Pivotal.

JPA with Spring

Object Relational Mapping with Spring
and Java Persistence API

Objectives

- After completing this lesson, you should be able to:
  - Explain the basic concepts of JPA
  - Configure JPA using Spring
  - Implement a JPA DAO
  - Explain how JPA Integration is implemented by Spring

Agenda

- **Introduction to JPA**

  - **General Concepts**

  - **Mapping**

  - **Querying**

- Configuring JPA in Spring

- Optional and Advanced Topics

Pivotal.

Introduction to JPA

- The Java Persistence API (JPA) is designed for operating on domain objects
  - Defined as POJO entities
  - No special interface required

- A common API for object-relational mapping
  - Derived from experience with existing products
    - JBoss Hibernate
    - Oracle TopLink (now EclipseLink)

About JPA

- Java Persistence API
  - Current version: 2.2 – released mid-2017
- Configuration
  - Persistence Unit
- Key Features
  - Entity Manager
  - Entity Manager Factory
  - Persistence Context

Pivotal

JPA Configuration
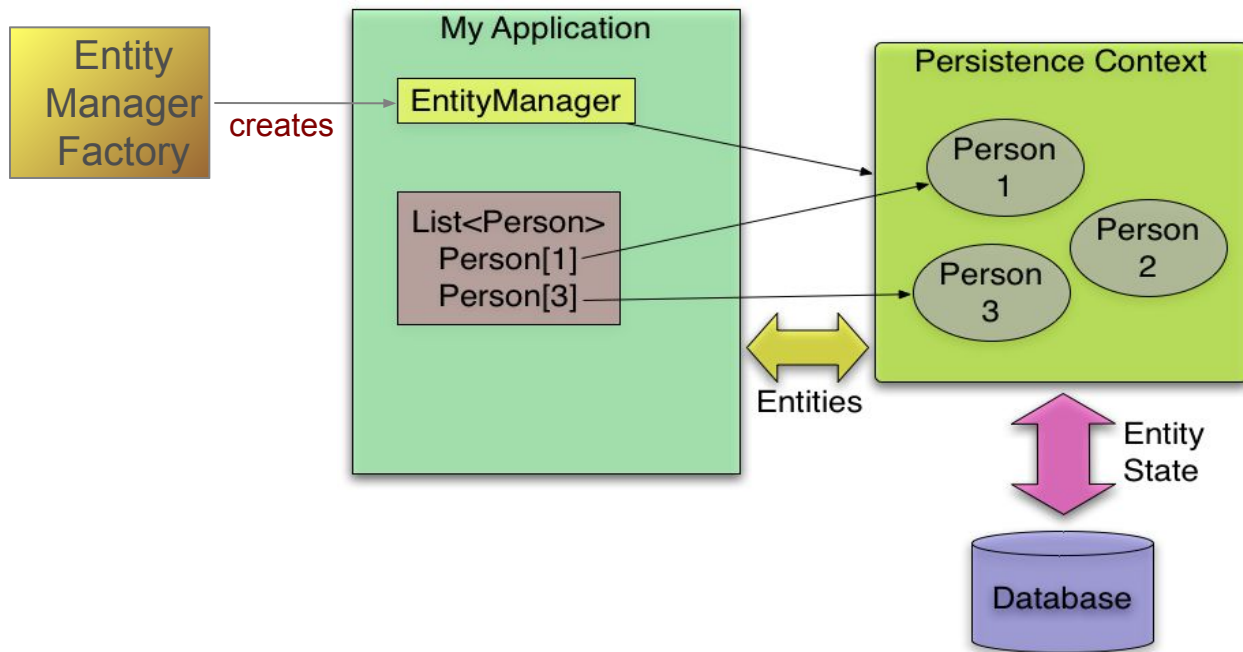
- **Persistence Unit**
  - Describes a group of persistent classes (entities)
  - Defines provider(s)
  - Defines transactional types (local vs JTA)
  - Multiple Units per application are allowed
  - Defined by the file: `persistence.xml`

JPA General Concepts

- **EntityManager**

  – Manages a unit of work and persistent objects therein: the *PersistenceContext*

  – Lifecycle often bound to a Transaction (usually container-managed)

- **EntityManagerFactory**

  – Thread-safe, shareable object that represents a single data source / persistence unit

  – Provides access to new application-managed EntityManagers

Pivotal

Persistence Context and EntityManager

Pivotal

The EntityManager API

|	
`persist(Object o)`	Adds the entity to the Persistence Context: *SQL: insert into table …*
`remove(Object o)`	Removes the entity from the Persistence Context: *SQL: delete from table …*

`find(Class entity,`
    `Object primaryKey)`
	Find by primary key:
	    SQL: select * from table where id = ?

`Query createQuery` `(String jpqlString)`	Create a JPQL query
`flush()`	Force changed entity state to be written to database immediately

Plus many other methods ...

Pivotal.

