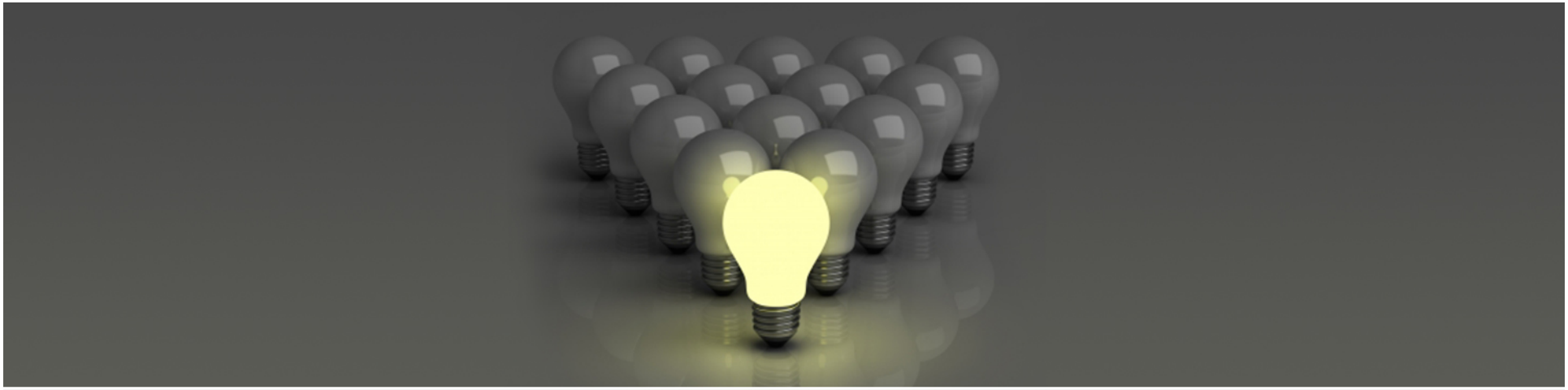
# UPA

## Unstructured Persistence API

**Yet another OPEN SOURCE ORM challenger**



# Agenda

- Motivation
  - ✓Hibernate, MyBatis, JPA, isn't that just enough
- Common features
  - ✓Soft Learning curve as features from other ORM are made easy
- Unique features
  - ✓Challenge is yet to come…
- Use cases
  - ✓How **not** to's…
- Getting Started
  - ✓First example…

# Motivations

- Dynamic (at Runtime Level) Schema Definition
  - ✓Overtake traditional ORMs that permit only mappings for existing (on compile time) Entities
  - ✓Provide a clean way to get customizable schema for end users
  - ✓Extra fields at Runtime (Contact/Invoice custom fields)
  - ✓Extra entities at runtime
  - ✓Add Entities/Associations for Plug-in based applications
  - ✓Should handle alter/modify/drop objects as well
  - ✓Helps enforcing “Pyramid Design Pattern”

# Motivations

- SQL concepts handling
  - ✓Focus on portability
  - ✓Yet provide mapping to
    - ✓Views
    - ✓Procedures
    - ✓Functions
    - ✓Triggers

# Motivations

- Productivity vs Portability
  - ✓Focus on productivity
  - ✓Don’t loose portability
- UPA? It all started as proof of concept !
  - ✓On 2011
  - ✓Upon lecture I presented in SFD/Software Freedom Day 2013 about “7 ORM sins”.

