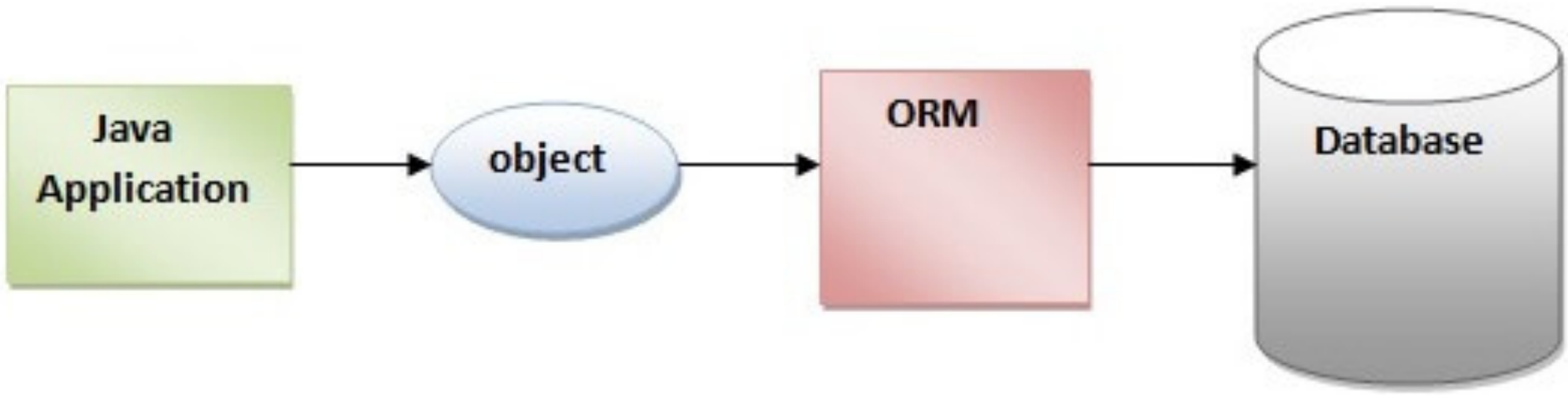
- O Spring possui uma arquitetura baseada em interfaces e POJOs (Plain Old Java Objects), oferecendo aos POJOs características como mecanismos de segurança e controle de transações. Também facilita testes unitários e surge como uma alternativa à complexidade existente no uso de EJBs. Com Spring, pode-se ter um alto desempenho da aplicacao.

- Esse framework oferece diversos módulos que podem ser utilizados de acordo com as necessidades do projeto, como módulos voltados para desenvolvimento Web, persistência, acesso remoto
e programação orientada a aspectos.

JPA

Ferramenta ORM

Uma ferramenta ORM simplifica a criação, manipulação e acesso a dados. É uma técnica de programação que mapeia o objeto para os dados armazenados no banco de dados.



JPA

- Aplicações corporativas manipulam dados em grande quantidade

- Esses dados são em sua maioria armazenados em banco de dados relacionais

- As aplicações corporativas costumam ser desenvolvidas com linguagens orientadas a objetos

- O modelo relacional e o modelo orientado a objetos diferem no modo de estruturar os dados

- Transformações devem ocorrer toda vez que uma informação trafegar da aplicação para o banco de dados ou vice-versa

JPA

- O JPA é o padrão de mapeamento objeto/entidade relacionamento e interface de gerência de persistência do JEE.

- Implementações do padrão JPA
  - Hibernate
  - EclipseLink
  - OpenJPA
  - …

JPA

What are the different ORM's supported by Spring?

Different ORM's supported by Spring are depicted via the below diagram:

Hibernate

Estrutura do Hibernate

O Hibernate é uma estrutura Java que simplifica o desenvolvimento de aplicativos Java para interagir com o banco de dados. É uma ferramenta de código aberto, leve, ORM (Object Relational Mapping). O Hibernate implementa as especificações da JPA (Java Persistence API) para persistência de dados.

Ferramenta ORM

Uma ferramenta ORM simplifica a criação, manipulação e acesso a dados. É uma técnica de programação que mapeia o objeto para os dados armazenados no banco de dados.

tutorial de hibernação, uma introdução ao hibernação

A ferramenta ORM usa internamente a API JDBC para interagir com o banco de dados.

O que é JPA?

A API de persistência de Java (JPA) é uma especificação Java que fornece certa funcionalidade e padrão para ferramentas ORM. O pacote javax.persistence contém as classes e interfaces JPA.

Vantagens do Hibernate Framework

A seguir, estão as vantagens da estrutura de hibernação:

1) Código Aberto e Leve

A estrutura do Hibernate é de código aberto sob a licença LGPL e é leve.

2) Desempenho Rápido

O desempenho da estrutura de hibernação é rápido porque o cache é usado internamente na estrutura de hibernação. Existem dois tipos de cache no cache de primeiro nível da estrutura de hibernação e no cache de segundo nível. O cache de primeiro nível está ativado por padrão.

3) Consulta Independente do Banco de Dados

HQL (Hibernate Query Language) é a versão orientada a objeto do SQL. Ele gera as consultas independentes do banco de dados. Portanto, você não precisa escrever consultas específicas do banco de dados. Antes do Hibernate, se o banco de dados for alterado para o projeto, também precisamos alterar a consulta SQL que leva ao problema de manutenção.

4) Criação automática de tabelas

A estrutura do Hibernate fornece a facilidade de criar as tabelas do banco de dados automaticamente. Portanto, não há necessidade de criar tabelas no banco de dados manualmente.

5) Simplifica a junção complexa

Buscar dados de várias tabelas é fácil na estrutura de hibernação.

6) Fornece estatísticas de consulta e status do banco de dados

O Hibernate suporta cache de consulta e fornece estatísticas sobre o status da consulta e do banco de dados.

JPA - ManyToOne

Anotação @JoinColumn

Em um relacionamento um para muitos / muitos para um, o lado proprietário geralmente é definido no lado 'muitos' do relacionamento. Geralmente é o lado que possui a chave estrangeira.

A anotação @JoinColumn define o mapeamento físico real no lado proprietário:

Atributo mappedBy

Depois de definirmos o lado proprietário do relacionamento, o Hibernate já possui todas as informações necessárias para mapear esse relacionamento em nosso banco de dados. Para tornar essa associação bidirecional, tudo o que precisamos fazer é definir o lado da referência. O lado inverso ou de referência simplesmente mapeia para o lado proprietário.

Podemos facilmente usar o atributo mappedBy da anotação @OneToMany para fazer isso. Então, vamos definir nossa entidade Turma:

o valor de mappedBy é o nome do atributo de mapeamento de associação no lado proprietário. Com isso, agora estabelecemos uma associação bidirecional entre nossos alunos e turmas.

JPA - ManyToOne

Aluno

```
    @ManyToOne(optional = false)
    private Turma turma;
```

Turma

```
    @OneToMany(cascade = CascadeType.ALL, mappedBy = "turma")
    private Collection<Aluno> alunoCollection;
```

```
mysql> desc aluno;
+----------+--------------+------+-----+---------+-------+
| Field    | Type         | Null | Key | Default | Extra |
+----------+--------------+------+-----+---------+-------+
| id       | int(11)      | NO   | PRI | NULL    |       |
| email    | varchar(255) | YES  |     | NULL    |       |
| nome     | varchar(255) | YES  |     | NULL    |       |
| turma_id | int(11)      | NO   | MUL | NULL    |       |
+----------+--------------+------+-----+---------+-------+

mysql> desc turma;
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| id    | int(11)      | NO   | PRI | NULL    |       |
| nome  | varchar(255) | YES  |     | NULL    |       |
+-------+--------------+------+-----+---------+-------+
```

OneToOne

- Implementando com uma chave estrangeira na JPA

- Colocamos a anotação @OneToOne no campo da entidade relacionada, Aluno.

- Além disso, precisamos colocar a anotação @JoinColumn para configurar o nome da coluna na tabela de usuários que mapeia para a chave primária na tabela de endereços. Se não fornecermos um nome, o Hibernate seguirá algumas regras para selecionar um padrão.

- Por fim, observe na próxima entidade que não usaremos a anotação @JoinColumn lá. Isso ocorre porque precisamos apenas do lado proprietário do relacionamento de chave estrangeira. Simplificando, quem possui a coluna de chave estrangeira recebe a anotação @JoinColumn.

- A entidade Endereco é mais simples:
  Também precisamos colocar a anotação @OneToOne. Isso porque esse é um relacionamento bidirecional. O lado do endereço do relacionamento é chamado de lado não proprietário.

OneToOne

- Aluno

```java
@OneToOne(cascade = CascadeType.ALL)
@JoinColumn(name = "endereco_id", referencedColumnName = "id")
private Endereco endereco;
```

- Endereco

```java
@OneToOne(mappedBy = "endereco")
private Aluno aluno;
```

```
desc aluno;
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| id          | int(11)      | NO   | PRI | NULL    |       |
| email       | varchar(255) | YES  |     | NULL    |       |
| nome        | varchar(255) | YES  |     | NULL    |       |
| turma_id    | int(11)      | NO   | MUL | NULL    |       |
| endereco_id | int(11)      | YES  | MUL | NULL    |       |
+-------------+--------------+------+-----+---------+-------+
desc endereco;
+--------+--------------+------+-----+---------+-------+
| Field  | Type         | Null | Key | Default | Extra |
+--------+--------------+------+-----+---------+-------+
| id     | int(11)      | NO   | PRI | NULL    |       |
| bairro | varchar(255) | YES  |     | NULL    |       |
| cidade | varchar(255) | YES  |     | NULL    |       |
| rua    | varchar(255) | YES  |     | NULL    |       |
+--------+--------------+------+-----+---------+-------+
```

JPA - ManyToMany

Turma:

```
@JoinTable(name = "turma_disciplina", joinColumns = {
        @JoinColumn(name = "turma_id", referencedColumnName =
"id") },
          inverseJoinColumns = {
        @JoinColumn(name = "disciplina_id", referencedColumnName
= "id") })
    @ManyToMany
    private Collection<Disciplina> disciplinaCollection;
```

Disciplina:

```
@ManyToMany(mappedBy = "disciplinaCollection")
     private Collection<Turma> turmaCollection;
```

JPA - ManyToMany

Resultado:

```
desc turmas_disciplinas;
+---------------+---------+------+-----+---------+-------+
| Field         | Type    | Null | Key | Default | Extra |
+---------------+---------+------+-----+---------+-------+
| turma_id      | int(11) | NO   | MUL | NULL    |       |
| disciplina_id | int(11) | NO   | MUL | NULL    |       |
+---------------+---------+------+-----+---------+-------+
```

```
desc turma;
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| id    | int(11)      | NO   | PRI | NULL    |       |
| nome  | varchar(255) | YES  |     | NULL    |       |
+-------+--------------+------+-----+---------+-------+
```

```
desc disciplina;
+-----------------+--------------+------+-----+---------+-------+
| Field           | Type         | Null | Key | Default | Extra |
+-----------------+--------------+------+-----+---------+-------+
| id              | int(11)      | NO   | PRI | NULL    |       |
| carga_horaria   | int(11)      | YES  |     | NULL    |       |
| nome_disciplina | varchar(255) | YES  |     | NULL    |       |
+-----------------+--------------+------+-----+---------+-------+\
```

JPA - ManyToMany

Como podemos ver, tanto a classe Turma quanto a Disciplina se referem uma à outra, o que significa que a associação entre elas é bidirecional.