JPA Providers

- Several major implementations of JPA spec
  - Hibernate EntityManager
    - Used inside JBoss
  - EclipseLink (RI)
    - Used inside GlassFish
  - Apache OpenJPA
    - Used by Oracle WebLogic and IBM WebSphere
  - Data Nucleus
    - Used by Google App Engine
- **Can all be used without application server as well**
 - Independent part of EJB 3 spec

Pivotal.

Hibernate JPA

- ## Hibernate adds JPA support through an additional library
  - The *Hibernate EntityManager*
  - Hibernate *sessions* used behind JPA *interfaces*
  - Custom annotations for Hibernate specific extensions not covered by JPA
    - less important since JPA
      version 2

JPA

API

Hibernate

Implementation

Pivotal.
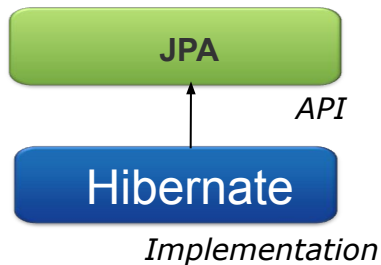
JPA Mapping

- JPA requires metadata for mapping classes/fields to database tables/columns
  - Usually provided as annotations
  - XML mappings also supported (orm.xml)
    - Intended for overrides only – not shown here
- JPA metadata relies on defaults
  - No need to provide metadata for the obvious
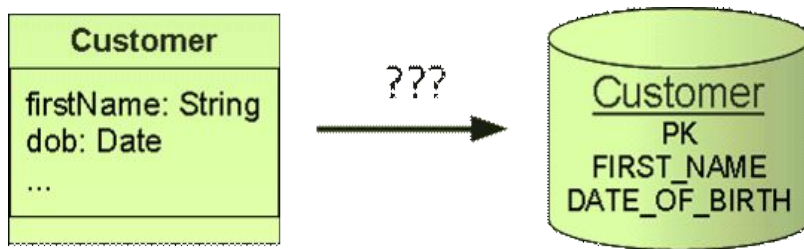
What can you Annotate?

- Classes
  - Applies to the entire class (such as table properties)
- Fields
  - Typically mapped to a column
  - By default, *all* treated as persistent
    - Mappings will be defaulted
    - Unless annotated with @Transient (non-persistent)
  - Accessed directly via Reflection
- Properties (getters)
  - Also mapped to a column
  - Annotate getters instead of fields

Pivotal.

Mapping Using Fields (Data-Members)

```java
@Entity
@Table(name= "T_CUSTOMER")
public class Customer {
    @Id
    @Column(name="cust_id")
    private Long id;

    @Column(name="first_name")
    private String firstName;

    @Transient
    private User currentUser;


    ...
```

Mark as an *entity*
   Optionally override *table name*

Mark *id-field*
   (primary key)

Optionally override *column names*

Not stored in database

Only *@Entity* and *@Id* are mandatory

Data members set *directly*
 - *using reflection*
 - *"field" access*
 - *no setters needed*

Pivotal.

Mapping Using Accessors (Properties)

Must place @Id on the *getter* method

Other annotations now also placed on *getter* methods

```java
@Entity @Table(name= "T_CUSTOMER")
public class Customer {
    private Long id;
    private String firstName;

    @Id
    @Column (name="cust_id")
    public Long getId()
        { return this.id; }

    @Column (name="first_name")
    public String getFirstName()
        { return this.firstName; }

    public void setFirstName(String fn)
        { this.firstName = fn; }
}
```

Pivotal.