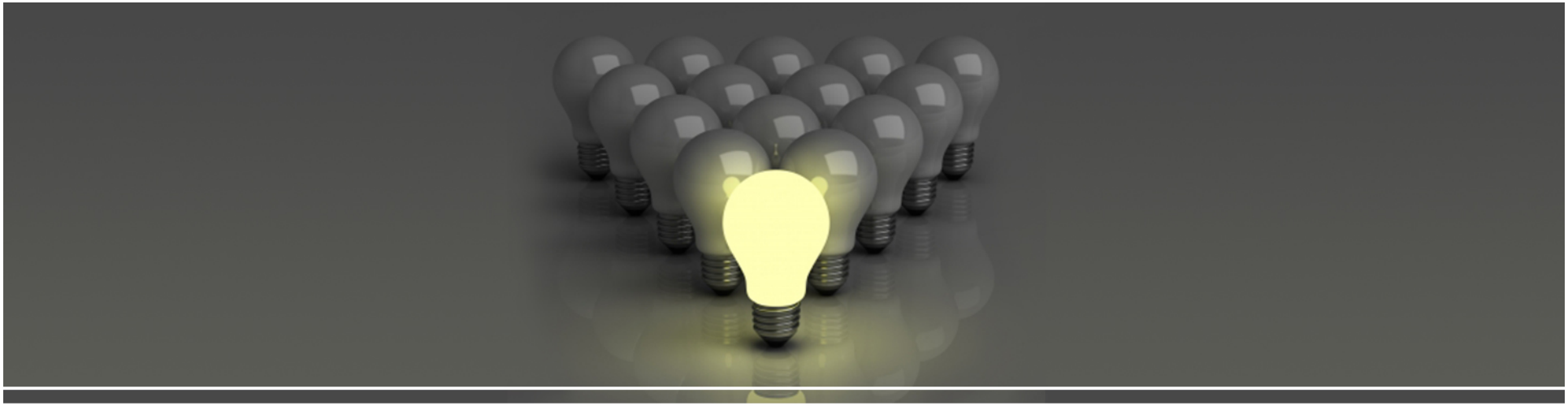
# But you worry about…

- Any Security issues  with DDL (drop/alter) ?
  - ✓May Simply  be handled by RDBMS.
- Any performance issues ?
  - ✓Think of vertical tables (properties table, parameters table, adresses table) that will be handled horizontally.

# UPA

## Common features

**So what's common about it**



# Common features

UPA is not exactly JPA still …

- Intuitive
  - ✓Almost all JPA features/concepts are maintained. Some Hibernate and Mybatis ones are bundled in a way it fits well with UPA philosophy.
- Vendor Neutral Persistence Layer
  - ✓Helps build a persistence layer that is vendor neutral and any persistence provider can be used. Although, UPA provides a **reference implementation** that is particularly ready to use.
- Pluggable Providers
  - ✓Supports pluggable, third party persistence providers as it is defined as an **API** with a reference implementation.

# Common features

- Annotations based meta-data
  - ✓No deployment descriptors required
  - ✓Very similar of JPA's annotations.
  - ✓Annotations defaults can be used in model class, which saves a lot of development time
  - ✓Support for XML Mapping
- Standardized ORM
  - ✓Provides clean, easy, and standardized object-relational mapping