Para mapear uma associação muitos-para-muitos, usamos as anotações @ManyToMany, @JoinTable e @JoinColumn. Vamos dar uma olhada neles.

A anotação @ManyToMany é usada nas duas classes para criar o relacionamento muitos-para-muitos entre as entidades.

Esta associação tem dois lados, isto é, o lado proprietário e o lado inverso. Em nosso caso, o lado proprietário é Turma, portanto a tabela de junção é especificada no lado proprietário usando a anotação @JoinTable na classe Turma. A @JoinTable é usada para definir a tabela de junção. Nesse caso, é turmas_disciplinas.

A anotação @JoinColumn é usada para especificar a coluna de junção / vinculação com a tabela principal. Aqui, a coluna de junção é disciplina_id e turma_id é a coluna de junção inversa, pois o Disciplina está no lado inverso do relacionamento.

Na classe Disciplina, o atributo mappedBy é usado na anotação @ManyToMany para indicar que a coleção de turmas é mapeada pela coleção de disciplinas (disciplinaCollection) do lado do proprietário.

JPA - ManyToMany
com uma nova entidade

Modelando Atributos de Relacionamento

Digamos que queremos permitir que os alunos se matriculem-se (inscrevam-se) nos cursos. Além disso, precisamos armazenar o ponto em que um aluno se matriculou em um curso específico.

Portanto, podemos introduzir uma entidade, que conterá os atributos do registro:

Nesse caso, a entidade de registro representa o relacionamento entre as outras duas entidades.

Por ser uma entidade, ela terá sua própria chave primária e duas chaves estrangeiras

JPA - ManyToMany com uma nova entidade

```
@Entity
class MatriculaDisciplina {
    @Id
    Long id;

    @ManyToOne
    @JoinColumn(name = "matricula_id")
    Matricula matricula;

    @ManyToOne
    @JoinColumn(name = "disciplina_id")
    Disciplina disciplina;
    LocalDateTime dataMatricula;
}
```

```
class Matricula {

…
@OneToMany(mappedBy = "matricula")
    Collection< MatriculaDisciplina > matriculas;

 …

}
```

```
class Disciplina {

…
    @OneToMany(mappedBy = "disciplina")
    Collection< MatriculaDisciplina > matriculas;

 …

}
```

JPA - ManyToMany

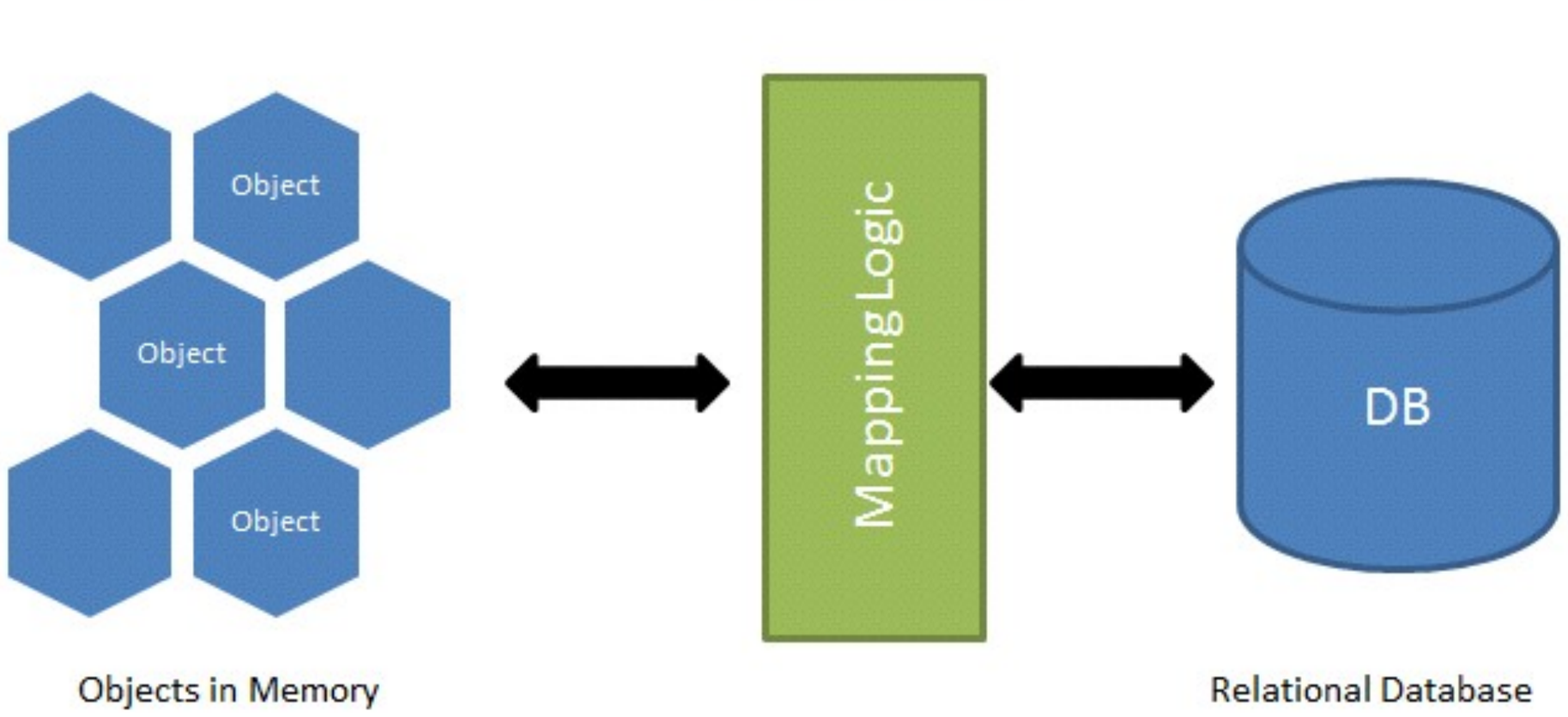
```
@Autowired
private TurmaRepository turmaRepository;

Optional<Turma> obj = turmaRepository.findById(turma);
Turma turmaEncontrada = new Turma();
if (obj.isPresent()){
        turmaEncontrada = obj.get();
}
n.setTurma(turmaEncontrada);
```

JPA - ManyToMany

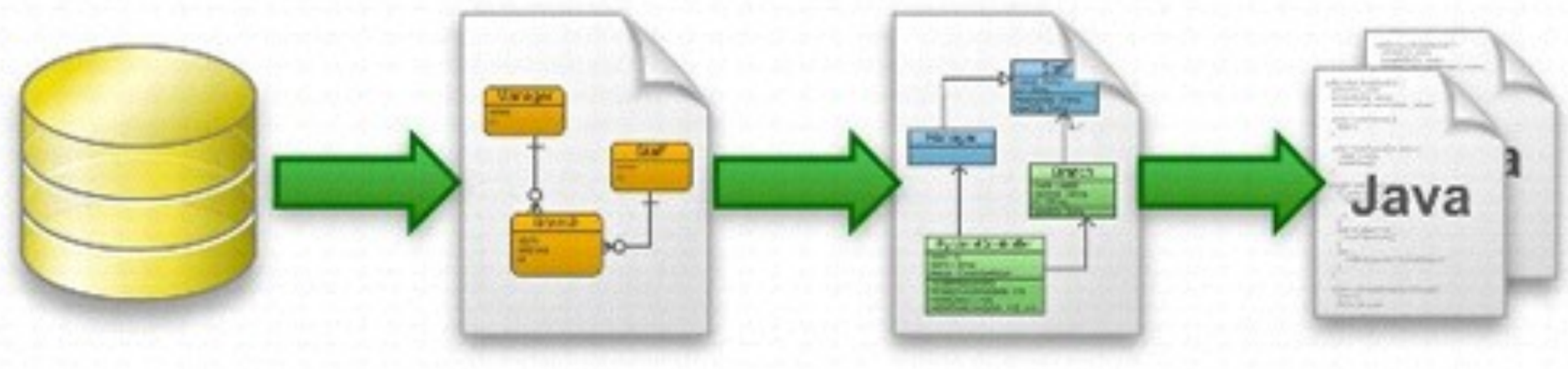
```
@Entity
public class Matricula implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

     @ManyToOne(optional = true)
     private Aluno aluno;

     @ManyToOne(optional = true)
     private Turma turma;

     @ManyToOne(optional = true)
     private Curso curso;

    @JoinTable(name = "matricula_disciplina", joinColumns = {
            @JoinColumn(name = "matricula_id", referencedColumnName =
"id") }, inverseJoinColumns = {
                    @JoinColumn(name = "disciplina_id",
referencedColumnName = "id") })
    @ManyToMany
    private Collection<Disciplina> disciplinaCollection;
}
```

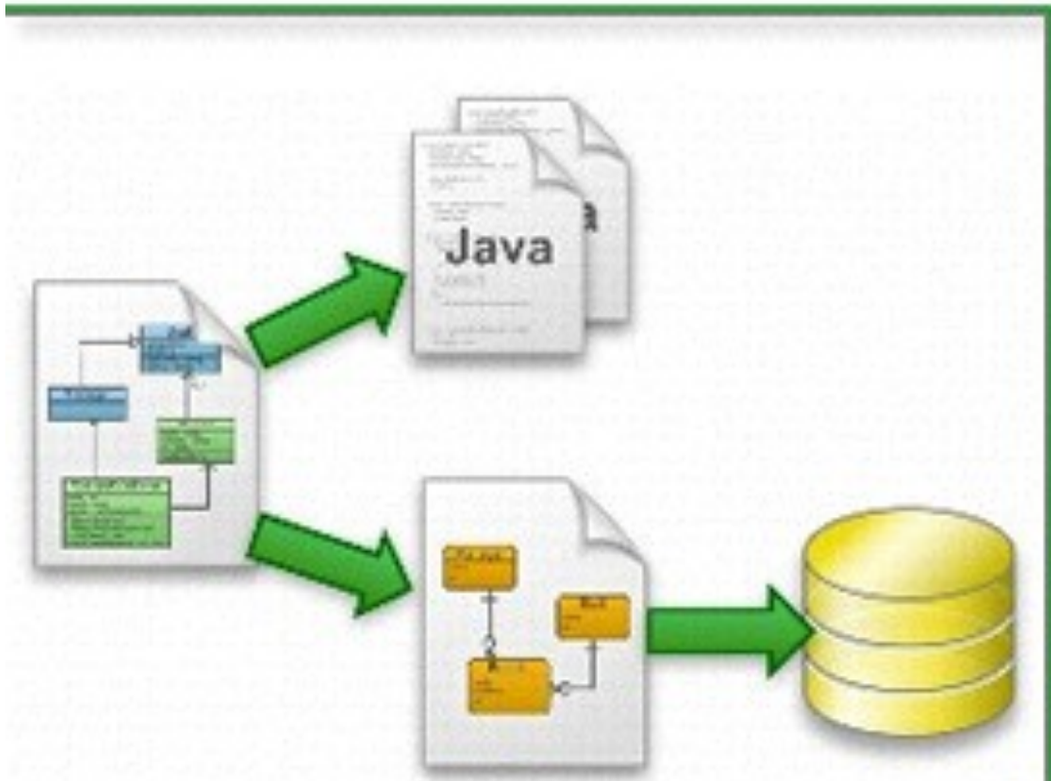
```java
public class FrutaFactory {

    private Connection c;

    public FrutaFactory (Connection c) {
        this.c = c;
    }

    public List<Fruta> getAll () throws SQLException {
        PreparedStatement p = this.c.prepareStatement("SELECT * FROM frutas;");
        ResultSet r = p.executeQuery();
        ArrayList <Fruta>list = new ArrayList <Fruta> ();
        while(r.next())
            list.add(new Fruta(r.getString("nome")));
        return list;
    }
}
```

