# Event tracker (ETR)

| | |
|---|---|
| Project: | Event tracker |
| Inception: | 2014-01-13 |
| Goal: | Based on sample of event domain model show how different combinations of industry popular (Java) frameworks(frmk-s)/approaches can be used to handle events. For persistence entity CRUD operations in different DBs to be explored. |

## Contents

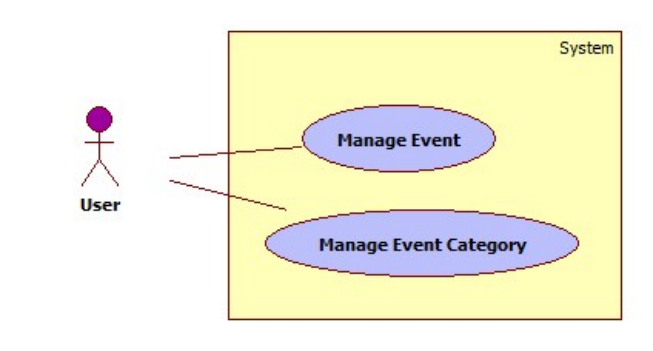


## Introduction

There are tons of approaches to work with popular frmks e,g, for databases persistence. It's easy to get lost among them even if you're seasoned developer. Once you start this endeavour, you realise that documentation varies from frmk to frmk in details and quality, sample code and configs are not for combination you assess, particular versions of frmk-s don't live together well. Forums - though useful, sometimes more confuse than help. It's especially common when the newest frmk versions are emerged (notice this set of projects generally grabs the latest stable version of whichever frmk).
Another way to look at it: you joined new team. They use some combination of frmk-s which you didn't try. Codebase is large and not easy to grasp at once. Performance optimizations and bug fixes/workarounds can greatly reduce code readability. Yes, there are books, forums, guru fellows who can help out. But forums can mislead(for instance discuss elder frmk versions), gurus are busy , and you want to see sample actionable code and run it now, learn in a matter of minutes how typical combination of those frmk-s works, how much config/code it "consumes". This project tries to embrace it via providing code snippets and configs which proven to work for particular frmk-s(generally the newest on time of addition).

## 1. Requirements/wishes/dreams

Use cases are extremely simple from high level:




For 'Manage Event' main scenarios are: 'create','update','delete','retrieve'(a few methods e.g. per category or severity, sorted by date, paginated). Actors/users can be humans or machines. As a sample: event producer can generate events and send them to topic/queue. Event consumers can be users who monitor events via dashboards. WebSocket clients can setup tcp connection from browser to topic/queue consumers and listen to new events in real time. Consumers can be also CEP(Complex Event Processing) systems. They can detect predetermined patterns (e.g. 10 ERROR events from one source during 1 minute) and generate alerts(events themselves). Some sort of AI/machine learning can be applied too if time allows. Another area is BPM (Business Process Management). They can be launched by events (or particular steps triggered) or produce events. So only imagination(and author lifespan) limits cases where events can be explored.

### 1.1. Language/frmk combinations

Current selection snapshot:

```
(lang: Java SE v7) (frmk: JDBC v4+ | JPA v2+ | Spring v4+ | Hibernate v4+ | Mybatis v3+)
(DB: SQL: (derby v10+ | mysql v5+| pgsql v9+) NoSQL: (Cassandra 2+ | Solr 4.7+)) (config: XML | Annotations).
```

TODO: Consider Spring Data for persistence.

You can ask why for DBs there are no e.g. Oracle, IBM DB2. They are solid, matured DBs. For now they are ignored for the sake of DB setup simplicity(e.g. try to find oracle for win7 64x machine which doesn't eat a few gigs of your HDD). Only immensely popular, easily available and open source DBs are selected.

TODO: Well, it would be fair to add at least one of big guys DB (Oracle?) and stored procedures via JDBC can be explored.

Side notes:

- It's incredible hard to embrace all combinations, so only some "mainstream" will be selected.
- "Container less" variations are considered only. JPA is the closest to JavaEE though. It's selected to avoid any clutter of additional configuring and dependencies from any EE container. Despite of it those testing DAOs will likely be used in EE containers for testing web-middleware-backend path.
- Some of frmk-s can be combined in "chains" e.g. (JPA + Spring + Hibernate). Folders and odcs will help to determine which combination is supported.
- No embedded DBs/modes are chosen. They are perceived as too far from PROD ready apps. Apache Derby (or JavaDB as modern name) has embedded mode. You can test it if you wish.