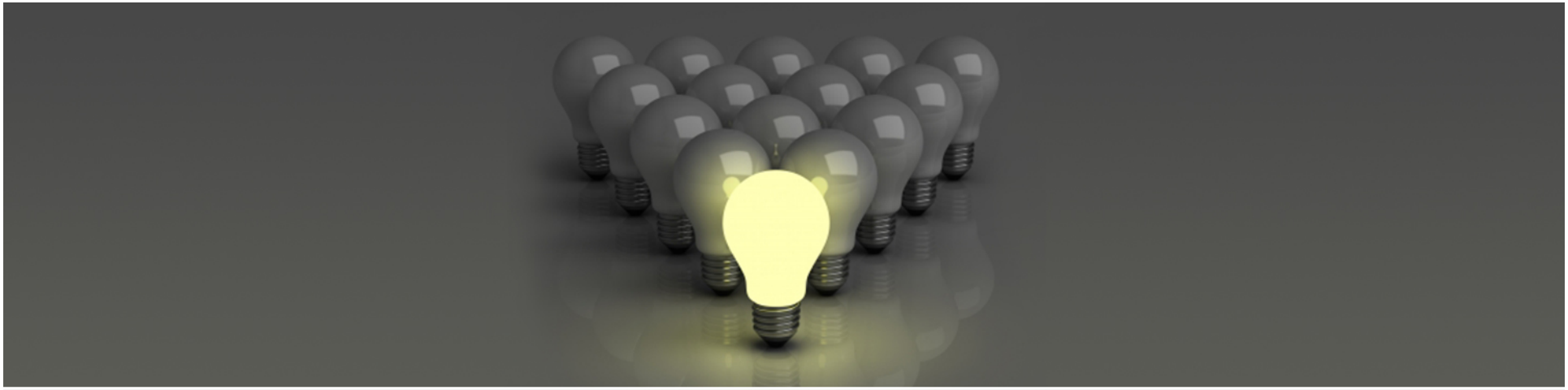
# Common features

- Query language
  - ✓UPQL is very powerfully query language provided by UPA providing abstraction layer over the persistence model. UPQL makes it possible to avoid specific RDBMS dialects.
  - ✓Similar to JPQL and HQL (almost compatible)
- Model generation
  - ✓UPA application can be configured to generate database schema based on persistence model
- Portability
  - ✓It is meant to be easy to switch to other persistence providers. You can move to any commercial persistence provider when needed. Yet there is no commercial provider planned to come.

# UPA

## Unique Features

**Still don't get it why this staff has not been done yet!**



Taha BEN SALAH - 2015

# Unique Features

- Inside/Outside containers
  - ✓UPA application can run outside the container also. So, developers can use UPA capabilities in desktop applications, Web containers, EJB containers, Spring containers, …
  - ✓The very same code runs Inside/Outside containers. **NO** modification is needed.
  - ✓Helps Sketching and Testing JEE Apps
  - ✓Depending on the current container, UPA context will be automatically created and maintained

```
public class InvoiceModule {

    public List<Invoice> findAllInvoices() {
        PersistenceUnit pu = UPA.getPersistenceUnit();
        return pu.findAll(Invoice.class);
    }
```

*Context aware utility method*

# Unique Features

- Annotations based meta-data
  - ✓Configuration is done equally through
    - ✓Model Classes (Annotation)
    - ✓Mapping Classes (Partial Annotations)
    - ✓Xml Files (upa.xml)
    - ✓Support for override mapping, partial mapping and precedence/order.

```
@Entity
public class Product {
    @Id  @Sequence
    private int productId;
    private String name;
    private double priceTaxFree;
    private double vat;
    private double quantity;
```

*Model Class*

```
public class Invoice {

    private String invoiceId;
    private Date date;
    private Integer customerId;
    private Customer customer;
```

*Plain Model Class*

```
@Entity(entityType = Invoice.class)
public class InvoiceMO {
    @Id
    private FieldDesc invoiceId;
    @Ignore
    private FieldDesc details;
```

*Mapping Object Class*

# Unique Features

- Reflexion API
  - ✓Large applications need better reflexion mechanisms to handle general purpose use cases. This feature enables developers to be aware of used data model at runtime : entities, fields, datatypes, and perform accordingly ...

```
PersistenceUnit pu = UPA.getPersistenceUnit();
Entity e = pu.getEntity(Product.class);
Field id = e.getField("id");
System.out.println(id.getDataType());
```

*Runtime metadata retrieval*

# Unique Features

- Dynamic data definition and alteration
  - ✓Enable at runtime data structure alteration by creating new entities or altering existing entities by introducing new fields, removing some fields etc.
  - ✓Helpful for dynamic model based applications that usually uses vertical tables (columns as rows) which are of very little performance.

```
PersistenceUnit pu = UPA.getPersistenceUnit();
Entity e = pu.addEntity("CustomEntity");
e.addField("id", null,
        FlagSets.of(FieldModifier.ID), null,
        TypesFactory.INT);
Entity e2 = pu.addEntity(Product.class);
e2.addField("extra", null, null, null,
        TypesFactory.STRING);
```