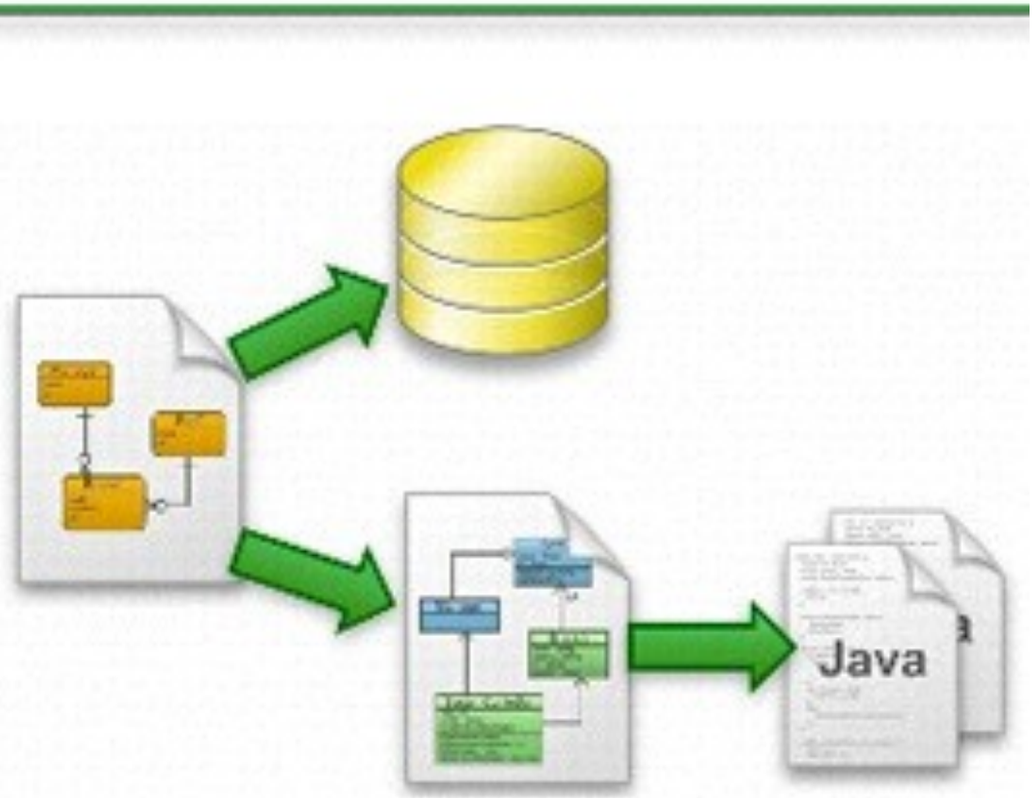
O/R Mapping



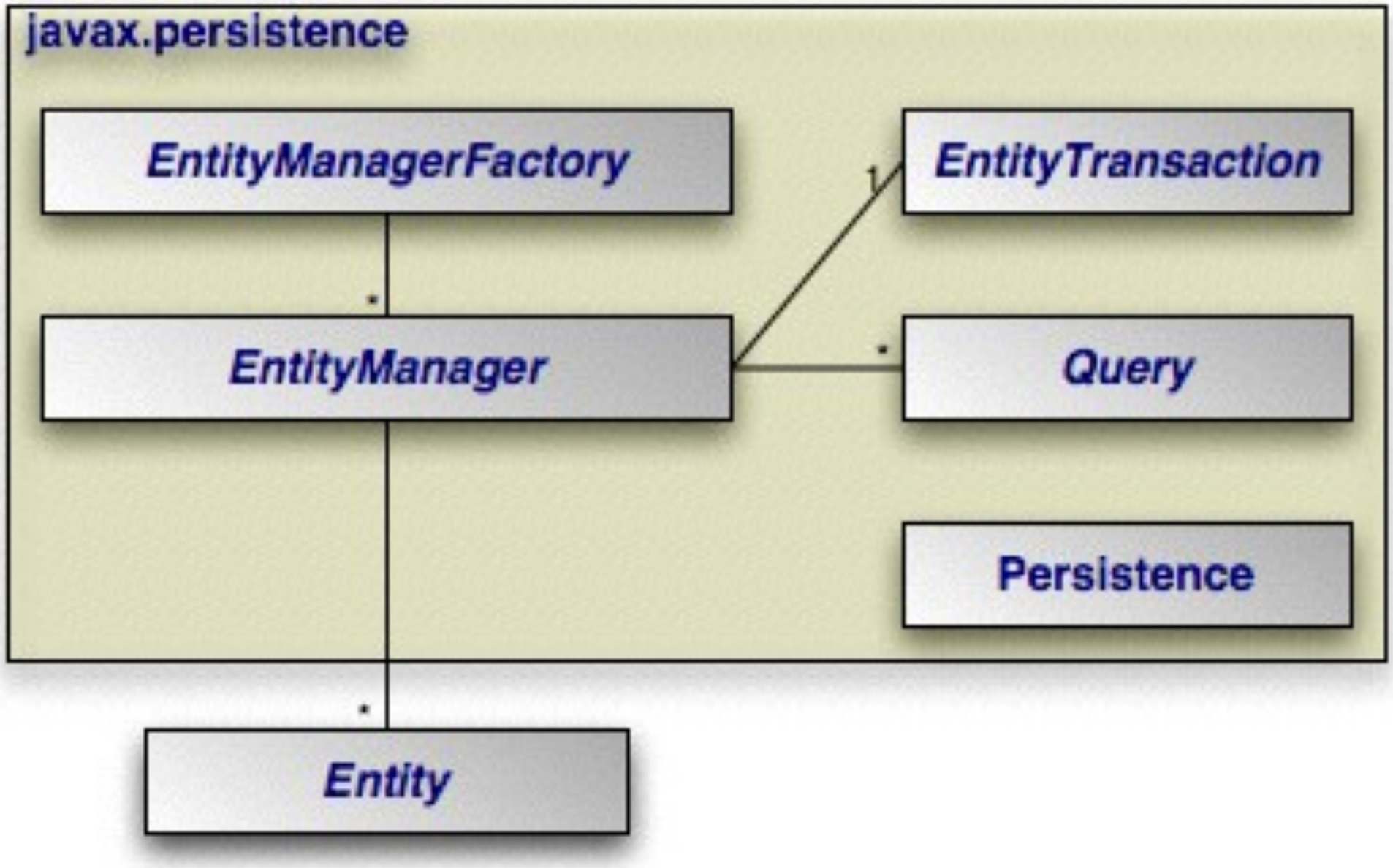
JPA



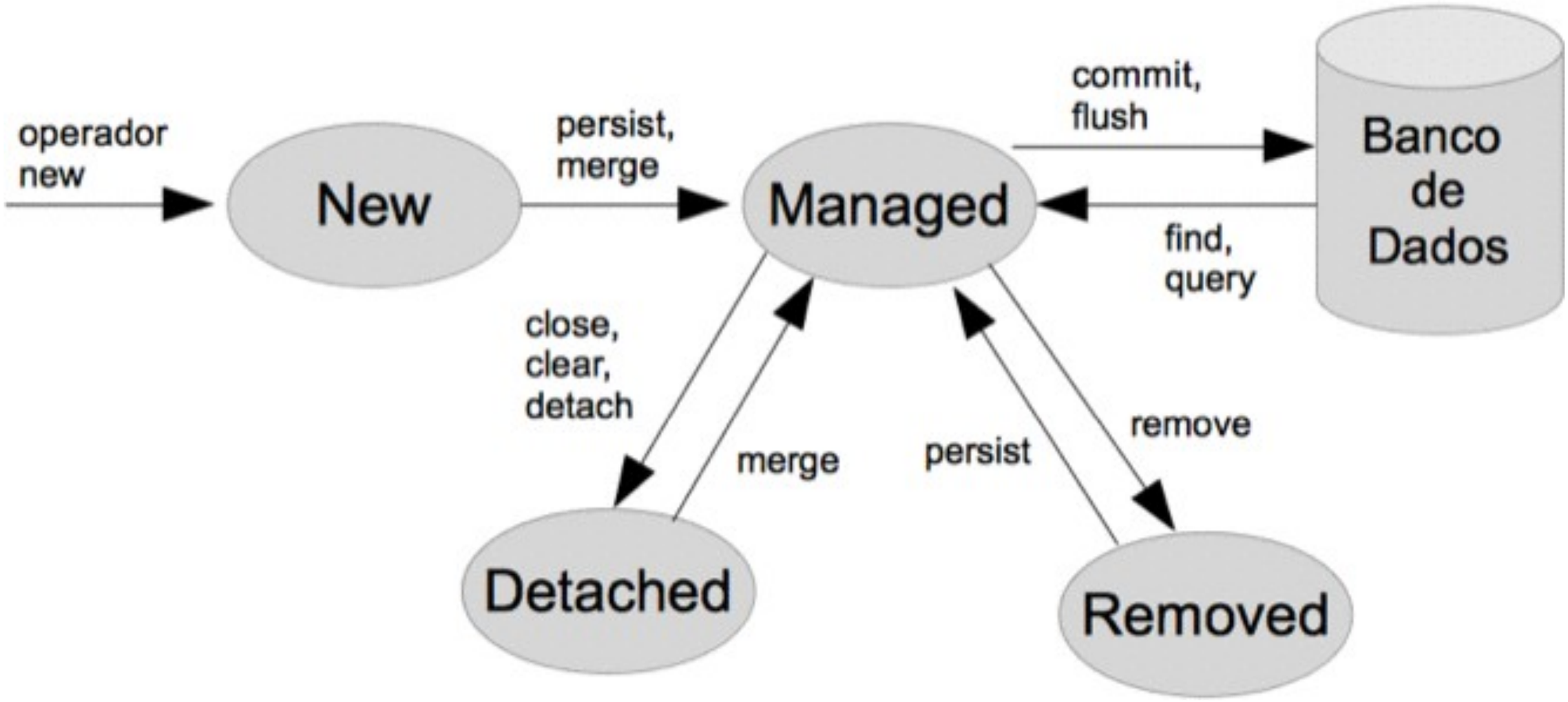




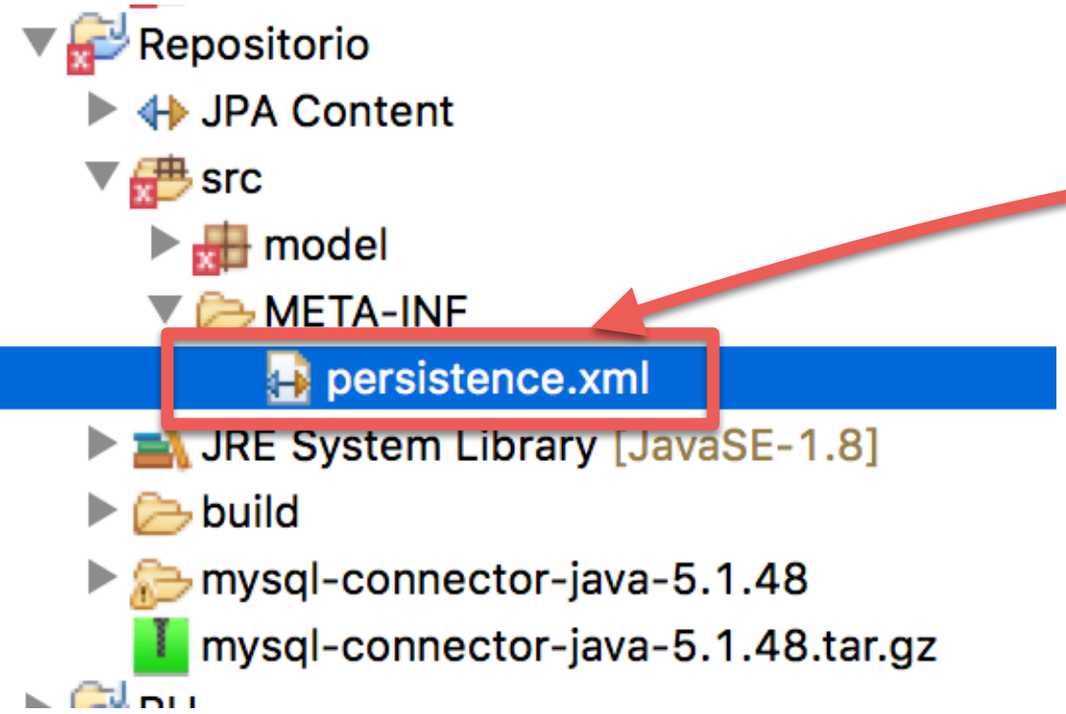
JPA


javax.persistence
EntityManagerFactory
EntityTransaction
1
*
EntityManager
*
Query
Persistence
*
Entity

JPA


operador
new
New
persist,
merge
Managed
commit,
flush
Banco
de
Dados
find,
query
close,
clear,
detach
merge
persist
remove
Detached
Removed

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
    <persistence-unit name="Repositorio" transaction-type="RESOURCE_LOCAL">
        <class>model.Account</class>
        <class>model.AccountRole</class>
        <class>model.AffiliateSpecialDiscount</class>
        <class>model.Affiliate</class>
        <class>model.AffiliatesLink</class>
        <class>model.Building</class>
        <class>model.BuildingContact</class>
        <class>model.BuildingEstate</class>
        <class>model.BuildingImage</class>
        <class>model.Chat</class>
        <properties>
            <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/database"/>
            <property name="javax.persistence.jdbc.user" value="root"/>
        </properties>
    </persistence-unit>
</persistence>
```




```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/p
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence htt
    <persistence-unit name="Repositorio" transaction-type="RESO
        <class>model.Account</class>
        <class>model.AccountRole</class>
        <class>model.AffiliateSpecialDiscount</class>
        <class>model.Affiliate</class>
        <class>model.AffiliatesLink</class>
        <class>model.Building</class>
        <class>model.BuildingContact</class>
        <class>model.BuildingEstate</class>
        <class>model.BuildingImage</class>
        <class>model.Chat</class>
        <properties>
            <property name="javax.persistence.jdbc.driver" va
            <property name="javax.persistence.jdbc.url" value=
            <property name="javax.persistence.jdbc.user" valu
        </properties>
    </persistence-unit>
</persistence>
```

```java
@Entity
public class Fruta {

    @Id
    @GeneratedValue
    private int id;

    @Version
    private int version;

    private String nome;
```

- **@Entity**: entidade a ser gerenciada no banco (Tabela)
- **@Id**: campo ID de uma tabela
- **@GeneratedValue**: indica que o campo é gerado automaticamente
- **@Version**: campo utilizando internamente pelo JPA

```
/**
 * @author Cassio Seffrin
 */
@Configuration
@EnableJpaRepositories(basePackages = { "com.cassio.spring.social.signinmvc.user.repository"})
@EnableTransactionManagement
public class PersistenceContext {