Relationships

- Common relationship mappings supported
  - Single entities and entity collections both supported
  - Associations can be uni- or bi-directional

```java
@Entity
@Table(name= "T_CUSTOMER")
public class Customer {
    @Id
    @Column (name="cust_id")
    private Long id;

    @OneToMany
    @JoinColumn (name="cid")
    private Set<Address> addresses;
    …
```
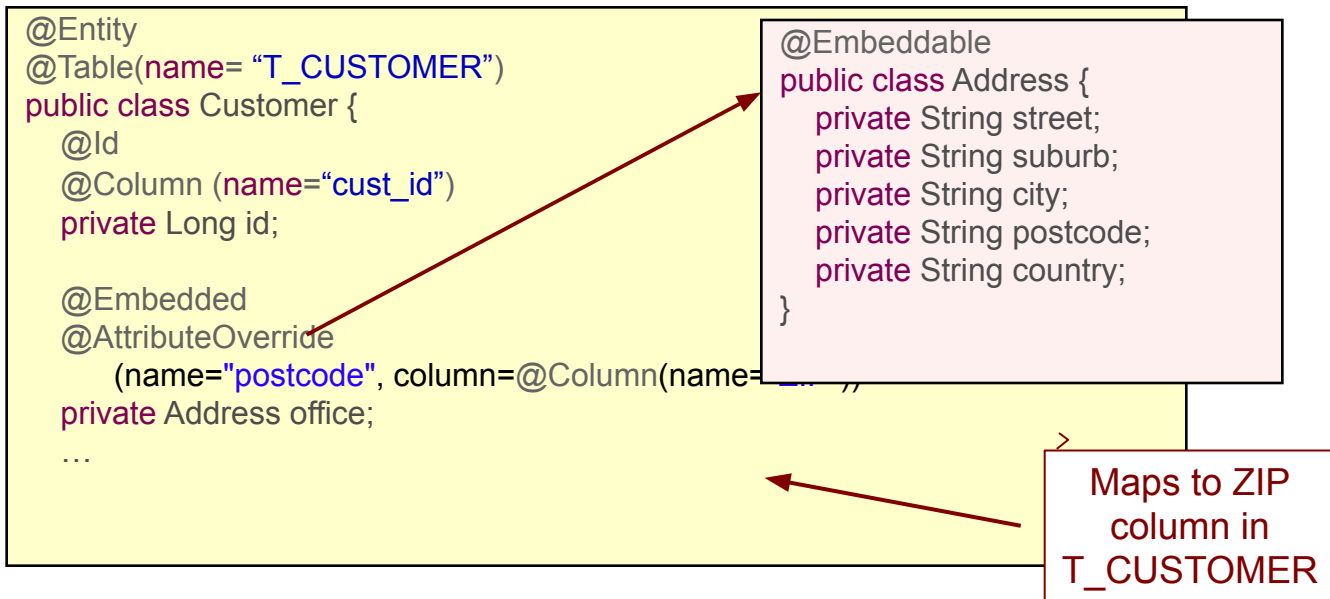
```java
@Entity
@Table(name= "T_ADDRESS")
public class Address {
    @Id private Long id;
    private String street;
    private String suburb;
    private String city;
    private String postcode;
    private String country;
}
```

Foreign key in
Address table

Pivotal

Embeddables

- Map a table row to multiple classes
  - Address fields also columns in T_CUSTOMER
  - @AttributeOverride overrides mapped column name

```
@Entity
@Table(name= "T_CUSTOMER")
public class Customer {
    @Id
    @Column (name="cust_id")
    private Long id;

    @Embedded
    @AttributeOverride
        (name="postcode", column=@Column(name=...
    private Address office;

    …
```

```
@Embeddable
public class Address {
    private String street;
    private String suburb;
    private String city;
    private String postcode;
    private String country;
}
```

Maps to ZIP column in T_CUSTOMER

Pivotal

JPA Querying

- JPA provides several options for accessing data
  - Retrieve an object by primary key
  - Query for objects using JPA Query Language (JPQL)
    - Similar to SQL and HQL
  - Query for objects using Criteria Queries (appendix)
    - API for creating ad hoc queries
  - Execute SQL directly to underlying database (appendix)
    - "Native" queries, allow DBMS-specific SQL to be used
    - Consider JdbcTemplate instead when not using managed objects – more options/control, more efficient

Pivotal.

JPA Querying: By Primary Key

- To retrieve an object by its database identifier simply call *find()* on the EntityManager

```
Long customerId = 123L;
Customer customer = entityManager.find(Customer.class, customerId);
```

returns **null** if no object exists for the identifier

No cast required – JPA uses generics

Pivotal

JPA Querying: JPQL

- SELECT clause required
- can't use *

- Query for objects based on properties or associations ...

```java
// Query with named parameters
TypedQuery<Customer> query = entityManager.createQuery(
        "select c from Customer c where c.address.city = :city", Customer.class);
query.setParameter("city", "Chicago");
List<Customer> customers = query.getResultList();

// … or using a single statement
List<Customer> customers2 = entityManager.
    createQuery("select c from Customer c ...", Customer.class).
    setParameter("city", "Chicago").getResultList();

// … or if expecting a single result
Customer customer = query.getSingleResult();
```

Specify class to
populate / return

Can also use bind ? variables
– indexed from 1 *like JDBC*

Pivotal.

Agenda

- Introduction to JPA

- **Configuring JPA in Spring**

- Optional and Advanced Topics

Pivotal

Quick Start – Spring JPA Configuration

- Steps to using JPA with Spring

1. Define Mapping Metadata (already covered)
2. Define an EntityManagerFactory bean.
3. Define Transaction Manager and DataSource beans
4. Define Repository/DAO

> Note: There are many configuration options for *EntityManagerFactory*, *persistence.xml*, and *DataSource*. See the optional section for details.

Pivotal.