*Define Entity by name or by Type*

# Comprehensive Model Structure

- Persistence Group
  - ✓UPA Context supports multiple Persistence Groups, each maps to application context with one or multiple Persistence Units
- Persistence Unit
  - ✓Defines Schema context with Datasource definition and includes Packages , Entities, Associations, …
- Package
  - ✓Enforces modularity by regrouping sub Packages, Entities,… Maps well to Application Plugins with subsequent model.
- Entity
  - ✓Maps to Tables, Views, Procedures, Plain Queries or any combination. Maps also to custom code.
- Section
  - ✓Maps to Field Sets of an Entity. Maps well to Application Plugins with subsequent model (think of pay module defining custom fields on Employee Entity)
- Field
  - ✓Maps to standard or custom (virtual, formulas) columns.

# Comprehensive Model Structure

```
PersistenceUnit pu = UPA.getPersistenceUnit();
for (net.vpc.upa.Package p : pu.getPackages()) {
    List<Entity> entities = p.getEntities();
    for (Entity e : entities) {
        for (Section s : e.getSections()) {
            for (Field f : s.getFields()) {
                System.out.println(f.getName());
                System.out.println(f.getDataType())
            }
        }
    }
}
```

*Browse Model structure (sub-packages and sub-sections are ignored here)*

# Generated Formula Fields

- Supports natively customizable formula fields
  - ✓ Values are generated according to custom expressions and conditions. (Example : Total field)
  - ✓Complex Expressions supported (almost any UPQL Expression)
  - ✓Rich type set (Live formulas, persisted formulas, insert only formulas, …)
  - ✓Support for formula dependency (fields formulas that depend on each other)

```
@Formula(value="Coalesce((Select Sum(x.priceTaxFree) "
        + "From InvoiceDetail x "
        + "Where "
        + "x.invoiceId=this.invoiceId),0)"
        ,type = FormulaType.UPDATE
)
private FieldDesc totalTaxFree;
```

*Define an update only persistent formula depending on subsequent asssociation*

# Generated Formula Fields

- Support for Sequences
  - ✓Incremental Fields (Includes Auto Increment ID Column but not only)
  - ✓Support for wide range of types (not only integers but also Strings, dates, …)
  - ✓Support for Grouped Sequences (reset on group change : think of year based sequences)

```
@Id
@Sequence(format = "{datepart(year,currentDate())}/{#}"
          ,initialValue=1,
          allocationSize=1)
private FieldDesc invoiceId;
```

*Define Sequence with values as follows 1014/1, 1014/2 … 1014/23, 1015/1, …*

# Ready to use Entity Patterns

- Tree Entities
  - ✓Recursive Associations
  - ✓Support for Depth first Search
- Singletons
  - ✓Single Row entities (think of parameters entity)
- Union Entities
  - ✓More generic than Inheritance, Union makes it very simple to build custom Entities
- Views
  - ✓Portable
  - ✓Defines views with simple UPQL

# Ready to use Entity Patterns

```
@Entity
public class Category {

    @Id @Sequence
    private int id;
    private String name;
    @Hierarchy
    private Category parent;
```

*Hierarchy defined on Category parent field*

```
@View(
    query = "Select o from Product where o.country='TN'"
)
public class TunisianProducts {
    @Id @Sequence
    private int id;
    private String name;
```

*View Entity with Query based on existing Entity*

```
@Singleton
public class Config {

    private String emailServer;
    private String emailAddress;
```