	private static final String[] PROPERTY_PACKAGES_TO_SCAN = {"com.cassio.dao.model" };

	protected static final String PROPERTY_NAME_DATABASE_DRIVER = "db.driver";
	protected static final String PROPERTY_NAME_DATABASE_PASSWORD = "db.password";
	protected static final String PROPERTY_NAME_DATABASE_URL = "db.url";
	protected static final String PROPERTY_NAME_DATABASE_USERNAME = "db.username";
	private static final String PROPERTY_NAME_HIBERNATE_DIALECT = "hibernate.dialect";
	private static final String PROPERTY_NAME_HIBERNATE_FORMAT_SQL = "hibernate.format_sql";
	private static final String PROPERTY_NAME_HIBERNATE_HBM2DDL_AUTO = "hibernate.hbm2ddl.auto";
	private static final String PROPERTY_NAME_HIBERNATE_NAMING_STRATEGY = "hibernate.ejb.naming_strategy";

	private static final String PROPERTY_NAME_HIBERNATE_SHOW_SQL = "hibernate.show_sql";

	@Resource
	private Environment env;

	@Bean
	public DataSource dataSource() {
		BoneCPDataSource dataSource = new BoneCPDataSource();
		dataSource.setDriverClass(env.getRequiredProperty(PROPERTY_NAME_DATABASE_DRIVER));
		dataSource.setJdbcUrl(env.getRequiredProperty(PROPERTY_NAME_DATABASE_URL));
		dataSource.setUsername(env.getRequiredProperty(PROPERTY_NAME_DATABASE_USERNAME));
		dataSource.setPassword(env.getRequiredProperty(PROPERTY_NAME_DATABASE_PASSWORD));
		return dataSource;
	}
	@Bean
	public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
		LocalContainerEntityManagerFactoryBean entityManagerFactoryBean = new LocalContainerEntityManagerFactoryBean();
		entityManagerFactoryBean.setDataSource(dataSource());
		entityManagerFactoryBean.setJpaVendorAdapter(new HibernateJpaVendorAdapter());
		entityManagerFactoryBean.setPackagesToScan(PROPERTY_PACKAGES_TO_SCAN);
		Properties jpaProperties = new Properties();
		jpaProperties.put(PROPERTY_NAME_HIBERNATE_DIALECT, env.getRequiredProperty(PROPERTY_NAME_HIBERNATE_DIALECT));
		jpaProperties.put(PROPERTY_NAME_HIBERNATE_FORMAT_SQL, env.getRequiredProperty(PROPERTY_NAME_HIBERNATE_FORMAT_SQL));
		jpaProperties.put(PROPERTY_NAME_HIBERNATE_HBM2DDL_AUTO, env.getRequiredProperty(PROPERTY_NAME_HIBERNATE_HBM2DDL_AUTO));
		jpaProperties.put(PROPERTY_NAME_HIBERNATE_NAMING_STRATEGY,env.getRequiredProperty(PROPERTY_NAME_HIBERNATE_NAMING_STRATEGY));
		jpaProperties.put(PROPERTY_NAME_HIBERNATE_SHOW_SQL, env.getRequiredProperty(PROPERTY_NAME_HIBERNATE_SHOW_SQL));
		entityManagerFactoryBean.setJpaProperties(jpaProperties);

		return entityManagerFactoryBean;
	}
}
```

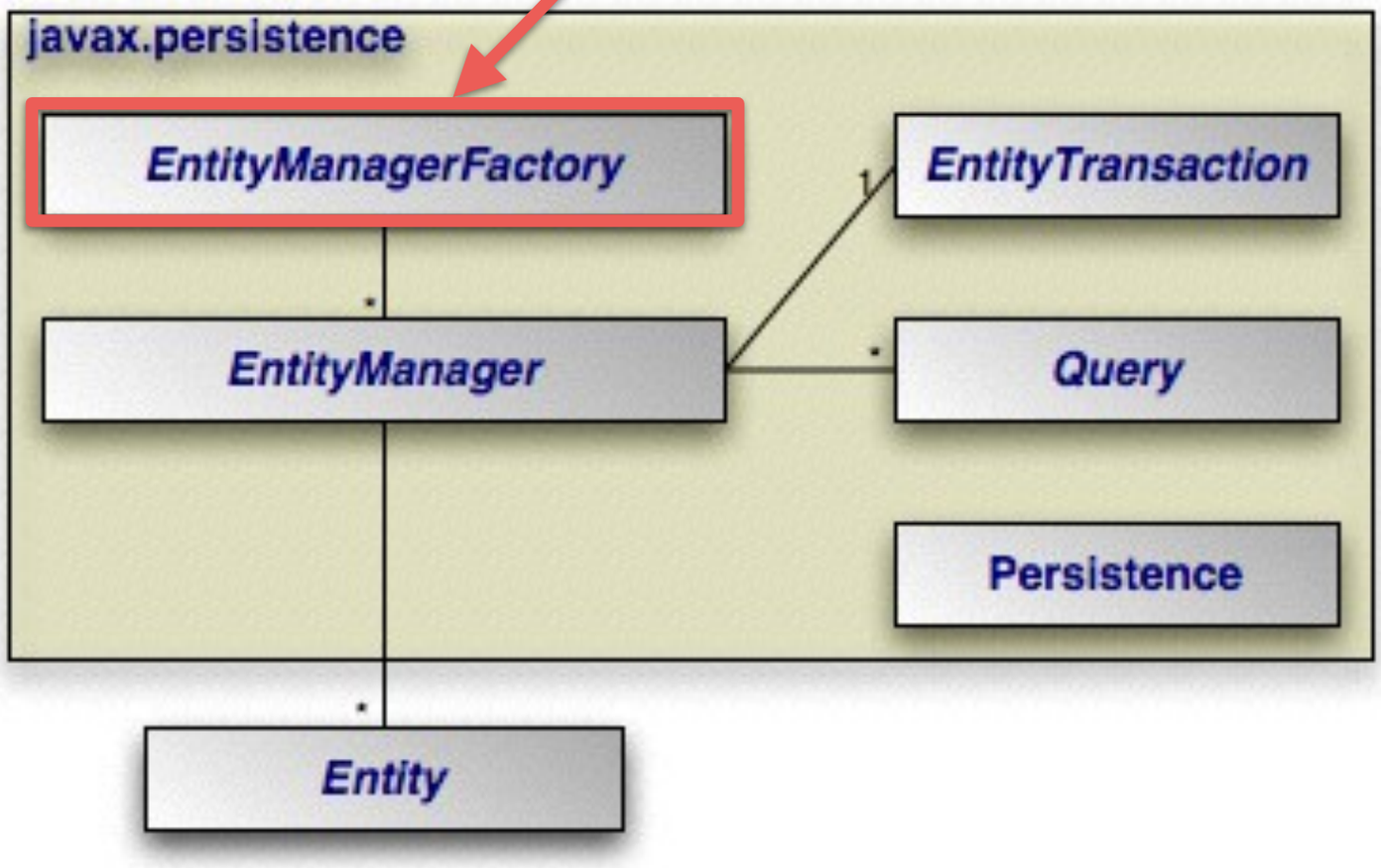
```
EntityManager em = factory.createEntityManager();
Fruta fruta = (Fruta) em.createQuery("SELECT t FROM Fruta t")
                        .getSingleResult();
em.getTransaction().begin();
em.remove(fruta);
em.getTransaction().commit();
em.close();
```