2. Define the EntityManagerFactory

```java
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory() {

    HibernateJpaVendorAdapter adapter = new HibernateJpaVendorAdapter();
    adapter.setShowSql(true);
    adapter.setGenerateDdl(true);
    adapter.setDatabase(Database.HSQL);

    Properties props = new Properties();
    props.setProperty("hibernate.format_sql", "true");

    LocalContainerEntityManagerFactoryBean emfb =
                new LocalContainerEntityManagerFactoryBean();
    emfb.setDataSource(dataSource);
    emfb.setPackagesToScan("rewards.internal");
    emfb.setJpaProperties(props);
    emfb.setJpaVendorAdapter(adapter);

    return emfb
}
```

NOTE: no persistence.xml needed when using Spring's *packagesToScan* property

Pivotal

3. Transaction Manager & DataSource

```
 @Bean
 public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
    // See previous slide
    ...
    return entityManagerFactoryBean;
 }



 @Bean
 public PlatformTransactionManager
                 transactionManager(EntityManagerFactory emf) {
    return new JpaTransactionManager(emf);
 }




 @Bean
 public DataSource dataSource() {  /* Lookup via JNDI or create locally */ }
```
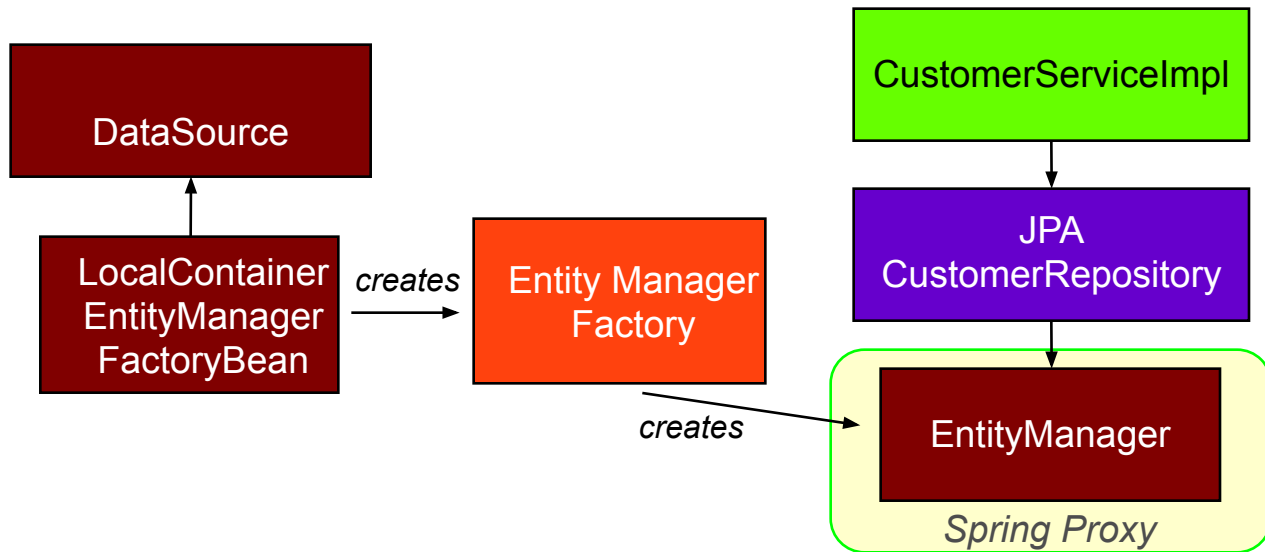
Method returns a *FactoryBean*...

… Spring calls getObject() on the FactoryBean to obtain the *EntityManagerFactory*:

Or … use `JtaTransactionManager`

Pivotal

EntityManagerFactoryBean Configuration

Pivotal.

4. Implementing JPA Repository

- The code – *no Spring dependencies*

Spring *Singleton* Bean

```java
public class JpaCustomerRepository implements CustomerRepository {

    private EntityManager entityManager;

    @PersistenceContext
    public void setEntityManager (EntityManager entityManager) {
        this. entityManager = entityManager;
    }

    public Customer findById(long orderId) {
        return entityManager.find(Customer.class, orderId);
    }
}
```