*Singleton Entity (needs no id)*

# Shared Model

- Classes can be mutualized for multiple Entities
- Same Table can be described by multiple Entities
- Examples
  - ✓Generic class holder "NamedEntity" (with solely id and name) to use for all drop down components
  - ✓Single Table with discriminator (semantically not an inheritance)

```
PersistenceUnit pu = UPA.getPersistenceUnit();
List<NamedEntity> users = pu.createQuery
  ("Select a from User a")
  .getTypeList(NamedEntity.class);
List<NamedEntity> categories = pu.createQuery
  ("Select a from Categories a")
  .getTypeList(NamedEntity.class);
```

*Dynamic mapping to mutualized types*

# Partial Operations

- Dynamic Lazy/Partial loading
  - ✓ Runtime selection of needed fields/associations for retrieval
  - ✓ In most cases, listing entities does not require retrieval of all information
- Dynamic Partial updates
  - ✓Better interaction with outside world

```
PersistenceUnit pu = UPA.getPersistenceUnit();
List<Customer> customers = pu.createQuery
  ("Select a.name,a.password from Customer a")
  .getEntityList();
```

*Define what fields to retrieve*

# Custom Persistence processing

- Very flexible persistence processing
- Each operation is uniquely customizable
- Example
  - ✓**Select** from View
  - ✓**Insert** into 2 Tables
  - ✓**Update** using stored procedure call
  - ✓**Delete** with custom Java call

```
PersistenceUnit pu = UPA.getPersistenceUnit();
Entity e = pu.getEntity(Customer.class);
e.getEntityPersister().setDeleteAction(
        new MyCustomEntityDeleteAction()
);
```

*Customize Delete Operation*

# Rich Callback System

- Interceptors on Data Updates and Structure Alterations
- Soft (written in Java)/Hard(Java translated to PLSQL*) Triggers
- Updatable data in Callback context (unlike JPA)

```
@Callback
public class InventoryServiceCallback
    extends EntityDataListenerAdapter
    implements EntityDataListener {

    @Override
    public void onInsert(Entity entity, Object insertedId, Record record)
```

*Soft Trigger*

(*) on progress

# Rich Callback System

```
@Callback
public class MyIntrospector extends DefinitionListenerAdapter
        implements EntityDefinitionListener, FieldDefinitionListener {

    @Override
    public void entityAdded(EntityDefinitionEvent event) {
        // ...
    }

    @Override
    public void fieldRemoved(FieldDefinitionEvent event) {
        // ...
    }

}
```

*Data Definition Trigger*

# Complex Datatypes

- Portable support for custom and complex datatypes
- Each operation is uniquely customizable
  - ✓Support for all common data types
  - ✓Provides a portable manner to extend supported types with new custom/complex ones
  - ✓Similar to Embedded/Embeddable features in JPA although it provides a more extensible manner

# Import / Export

- Import/Export API
  - ✓Support for (very) large files
  - ✓CSV, XML, XLS, XLSX
  - ✓Support for associations

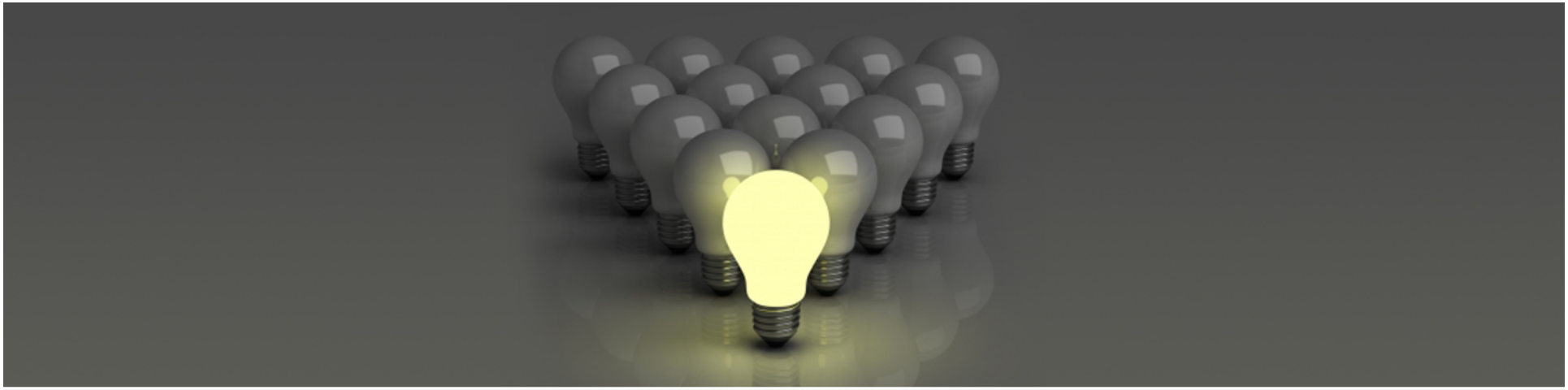
```
PersistenceUnit pu = UPA.getPersistenceUnit();
ImportExportManager iem = pu.getImportExportManager();
SheetFormatter f = iem.createSheetFormatter(new File("a.xls"));
f.setContentType(SheetContentType.XLS );
DataWriter w = f.createWriter();
List<Customer> customers = pu.findAll(Customer.class);
for (Customer c : customers) {
    w.writeObject(c);
}
w.close();
```