```
@Entity
public class Fruta {

    @Id
    @GeneratedValue
    private int id;

    @Version
    private int version;

    private String nome;
```

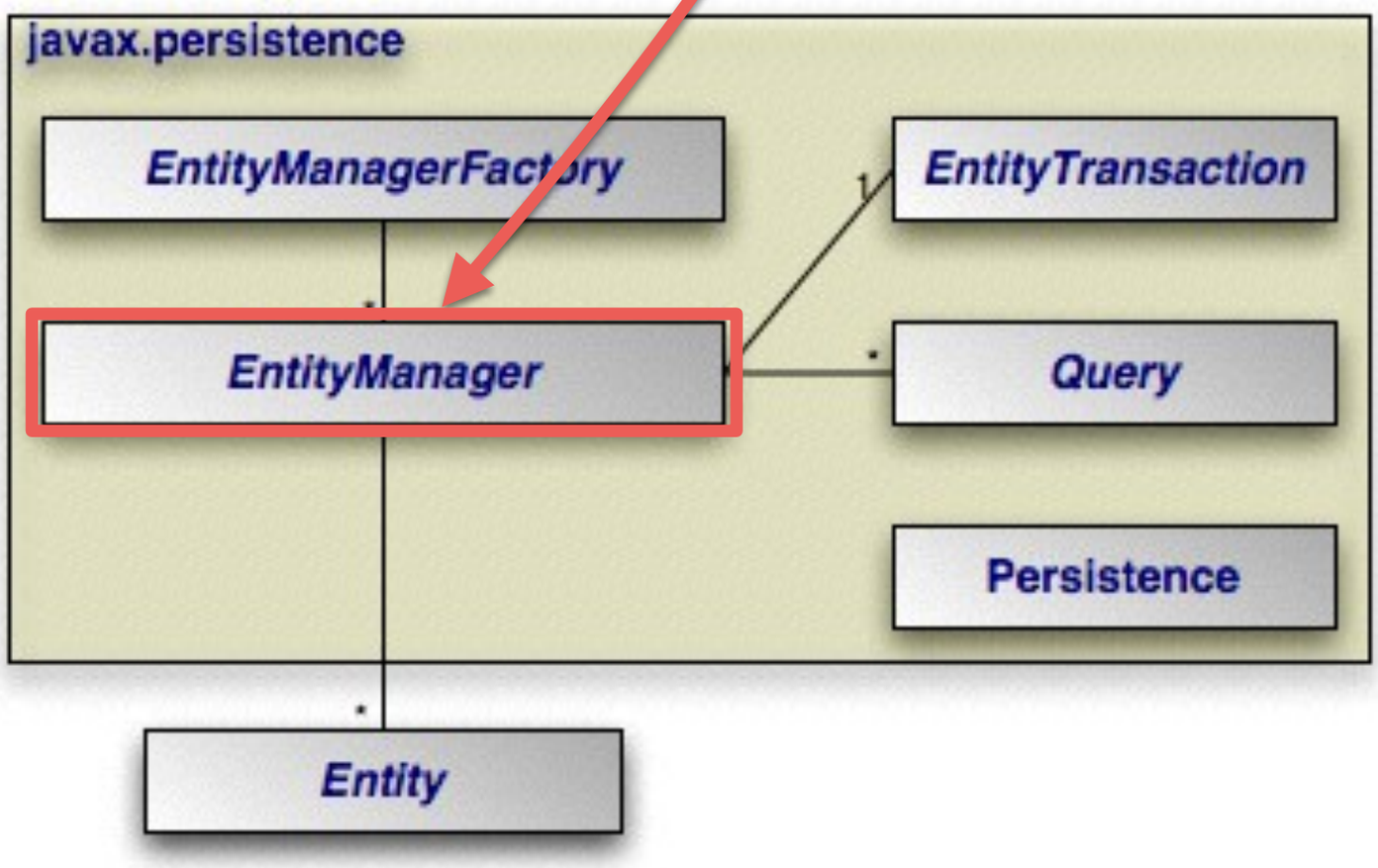
```java
EntityManager em = factory.createEntityManager();
Fruta fruta = (Fruta) em.createQuery("SELECT t FROM Fruta t WHERE t.nome=:nome")
                        .setParameter("nome", "abacaxi")
                        .getSingleResult();
fruta.setNome("amora");
em.getTransaction().begin();
em.persist(fruta);
em.getTransaction().commit();
em.close();
```


```java
@Entity
public class Fruta {

    @Id
    @GeneratedValue
    private int id;

    @Version
    private int version;