JPA's equivalent to *@Autowired*

Entity-manager is a *proxy*

Proxy resolves to entity-manager of *current request* when used

Pivotal

Define JPA Repository as Spring Bean

- The definition
  - No need to explicitly call `setEntityManager()`

```
@Bean
public CustomerRepository jpaCustomerRepository() {
    return new JpaCustomerRepository();
}
```
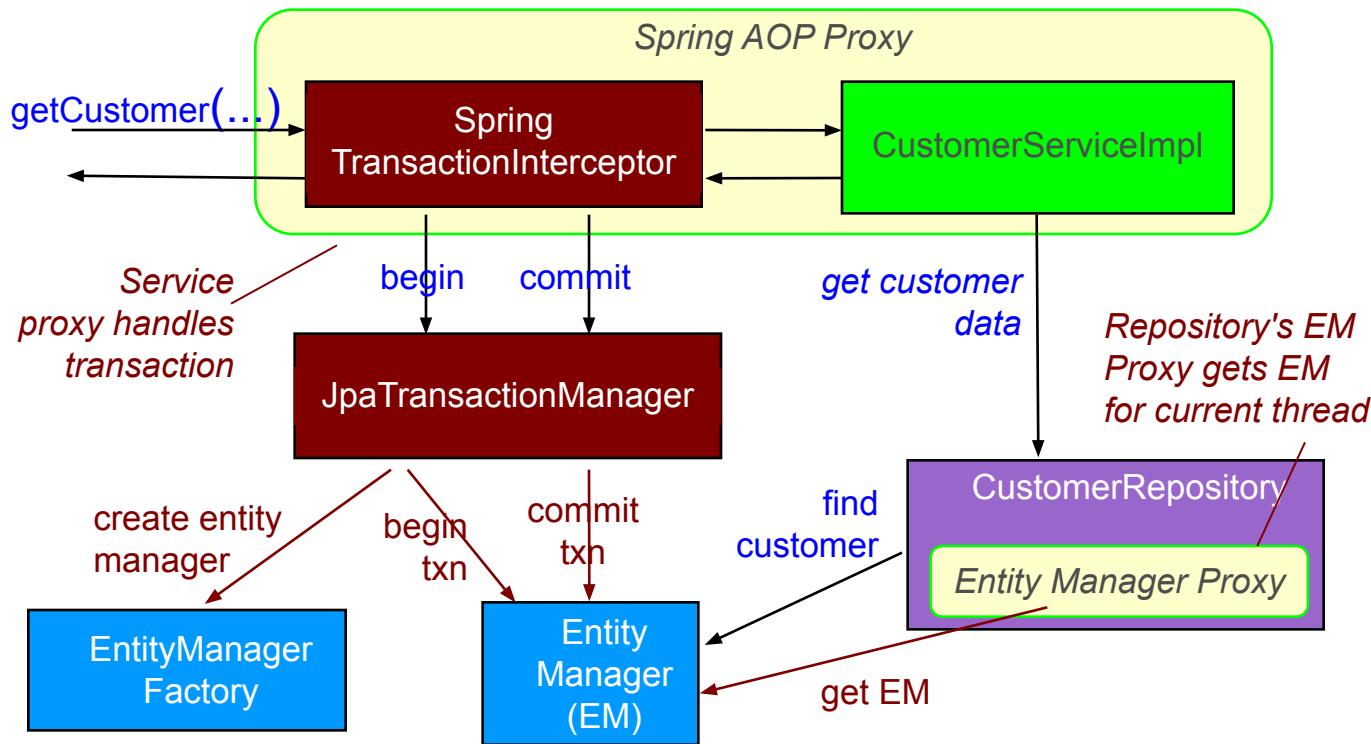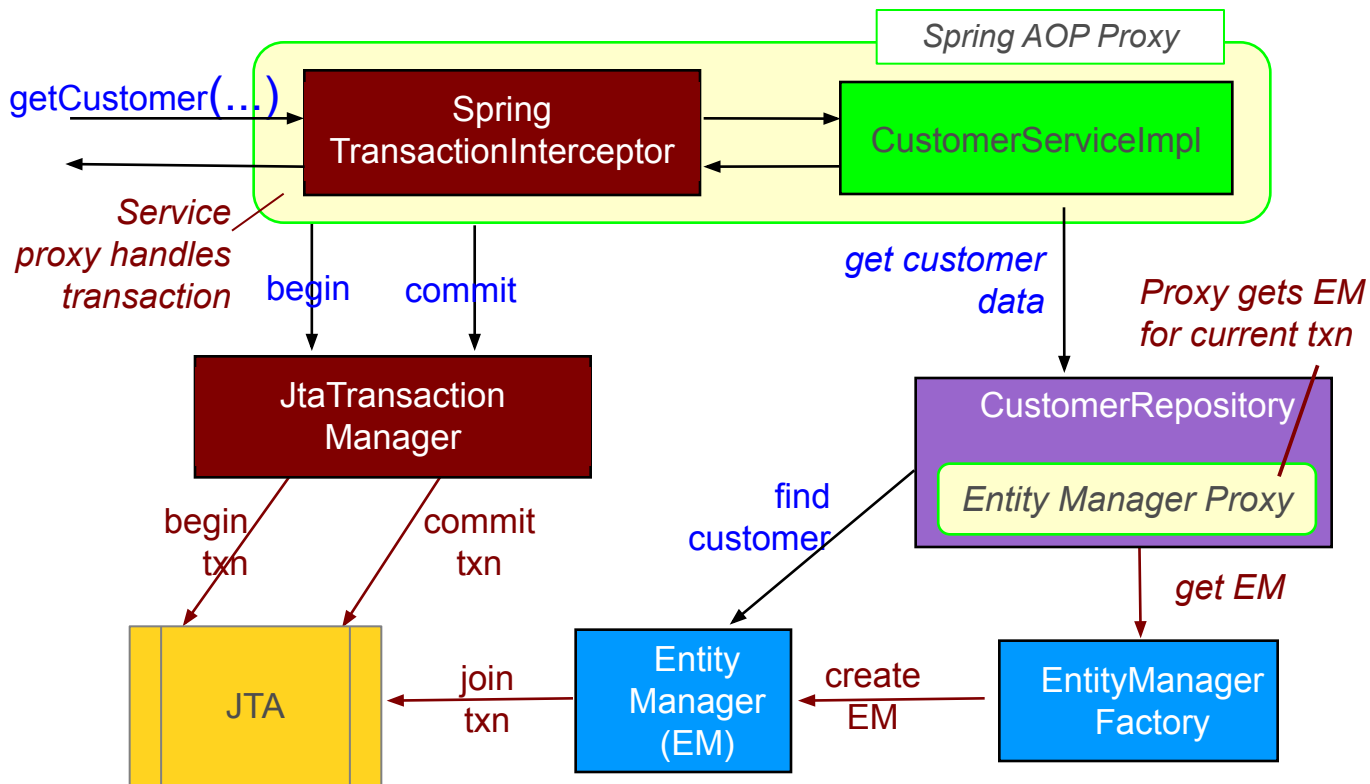
Automatic injection of entity-manager *proxy*