*Export all Customers to a.xls file*

# UPA

## Use Cases

**How not to do ugly things…using beautiful UPA**



Taha BEN SALAH – ENISo – Tunisia - 2015

# Use Case 1

- Use case 1 : How not to Break 3 tiers paradigm
- Use case 2 : How not to Tediously track object modifications
- Use case 3 : How not to Ugly support hashed passwords in User like entities
- Use case 4 : How not to Refactor entire application for multitenant support
- Use case 5 : How not to Painfully rename physical model (tables...)

# Use Case 1

How not to Break 3 tiers paradigm

# How not to Break 3 tiers paradigm

- Introduce Object Mapping (MO) Pattern
- Model Class no longer has to be spoiled with DAL annotations
- Example
  - ✓Invoice MO adds @Id on invoiceID
  - ✓FieldDesc is used not to alter original field type

```
public class Invoice {

    private String invoiceId;
    private Date date;
    private Integer customerId;
    private Customer customer;
```

```
@Entity(entityType = Invoice.class)
public class InvoiceMO {
    @Id
    private FieldDesc invoiceId;
    @Ignore
    private FieldDesc details;
```

*Model Class no longer get spoiled with annotations*

# Use Case 2

How not to Tediously track object modifications

# Track objects

- Although added creationDate/creationUser is a common use case, it is still tedious to generalize its use in real world applications.
- Callbacks combined with formula concepts in UPA makes it very easy to face this problem
- Each time a new entity is detected, new fields will be added automatically with default values interpreted as formulas (current user, and current date are retrieved from system context)

# Track objects

```
@Callback
public class TrackingFeature extends DefinitionListenerAdapter implements EntityDefinitionListene
    @Override
    public void entityAdded(EntityDefinitionEvent event) {
        if (event.isAfter()) {
            Entity entity = event.getEntity();
            Section tracking = entity.addSection("Tracking");
            tracking.addField("creationDate", null, null, TimestampType.DEFAULT)
                    .setInsertFormula("currentTimestamp()");
            tracking.addField("creationUser", null, null, StringType.DEFAULT)
                    .setInsertFormula("currentUser()");
            tracking.addField("modificationDate", null, null, TimestampType.DEFAULT)
                    .setFormula("currentTimestamp()");
            tracking.addField("modificationUser", null, null, StringType.DEFAULT)
                    .setFormula("currentUser()");
            tracking.addField("revision", null, 0L, LongType.DEFAULT)
                    .setUpdateFormula("revision+1");
        }
    }
```