    private String nome;
```

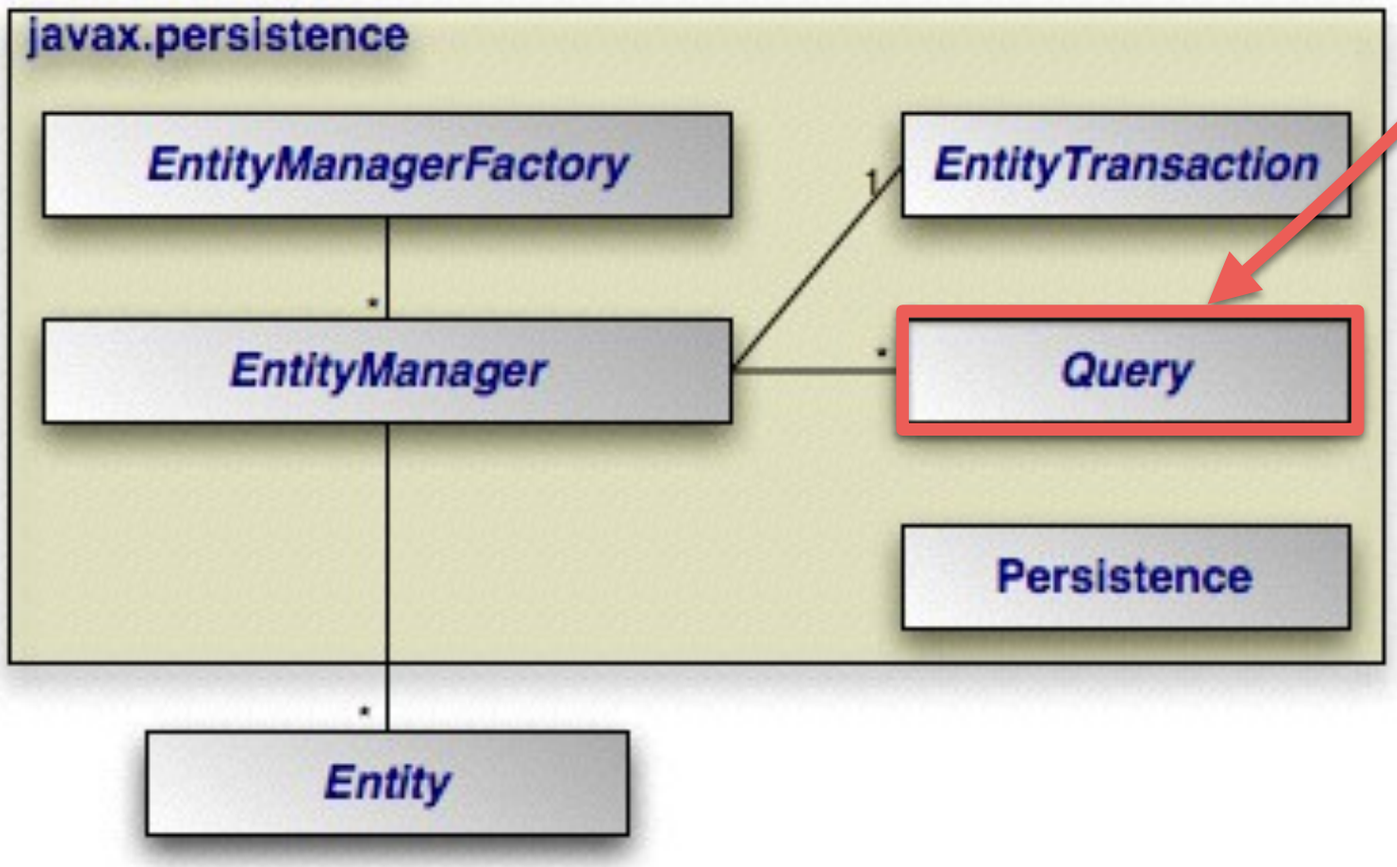
```
EntityManager em = factory.createEntityManager();
Fruta fruta = (Fruta) em.createQuery("SELECT t FROM Fruta t WHERE t.nome=:nome")
                        .setParameter("nome", "abacaxi")
                        .getSingleResult();
fruta.setNome("amora");
em.getTransaction().begin();
em.persist(fruta);
em.getTransaction().commit();
em.close();
```



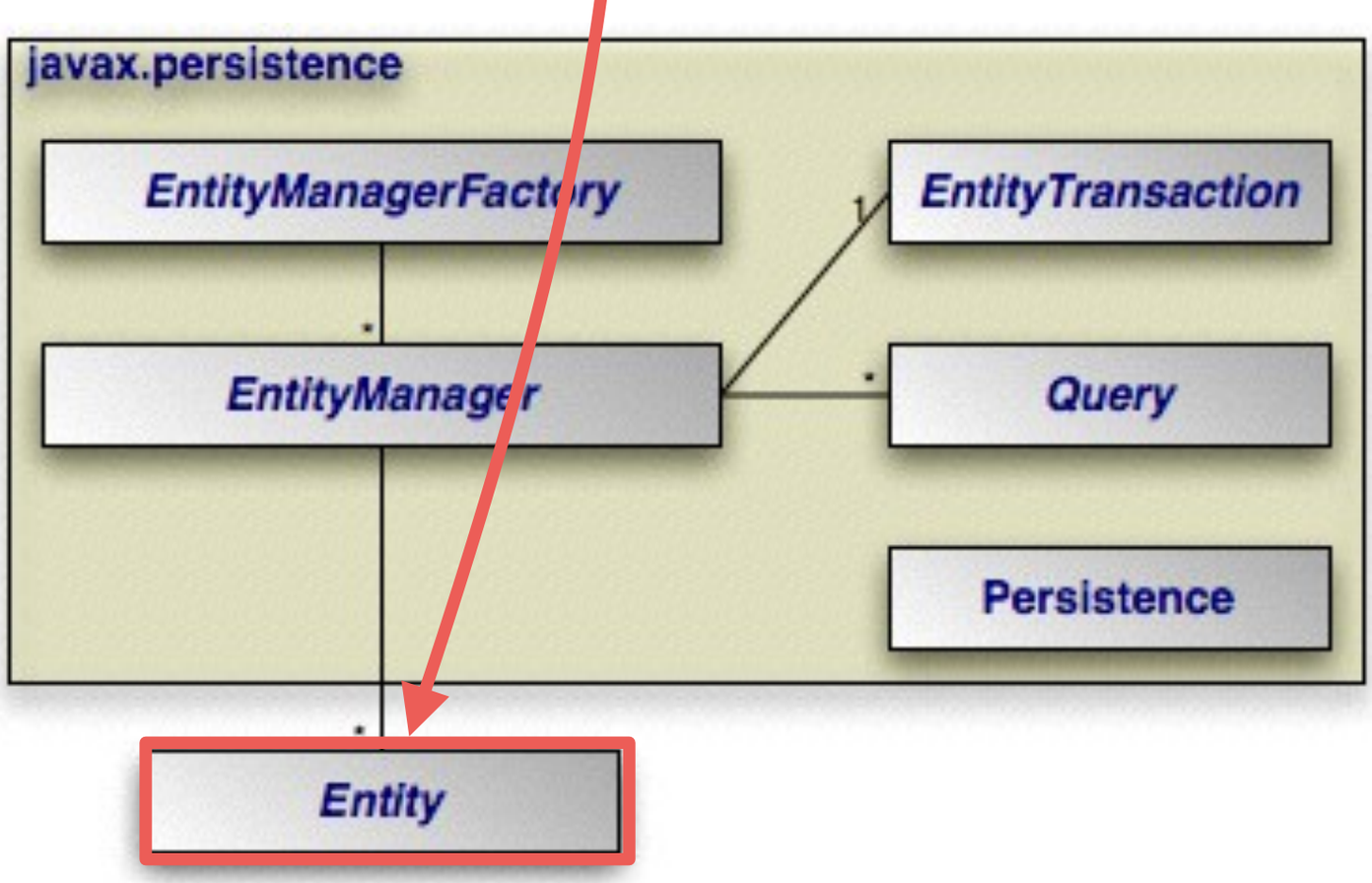
```
EntityManager em = factory.createEntityManager();
Fruta fruta = (Fruta) em.createQuery("SELECT t FROM Fruta t WHERE t.nome=:nome")
                        .setParameter("nome", "abacaxi")
                        .getSingleResult();
fruta.setNome("amora");
em.getTransaction().begin();
em.persist(fruta);
em.getTransaction().commit();
em.close();
```



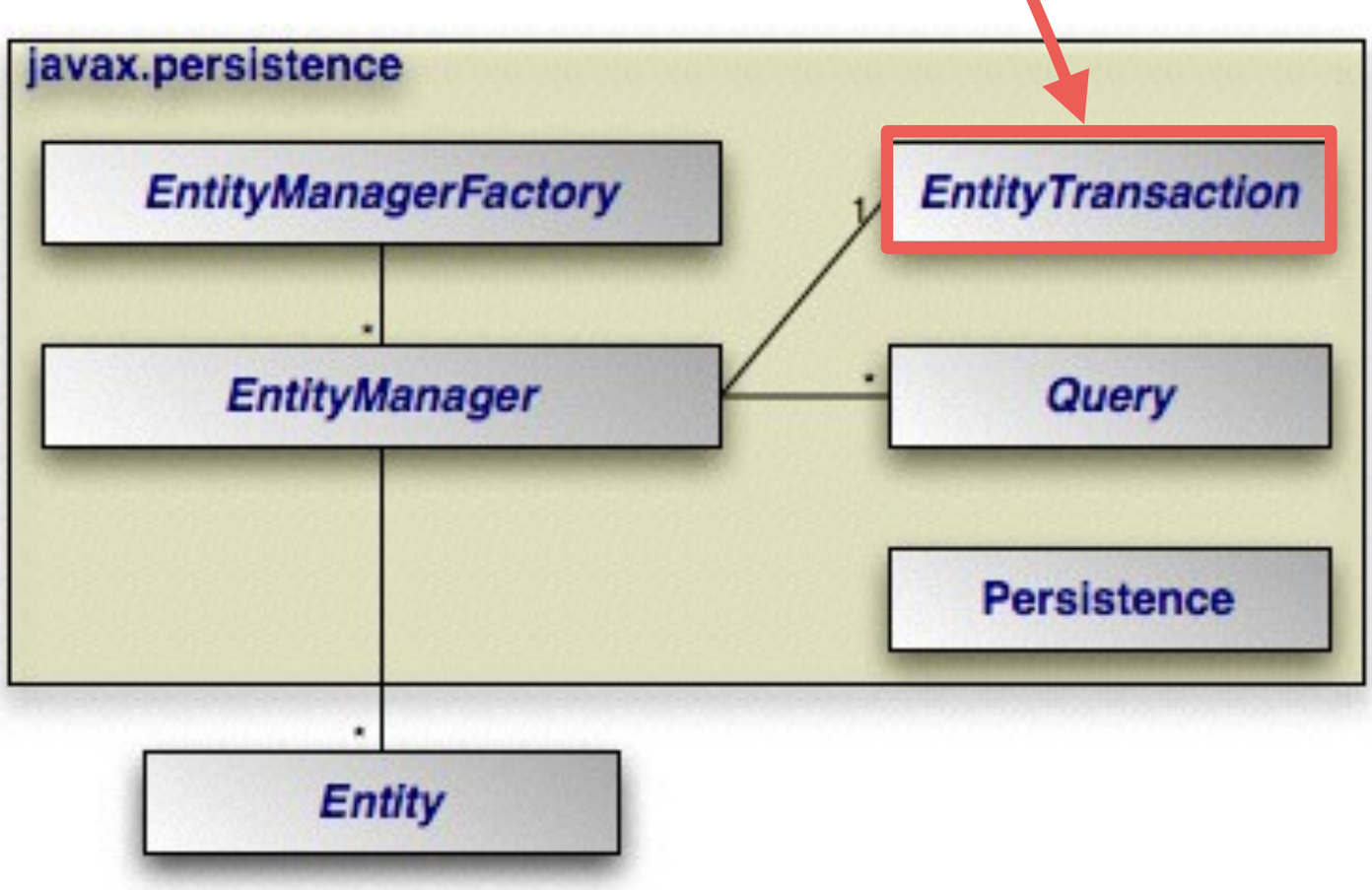
```
EntityManager em = factory.createEntityManager();
Fruta fruta = (Fruta) em.createQuery("SELECT t FROM Fruta t WHERE t.nome=:nome")
                      .setParameter("nome", "abacaxi")
                      .getSingleResult();
fruta.setNome("amora");
em.getTransaction().begin();
em.persist(fruta);
em.getTransaction().commit();
em.close();
```



```
EntityManager em = factory.createEntityManager();
Fruta fruta = (Fruta) em.createQuery("SELECT t FROM Fruta t WHERE t.nome=:nome")
                        .setParameter("nome", "abacaxi")
                        .getSingleResult();
fruta.setNome("amora");
em.getTransaction().begin();
em.persist(fruta);
em.getTransaction().commit();
em.close();
```



```
EntityManager em = factory.createEntityManager();
Fruta fruta = (Fruta) em.createQuery("SELECT t FROM Fruta t WHERE t.nome=:nome")
                        .setParameter("nome", "abacaxi")
                        .getSingleResult();
fruta.setNome("amora");
em.getTransaction().begin();
em.persist(fruta);
em.getTransaction().commit();
em.close();
```



JPA

```
// cria o porsche no estado new
Marca porsche = new Marca("Porsche");

// passa o porsche para estado gerenciado
em.persist(porsche);

// cria a ferrari no estado novo
Marca ferrari = new Marca(1, "Ferrari");

//devolve a instância gerenciada
Marca outraFerrari = em.merge(ferrari);

// vai imprimir false
System.out.println(ferrari == outraFerrari);
```

JPA Annotations

JPA Annotations

Atributos ignorados

- Utilizados para atributos calculados, ou seja, que não serão persistidos
- Isso é feito com atributos transientes:

```
@Transient
private double precoItem;
```

Ou

```
private transient double precoItem;
```

JPA Annotations

Alterando o nome dos campos

```
@Entity
@Table(name=“T_AUTOMOVEIS")
   public class Automovel {

        @Id  @GeneratedValue
        @Column(name="COD")
        private Integer id;

        @Column(name="DESC", nullable=false, length=100)
        private String descricao;

        @Temporal(TemporalType.TIMESTAMP)
        @Column(name="DT_CADASTRO", nullable=false,
updatable=false)
        private LocalDate dataCadastro;

        ...