Pivotal

How it Works (JPA)

Spring defines *where* transactions are … JPA *implements* them

How it Works (JTA)

Spring defines *where* transactions are
… JTA *implements* them
… JPA joins and uses them

Spring AOP Proxy

getCustomer(…)

Spring
TransactionInterceptor

CustomerServiceImpl

Service proxy handles transaction

begin   commit

get customer data

Proxy gets EM for current txn

JtaTransaction
Manager

CustomerRepository

Entity Manager Proxy

begin txn   commit txn

find customer

get EM

JTA

join txn

Entity
Manager
(EM)

create EM

EntityManager
Factory

Pivotal.

Implementing JPA DAOs with Spring

- Spring defines *where* transactions occur
  - Delegates to a JPA EntityManager to implement them
  - Supports local or global (JTA) transactions

- *EntityManager is a* "proxy"
  - Allows a singleton Repository to access the right *EntityManager* for current transaction in current thread

- There are *no* Spring dependencies in your Repository (DAO) implementations

Pivotal.

Summary

- Use 100% JPA to define entities and access data
  - Repositories have no Spring dependency

- Use Spring to configure JPA entity-manager factory
  - Smart proxy works with Spring-driven transactions
  - Spring defines where transactions begin & end
  - JPA entity-managers implements them

Pivotal.

Agenda

- Introduction to JPA

- Configuring JPA in Spring

- **Optional and Advanced Topics**
  - **Exception Translation**
  - JPA Typed Queries / Native Queries
  - EntityManagerFactoryBean alternatives / persistence.xml

Pivotal.
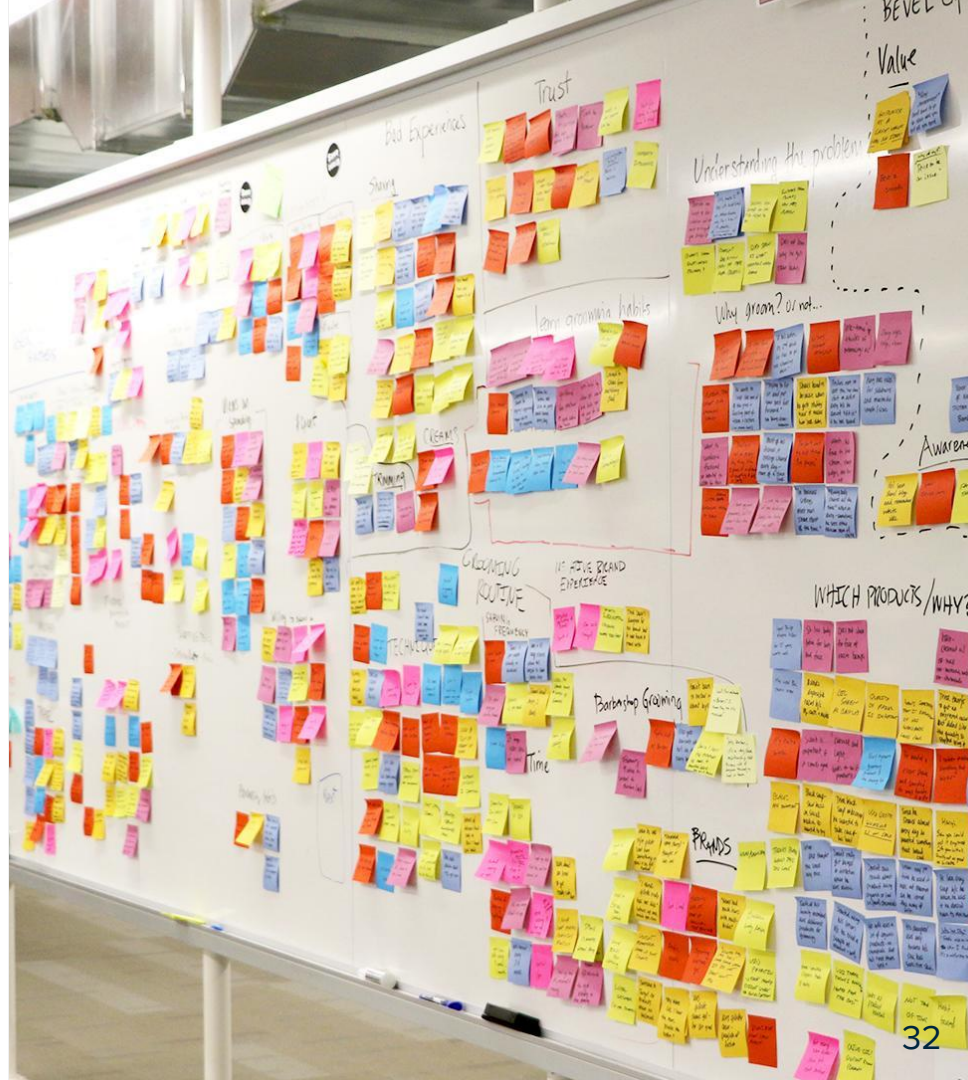
Transparent Exception Translation (1)