*Added 5 fields dynamically for all entities*

# Use Case 3

How not to ugly support hashed passwords in User like entities

# Hashed passwords

- Using @Password annotation to mention hashed field
- No other changes have to be made on queries

```
@Entity
@Path("Security")
public class User {
    @Id @Sequence
    private int id;
    @Password
    private String password;
```

```
public User findUser(String login, String password) {
    PersistenceUnit pu = UPA.getPersistenceUnit();
    return (User) pu
            .createQuery("Select u from User u "
                    + "where "
                    + "u.login=:login "
                    + "and u.password=:password")
            .setParameter("login", login)
            .setParameter("password", password)
            .getEntity();
}
```

*Query does not care about hash mechanism*

# Hashed passwords

- Several Hash Algorithms supported (MD5, SHA,...)
- Support for custom digest

```
@Entity
@Path("Security")
public class User {
    @Id @Sequence
    private int id;
    @Password(strategyType = PasswordStrategyType.MD5)
    private String password;
```

*Use custom hash digest*

# Use Case 4

How not to Refactor entire application for multitenant support

# Multi-tenant support

- Need add “tenantId” field to all entities
  - ✓Use @Callback feature
- Need filter all queries by current “tenantId”
  - ✓User Entity Filter feature
- Need change all inserts to  mention current “tenantId”
  - ✓Use formula (persistent) feature

# Multi-tenant support

- Create **CurrentTenant** function
  - ✓For demonstration purposes will use thread local context to store current tenant info

```
@net.vpc.upa.config.Function
public static class CurrentTenant implements net.vpc.upa.Function {

    public final static ThreadLocal<Integer>
            currentTenant = new ThreadLocal<Integer>();

    static {
        currentTenant.set(15);
    }

    @Override
    public Integer eval(String name, Object[] args,
            PersistenceUnit persistenceUnit) {
        return currentTenant.get();
    }
}
```

*Custom* UPQL *function*

# Multi-tenant support

- Create **MultiTenantFeature** feature
  - ✓Add field formula tenantId evaluated to formula “CurrentTenant()” function just defined
  - ✓Add filter to Entity by tenantId

```
@Callback
public class MultiTenantFeature extends DefinitionListenerAdapter
   implements EntityDefinitionListener {

    @Override
    public void entityAdded(EntityDefinitionEvent event) {
        if (event.isAfter()) {
            Entity entity = event.getEntity();
            entity.addField("tenantId", "MultiTenant", null, null, IntType.DEFAULT)
                    .setInsertFormula("CurrentTenant()");

            // filter entities by tenantId
            entity.addFilter("MultiTenant", "tenantId=CurrentTenant()");
        }
    }
```

*Custom tenant field and filter*

# Use Case 5

How not to Painfully rename physical model (tables...)

# Naming strategy

- Name pattern
  - ✓Can be in Java Code or in xml (META-INF/upa.xml file)
  - ✓In this example tables are upper cased and prefixed with “T_”