}
```

JPA Annotations

- Atributos do tipo **Date** ou **Calendar**
- @Temporal:
  - **DATE**: somente a data, sem informação de hora;
  - **TIME**: somente informação de hora, sem data;
  - **TIMESTAMP**: valor padrão, que considera tanto data quanto hora

JPA Annotations

- Anotação: @ManyToOne

```
@Entity
public class Modelo
    {   @Id
        @GeneratedValue
        private Long id;

        @ManyToOne
        private Marca marca;
}
```

JPA Annotations

- Anotação: @OneToMany

```
@Entity
public class
    Marca{  @Id
    @GeneratedValue
    private Long id;

    @OneToMany
    private List<Modelo> modelos;
}
```

JPA Annotations

Permite acessar os objetos de associação através das duas pontas do relacionamento

```
public class Marca
   {  @Id
   @GeneratedValue
   private Integer id;
   ...
   @OneToMany(mappedBy="montadora")
   private List<Modelo> modelos;
   ...
}
```

```
public class Modelo {
  @Id  @GeneratedValue
   private Integer id;
   ...
   @ManyToOne
   private Marca montadora;
   ...
}
```

JPA Annotations

- **Lazy Loading** (padrão)
  - Carrega o(s) objeto(s) do relacionamento apenas quando necessário
- **Eager Loading**
  - Força o carregamento do(s) objeto(s) do relacionamento sempre

```
@Entity
public class Marca {
  @Id @GeneratedValue
  private Integer id;

  @OneToMany(fetch=FetchType.EAGER)
  private List<Modelo> modelos;
}
```

JPA Annotations

- Permite associar uma entidade com uma lista de algum tipo básico, como a String

```
@Entity
public class Automovel {
  @Id @GeneratedValue
  private Integer id;

  @ElementCollection @Column(length=20)
  private List<String> tags;

  ...
}
```

JPA Annotations

- Anotação: @ManyToMany

```
@Entity
public class Automovel {
   ...
   @ManyToMany
   private List<Opcional> opcionais;
   ...
}

@Entity
public class Opcional {
   ...
   @ManyToMany(mappedBy="opcionais")
   private List<Automovel> automoveis;
   ...
}
```

JPQL

JPQL

- ***Java Persistence Query Language***
  - JPQL é uma linguagem de consulta, assim como a SQL, porém orientada a objetos

- SQL (Structured Query Language:
  - `select * from T_Automoveis where ano_modelo >= 2019`
- JPQL:
  - `select a from Automovel a where a.anoModelo >= 2019`

JPQL

- Suporte aos operadores básicos do SQL:
  - =, >, >=, <, <=, <>, NOT, BETWEEN, LIKE, IN, IS NULL, IS EMPTY, MEMBER [OF], EXISTS
- Funções:
  - CONCAT(String, String...)
  - SUBSTRING(String, int start [, int length])
  - TRIM([[LEADING | TRAILING | BOTH] [char] FROM] String)
  - LOWER(String); UPPER(String); LENGTH(String)
  - LOCATE(String original, String substring [, int start])
  - ABS(int), SQRT(int); MOD(int, int)
  - CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP

JPQL

- SQL:
  - `select * from Automovel auto left outer join Modelo modelo on auto.modelo_id = modelo.id left outer join Marca marca on modelo.marca_id = marca.id where marca.nome = 'Ferrari'`
- JPQL:
  - `select a from Automovel a where a.modelo.marca.nome = 'Ferrari'`

JPQL

```
String jpql = "select a from Automovel a where
   a.modelo.marca = :marca";
Marca marca = ...
Query query = entityManager.createQuery(jpql);
query.setParameter("marca", marca);

List<Automovel> automoveis = query.getResultList();
```

JPQL

- Funções de agregações:
  - AVG(property)
  - MAX(property)
  - MIN(property)
  - SUM(property)
  - COUNT(property)

- Exemplo:

```
String jpql = "select AVG(a.anoModelo)
  from Automovel a";
Query query =
  entityManager.createQuery(jpql);
Double media = query.getSingleResult();
```

JPQL

- Recurso chamado de **NamedQuery**
- Exemplo:

```
@NamedQuery(name="Automovel.listarTodos",
  query="select a from Automovel a")
@Entity
public class Automovel {
  // atributos e métodos
}
```

- Como usar:

```
Query query =
  em.createNamedQuery("Automovel.listarTodos");
List<Automovel> automoveis =
  query.getResultList();
```

JPQL

- Classe

```
@NamedQuery(name="Usuario.findByUsuario",
  query="SELECT u FROM Usuario u WHERE u.usuario
  = :usuario AND u.senha = :senha AND u.ativo = true")
@Entity
public class Usuario {
  ...
}
```

- Chamada

```
login = (Usuario)
em.createNamedQuery("Usuario.findByUsuario")
.setParameter("usuario", usuario)
.setParameter("senha", senha)
.getSingleResult();
```

Testes / Linter

EclEmma

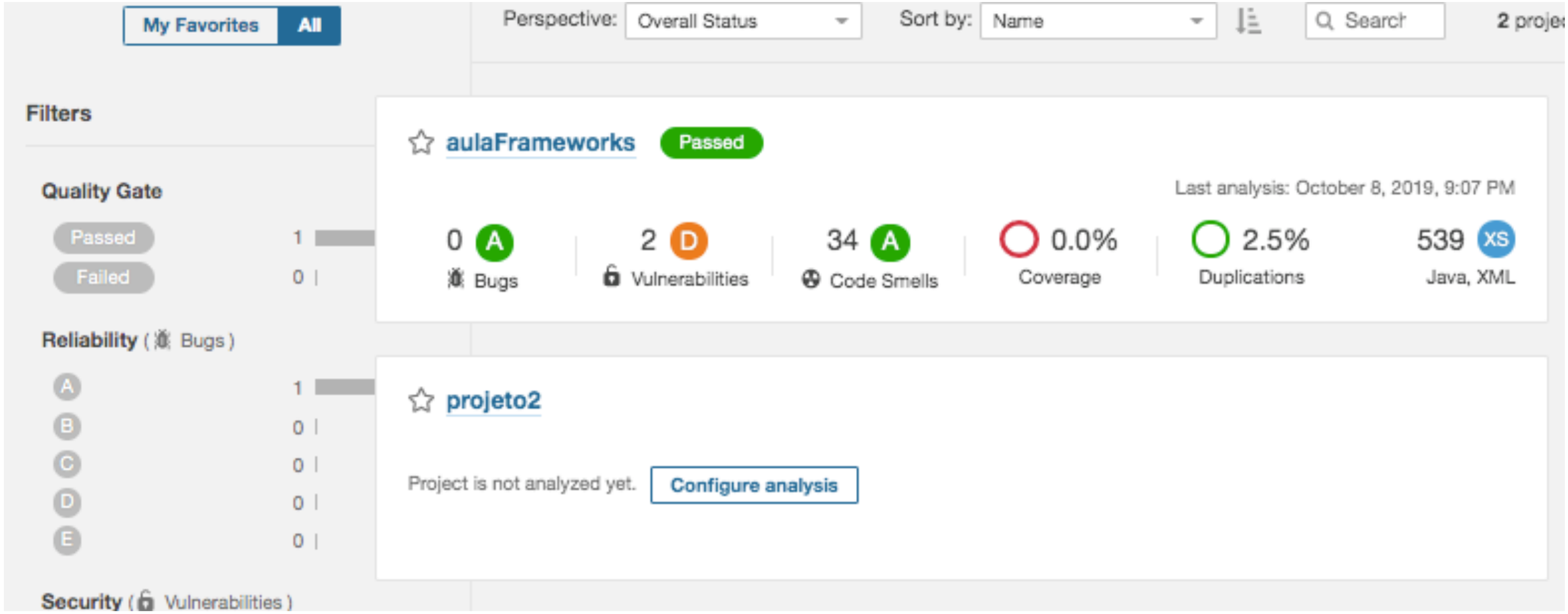
sonarqube linter

http://18.231.171.213:9000/

```
mvn sonar:sonar \
  -Dsonar.projectKey=projeto2 \
  -Dsonar.host.url=http://18.231.171.213:9000 \
  -Dsonar.login=aa70d64ad1911ff33d6905f9d720dab2a3ae5e7b
```

Sonar Linter

http://18.231.171.213:9000/



Testes

Não há como testar os repositórios JPA do Spring Data razoavelmente por uma razão simples: é muito complicado mocar (mock) todas as partes da API JPA que invocamos para inicializar os repositórios. Os testes de unidade não fazem muito sentido aqui, pois você normalmente não está escrevendo nenhum código de implementação.
Para este cenário um teste de integração é mais apropriado.
Detalhes

Realizamos várias validações e configurações iniciais para garantir que você possa inicializar apenas um aplicativo que não tenha consultas derivadas inválidas etc.

Criamos e armazenamos em cache as instâncias CriteriaQuery para consultas derivadas para garantir que os métodos de consulta não contenham erros de digitação. Isso requer o trabalho com a API de critérios, bem como com o meta.model.
Verificamos as consultas definidas manualmente solicitando ao EntityManager que crie uma instância de consulta para elas (o que efetivamente aciona a validação da sintaxe da consulta).
Inspecionamos o Metamodelo em busca de metadados sobre os tipos de domínio manipulados para preparar novas verificações, etc.
Tudo o que você provavelmente adiaria em um repositório escrito à mão que poderia causar a quebra do aplicativo em tempo de execução (devido a consultas inválidas etc.).

Testes de Integração

Se você pensar bem, não há código que você escreva para seus repositórios, então não há necessidade de escrever testes unitários neste caso. Simplesmente não é necessário, pois você pode confiar em nossa base de testes para detectar bugs básicos (se você ainda encontrar um deles, sinta-se à vontade para pedir uma multa). No entanto, há definitivamente a necessidade de testes de integração para testar dois aspectos da sua camada de persistência, pois são os aspectos relacionados ao seu domínio:

Mapeamentos de entidade
semântica de consulta (a sintaxe é verificada em cada tentativa de inicialização) de qualquer maneira).
Testes de integração
Isso geralmente é feito usando um banco de dados na memória e casos de teste que inicializam um Spring ApplicationContext geralmente através da estrutura de contexto de teste (como você já faz), preenchem previamente o banco de dados (inserindo instâncias de objeto por meio do EntityManager ou repo, ou via um arquivo SQL simples) e, em seguida, execute os métodos de consulta para verificar o resultado deles.

Testando implementações personalizadas
Partes de implementação personalizadas do repositório são gravadas de uma maneira que eles não precisam saber sobre o Spring Data JPA. Eles são simples Spring beans que geram um EntityManager injetado. É claro que você pode tentar zombar das interações com ele, mas, para ser sincero, testar a unidade JPA não foi uma experiência muito agradável para nós, além de funcionar com muitas indiretas (EntityManager -> CriteriaBuilder, CriteriaQuery etc. .) para que você acabe com zombarias retornando zombarias e assim por diante.

Testes de Integração

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
@Transactional
@TransactionConfiguration(defaultRollback = true)
public class UserRepositoryTest {
@Configuration
@EnableJpaRepositories(basePackages = "com.anything.repository")
static class TestConfiguration {
    @Bean
    public EntityManagerFactory entityManagerFactory() {
        return mock(EntityManagerFactory.class);
    }
}
```

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class AlunoRepositoryIntegrationTest {

    @Autowired
    private AlunoRepository alunoRepository;

    @Autowired
    private EnderecoRepository enderecoRepository;