- Used as-is, the DAO implementations described earlier will throw unchecked JPA PersistenceExceptions
  - Not desirable to let these propagate up to the service layer or other users of the DAOs
  - Introduces dependency on the specific persistence solution that should not exist

- AOP allows translation to Spring's rich, vendor-neutral `DataAccessException` hierarchy
  - Hides access technology used

Transparent Exception Translation (2)

- Spring provides this capability out of the box
  - Annotate with `@Repository`
 - Is also an `@Component`, so will be found by component-scanner

```
@Repository
public class JpaCustomerRepository implements CustomerRepository {
    ...
}
```

Pivotal.

Transparent Exception Translation (3)

- Make Spring apply AOP to the the annotated repository
  - Requires a Spring-provided BeanPostProcessor
    - `PersistenceExceptionTranslationPostProcessor`
  - If any Spring bean defines
    `PersistenceException-Translator`
    - Spring creates a
      `PersistenceException-TranslationPostProcessor` bean *automatically*

- `LocalContainerEntityManagerFactoryBean` does this for you
  - All you need to do is use `@Repository`

Agenda

- Introduction to JPA

- Configuring JPA in Spring

- **Optional and Advanced Topics**
    - Exception Translation
    - **JPA Typed Queries / Native Queries**
    - EntityManagerFactoryBean alternatives / persistence.xml