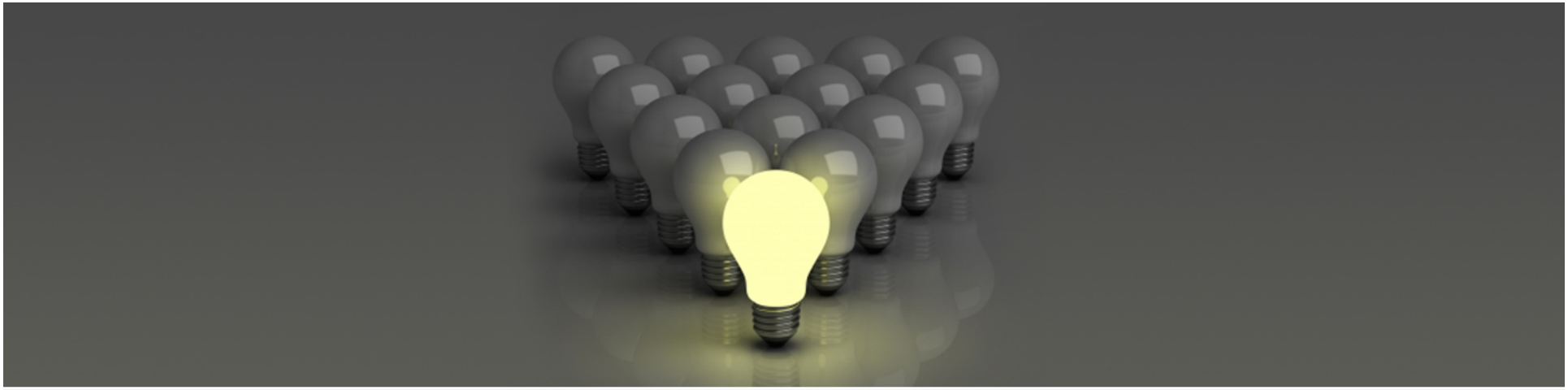
```
@PersistenceNameStrategy(
        persistenceName = "{OBJECT_NAME}",
        names = {
            @PersistenceName(
                    type = PersistenceNameType.TABLE,
                    value = "T_{OBJECT_NAME}"
            ),
            @PersistenceName(
                    type = PersistenceNameType.FK_CONSTRAINT,
                    value = "FK_{OBJECT_NAME}"
            )
        }
)
public class NamingStrategyFeature {
}
```

*Rename Tables, and FK constraints*

# UPA

## Getting started

**Let’s rock!**



# Getting Started

- Configure Maven dependencies
  - ✓Add github repository reference
  - ✓Add « upa-api » as compile dependency
  - ✓Add « upa-impl » as runtime dependency
  - ✓Add « upa-web» (optional) as runtime dependency if used in web context

```
<repositories>
    <repository>
        <id>vpc-public-maven</id>
        <url>https://raw.github.com/thevpc/vpc-public-maven/master</url>
        <snapshots>
            <enabled>true</enabled>
            <updatePolicy>always</updatePolicy>
        </snapshots>
    </repository>
</repositories>
<dependencies>
    <dependency>
        <groupId>net.vpc.upa</groupId>
        <artifactId>upa-api</artifactId>
        <version>[1.2,)</version>
    </dependency>
    <dependency>
        <groupId>net.vpc.upa</groupId>
        <artifactId>upa-impl</artifactId>
        <version>[1.2,)</version>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>net.vpc.upa</groupId>
        <artifactId>upa-web</artifactId>
        <version>[1.0,)</version>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

*Maven support*

# Getting Started

- Create /META-INF/upa.xml resource config file
  - ✓Needed to customize datasource
  - ✓Dataousrce is language independant (may use jdbc format or universal connection string)
  - ✓May be used to define naming strategy

```
<?xml version="1.0" encoding="UTF-8"?>
<upa version="1.0">
    <connection>
        <connectionString>
            derby:default://localhost/upatutorial;structure=create
        </connectionString>
        <userName>me</userName>
        <password>never</password>
    </connection>
    <property name="persistenceName">upa_{object_name}</property>
</upa>
```

*Sample upa config file*

# Getting Started

- All you need than need now is to create entity classes and use PersistenceUnit.
- That's it!

```java
public class Invoice {

    private String invoiceId;
    private Date date;
    private Integer customerId;
    private Customer customer;
```

*Model Class*

```java
@Entity(entityType = Invoice.class)
public class InvoiceMO {
    @Id
    private FieldDesc invoiceId;
    @Ignore
    private FieldDesc details;
```

*Mapping Object Class*

```java
public class InvoiceModule {

    public List<Invoice> findAllInvoices() {
        PersistenceUnit pu = UPA.getPersistenceUnit();
        return pu.findAll(Invoice.class);
    }
```

*Call PU*

UPA is more than a simple ORM
We try to think of it as next generation ORM Framework
How do you find it ? I'm curious about that...

Thank you

Taha BEN SALAH – 2015
taha.bensalah@gmail.com