    @Test
    public void salvarAluno() throws Exception {
        Aluno aluno = new Aluno();
      }
}
```

Testes referencias

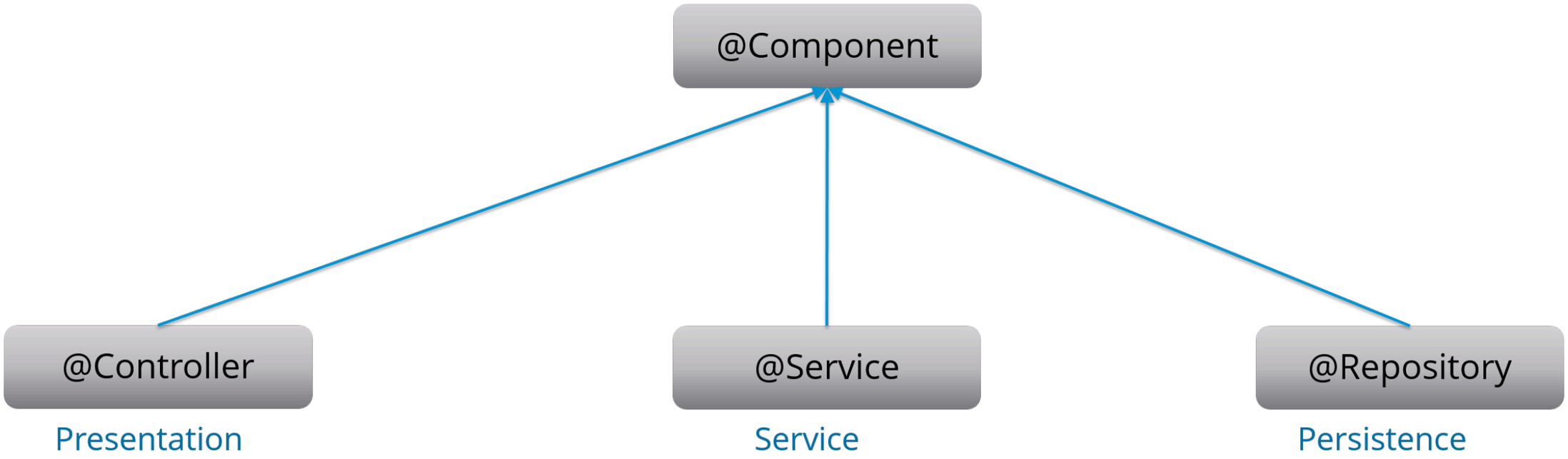
https://www.luckyryan.com/2013/06/25/integration-testing-spring-data-jpa/

Spring MVC

DI - injeção de dependência

Na Injeção de Dependência, você não precisa criar seus objetos, mas descrever como eles devem ser criados. Você não conecta seus componentes e serviços diretamente no código, mas descreve quais serviços são necessários e quais componentes no arquivo de configuração. O contêiner de IoC os conectará.

Spring MVC - Anotações



Spring MVC - Anotações

Nas aplicações mais comuns, temos camadas distintas, como acesso a dados, apresentação, serviço/negócios e controladores.

E, em cada camada, temos vários Java beans. Para detectá-los automaticamente, o Spring usa anotações de verificação de caminho de classe. Em seguida, registra cada bean no ApplicationContext.

As principais anotações do Spring são:

@Component é um estereótipo genérico para qualquer componente gerenciado pelo Spring;

@Service são classes na camada de serviço (lógica de negócio);
@Repository anota classes na camada de persistência, que atuará como um repositório de banco de dados;
@Controller é usado para classes de controle do MVC.

Estas anotações são usados para diferentes classificações. Quando anotamos uma classe para detecção automática, devemos usar o respectivo estereótipo.

Spring MVC - Anotações

É importante escolhermos a anotação com base em suas convenções de camada:

@Component
Podemos usar o @Component para marcar os beans como componentes gerenciados do Spring. Em geral o Spring apenas registra beans com @Component e não procura @Service, @Repository e @Controller.

Eles são registrados no ApplicationContext quando são anotados com @Component

@Service, @Repository e @Controller são especializações de @Component. Eles são tecnicamente iguais, mas os usamos para diferentes propósitos.

O trabalho do @Repository (especialização do Component) é capturar exceções específicas de persistência e repeti-las novamente como uma das exceções não verificadas e unificadas do Spring e fornece PersistenceExceptionTranslationPostProcessor. Além de importar os DAOs para o contêiner através de DI (Dependency Injection.

…

Spring MVC - Anotações

Anotamos um bean com **@Service** (especialização do Component) para indicar que ele está mantendo a lógica de negócios. Não fornece nenhum comportamento adicional sobre a anotação @Component, mas é uma boa ideia usar @Service invés @Component, pois especifica melhor a intenção da classe. Além disso, o suporte a ferramentas e o comportamento adicional podem contar com isso no futuro.

A anotação **@Controller** (especialização do Component) marca uma classe como um controlador Spring Web MVC. Também é uma especialização @Component, portanto, os beans marcados com ele são importados automaticamente para o contêiner DI. Quando adicionamos a anotação @Controller a uma classe, podemos usar outras anotações derivadas, ou seja, @GetMapping, @PostMapping etc.

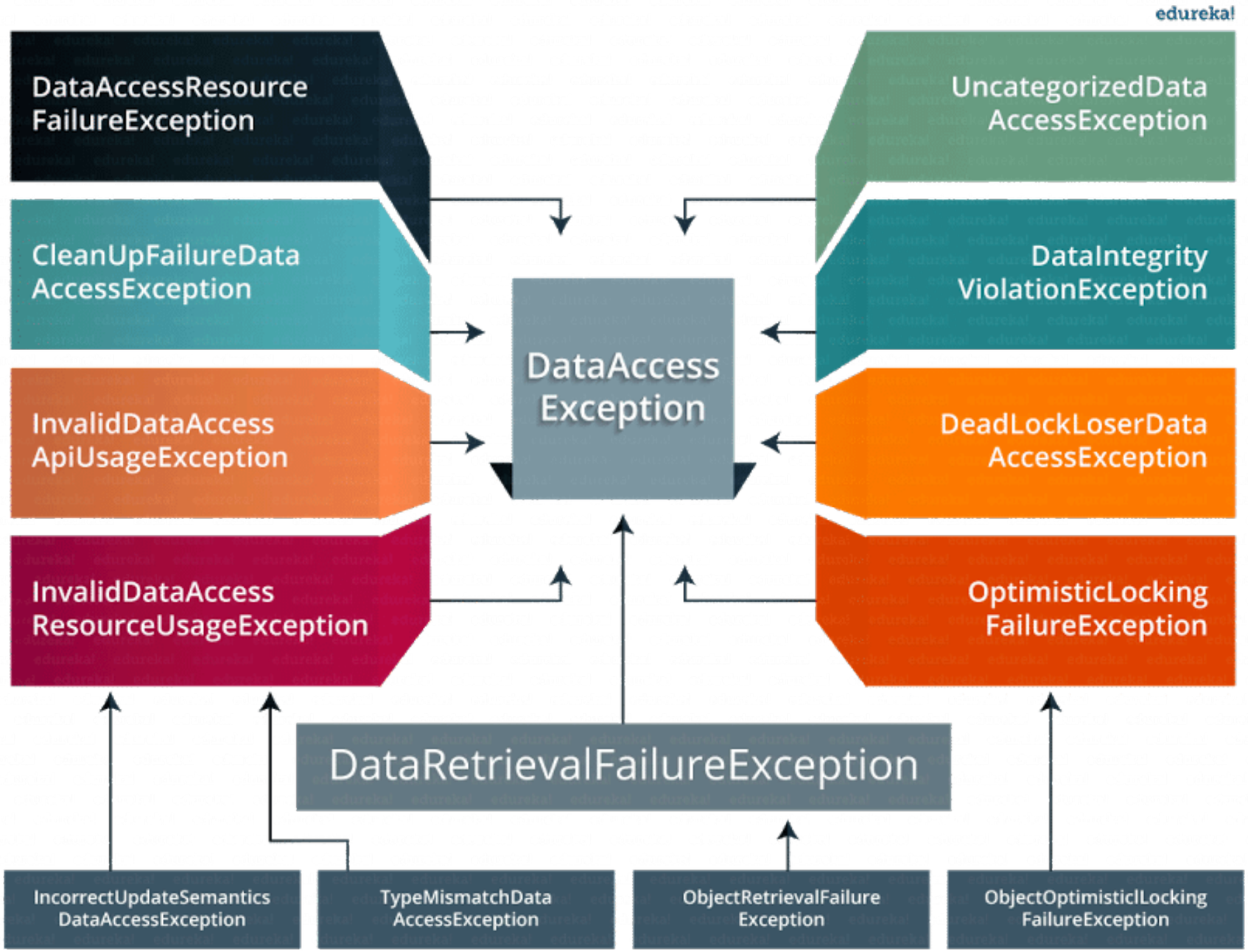
Spring MVC - Anotações

@Configuration
A anotação Spring @Configuration auxilia na configuração baseada em anotações Spring, indicando que uma classe declara um ou mais métodos @Bean e podem ser processados pelo contêiner Spring para gerar definições de bean e solicitações de serviço para esses beans em tempo de execução.
Ex:

```
@Configuration
@ComponentScan(basePackages = “com.cassio.edu”)
class MatriculaConfig {
    @Bean
    Matricula matricula() {
        retorn new Matricula ();
    }
}
```

A anotação **@Bean** que possui comportamentos destinos das outras:
O @Bean é usado para declarar explicitamente um único bean, em vez de permitir que o Spring faça isso automaticamente para nós.
Outra grande diferença é que o @Bean é uma anotação no nível do método e não da classe e, por padrão, o nome do método serve como o nome do bean.

Spring Data - Excessões



Spring MVC

```
@Slf4j
public abstract class RESTController<T, ID extends Serializable> {

    private static final String SUCCESSO = "successo";

    private CrudRepository<T, ID> repo;

    public RESTController(CrudRepository<T, ID> repo) {
        this.repo = repo;
    }

    @GetMapping
    public @ResponseBody Collection<T> todos() {
        Iterable<T> todos = this.repo.findAll();
        return Uteis.converterInterabelParaList(todos);
    }


    @PostMapping(consumes={MediaType.APPLICATION_JSON_VALUE})
    public @ResponseBody Map<String, Object> salvar(@RequestBody T json) {
        log.info("criado() com body {} do tipo {}", json, json.getClass());
        T objetoCriado = this.repo.save(json);
        Map<String, Object> m =  new HashMap<>();
        m.put(SUCCESSO, true);
        m.put("criado", objetoCriado);
        return m;
    }
}
```

Spring MVC

```
@Controller
@RequestMapping("/curso")
public class CursoController extends RESTController<Curso,
Integer> {

    @Autowired
    public CursoController(CrudRepository<Curso, Integer> repo) {
        super(repo);
    }
}
```