Pivotal.
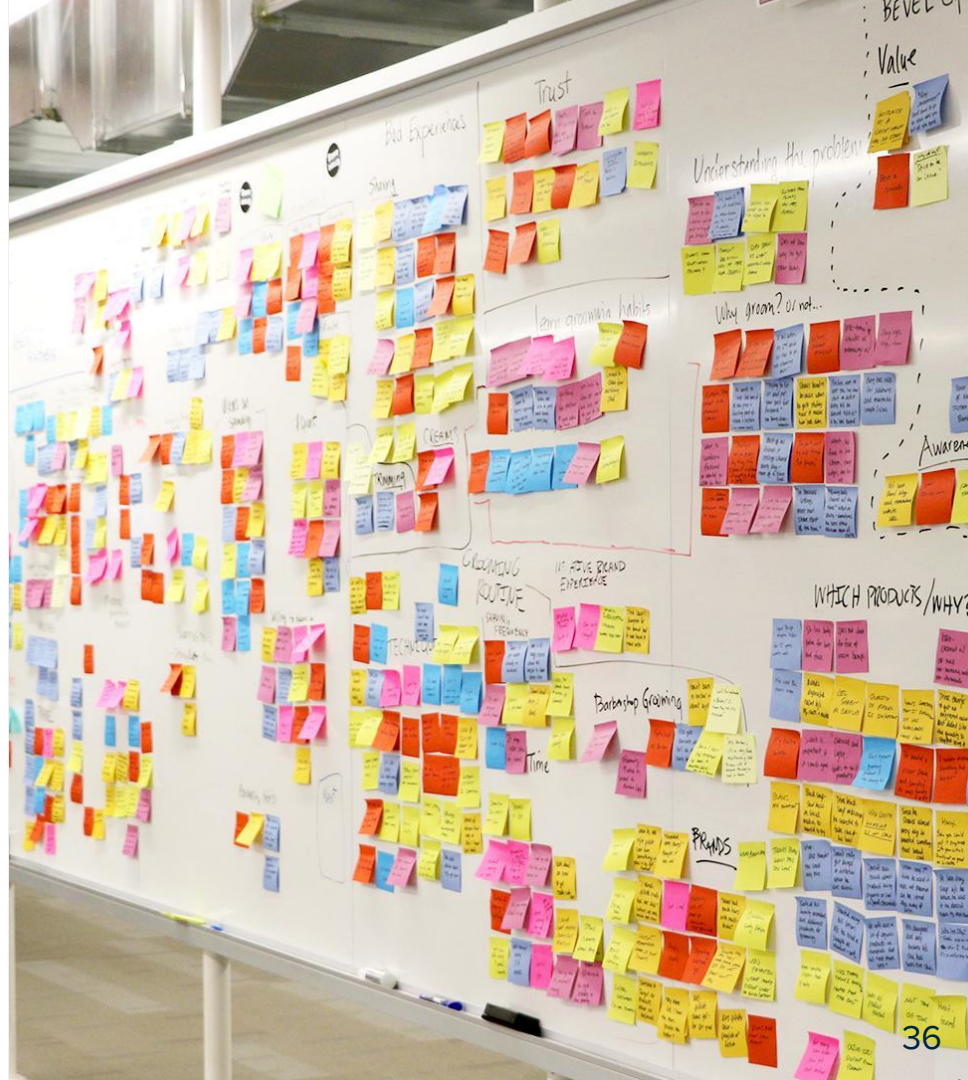
JPA Querying: Typed Queries

- Criteria Query API (JPA 2)
  - Build type safe queries: fewer run-time errors
  - Much more verbose

```java
public List<Customer> findByLastName(String lastName) {
    CriteriaBuilder builder = entityManager.getCriteriaBuilder();
    CriteriaQuery<Customer> cq = builder.createQuery(Customer.class);
    Predicate condition =
        builder.equal( cq.from(Customer.class).get(Customer_.lastNname), lastName);
    cq.where(condition);

    return entityManager.createQuery(cq).getResultList();
}
```

Meta-data class created by JPA (note underscore)

Pivotal

JPA Querying: SQL

- Use a *native* query to execute raw SQL

```
// Query for multiple rows
Query query = entityManager.createNativeQuery(
    "SELECT cust_num FROM T_CUSTOMER c WHERE cust_name LIKE ?");
query.setParameter(1, "%ACME%");
List<String> customerNumbers = query.getResultList();

// … or if expecting a single result
String customerNumber = (String) query.getSingleResult();

// Query for multiple columns
Query query = entityManager.createNativeQuery(
    "SELECT ... FROM T_CUSTOMER c WHERE ...", Customer.class);
List<Customer> customers = query.getResultList();
```

No *named* parameter support

Indexed from 1 – *like* JDBC

Specify class to populate/return

Pivotal.

Agenda

- Introduction to JPA

- Configuring JPA in Spring

- **Optional and Advanced Topics**
  - Exception Translation
  - JPA Typed Queries / Native Queries
  - **EntityManagerFactoryBean alternatives / persistence.xml**

Pivotal.

Setting up an EntityManagerFactory

- Three ways to set up an EntityManagerFactory:
  - LocalEntityManagerFactoryBean
  - LocalContainerEntityManagerFactoryBean
  - Use a JNDI lookup

- persistence.xml required for configuration
  - From version 3.1, Spring allows no *persistence.xml* with LocalContainerEntityManagerFactoryBean

Pivotal.

`persistence.xml`

<?xml?>

- Always stored in META-INF
- Specifies "persistence unit":
  - optional vendor-dependent information
  - DB Connection properties often specified here

```xml
<persistence version="1.0"
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
  <persistence-unit name="rewardNetwork"/>

  ...
</persistence>
```

  - File required by JPA, optional when using Spring with JPA!

LocalContainer EntityManagerFactoryBean

- Provides full JPA capabilities
- Integrates with existing DataSources
- Useful when fine-grained customization needed
  – Can specify vendor-specific configuration

We saw this earlier using
100% Spring configuration
In both XML and Java

Configuration – Spring *and* Persistence Unit

```java
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
    LocalContainerEntityManagerFactoryBean emfb =
        new LocalContainerEntityManagerFactoryBean();
    emfb.setDataSource(dataSource);
    emfb.setPersistenceUnitName("rewardNetwork");
    return emfb;
}
```

Minimal Spring Configuration

```xml
<persistence-unit name="rewardNetwork">
  <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <properties>
     <property name="hibernate.dialect"
               value="org.hibernate.dialect.HSQLDialect"/>
     <property name="hibernate.hbm2ddl.auto" value="create"/>
     <property name="hibernate.show_sql" value="true" />
     <property name="hibernate.format_sql" value="true" />
    </properties>
</persistence-unit>
```

Do JPA configuration in persistence.xml

If using JTA, declare *<jta-data-source>* in persistence.xml

Pivotal.

JNDI Lookups

- JNDI lookup used to retrieve an *EntityManagerFactory* from an application server
  - Use when deploying to JEE Application Servers
    - WebSphere, WebLogic, JBoss, Glassfish …

```
@Bean
public EntityManagerFactory entityManagerFactory() throws Exception {
    Context ctx = new InitialContext();
    return (EntityManagerFactory) ctx.lookup("persistence/rewardNetwork");
}
```

Topics Covered

- Introduction to JPA
- Configuring JPA in Spring
- Optional and Advanced Topics

Pivotal.