Current iteration (Mar 2014) already embraced planned SQL and NoSQL DBs persistence. Next iterations can embrace e.g.:

- (JavaEE: version: (v6 | v7) ; profile: (web | full))
- (lang: (Java SE 7 | JVM: (groovy v2+ | scala)) | python | perl | MS.Net: (c#))
- (DB: SQL: (Oracle)) (NoSQL: (MongoDB | Neo4j)) (idea is to embrace major NoSQL DB types: key/value or columnar, document, graph)
- 2nd level caching (notice ehcache is already used in a few persistence combinations)

### 1.2. Traits to consider/embrace in selected persistent frmk-s

- How easy to code/config a solution.
- Transaction(TX) support for ACID DBs(mostly SQL, but e.g. graph DB neo4j supports ACID too).
- XA distributed TX support (2 phase commits; JavaEE container is required).
- Date handling support (on very basic level; as a sample of not totally primitive data type).
- PK support. Client and DB side PK generation. Interesting case is complex keys support. Many legacy databases have composite keys.
- How easily Object/Relational mismatch is handled(those inheritance hierarchies).
- "Reconfigurability" (especially for env specific params).
- Pagination support (can be critical for large storage).
- Lazy loading (again critical for big data).
- Large data types support(for now long string is enough; e.g. via Clob or longvarchsr).
- Performance: latency, throughput.
- Memory consumption.
- Multi-threading support(how frmk-s sustain concurrent use; mutasker external lib can be used for load tests).
- Security (secured passwords, connections etc. Can be too large topic for this small project).
- Caching(consider *ehcache* as a sample).
- i18n(e.g. using UTF-8 charset for data).

### 1.3. Sample test cases for persistent frmk-s

- Create event
- Update event fields
- Retrieve event by ID
- Retrieve limited amount of recent events(useful for feeds)
- Retrieve events for period (check cases where some or all date params are null)
- Delete event by ID

### 1.4. Implemented approaches for persistent frmk-s

Base dir is **projects/java_se7_sql/approaches**.

| Approach | Module path |
|---|---|
| Cassandra/CQL | cassandra/cql |
| Hibernate/JPA/Annotations | hibernate/jpa/annot |
| Hibernate/Spring/Annotations | hibernate/spring/annot |
| Hibernate/Spring/JPA/Annotations | hibernate/spring/jpa/annot |
| Hibernate/Spring/XML | hibernate/spring/xml |
| Hibernate/XML | hibernate/xml |
| JDBC | jdbc |
| MyBatis/Annotations | mybatis/annot |
| MyBatis/Spring/Annotations | mybatis/spring/annot |
| MyBatis/XML | mybatis/xml |
| Solr/Direct | solr/direct |
| Spring/Templates | spring/templates |

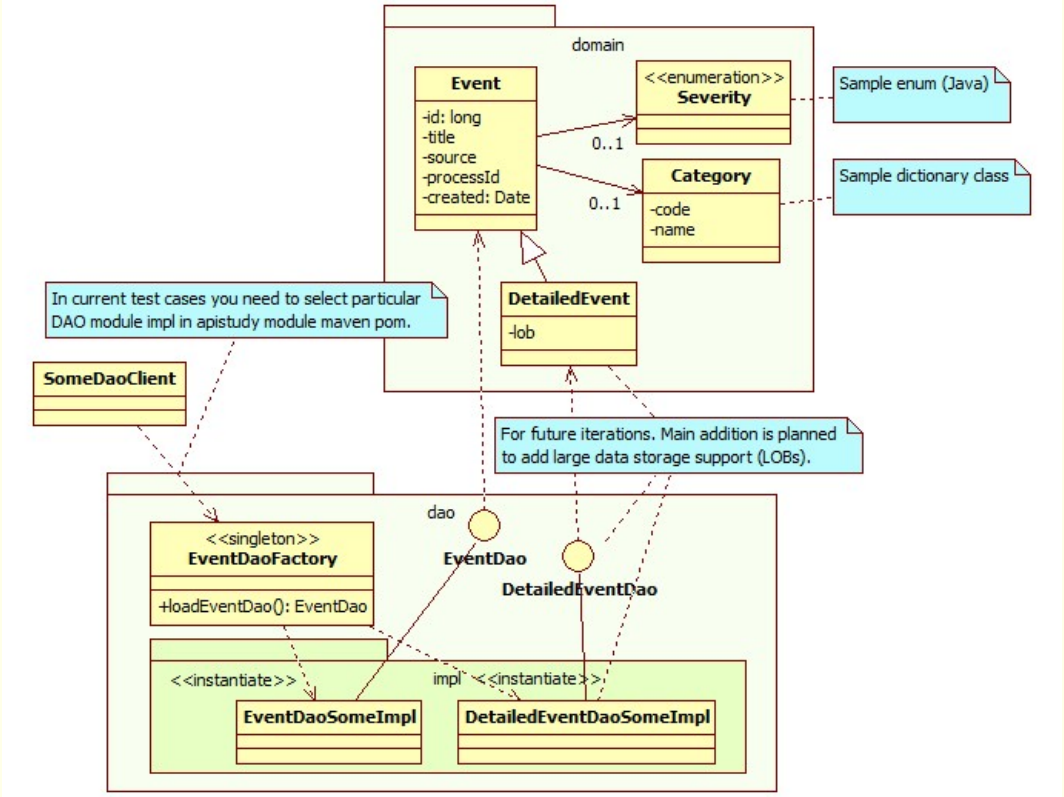
It's expected for codebase to show some of them, but not all. Each module should describe what feature set is implemented.

## 2. Design

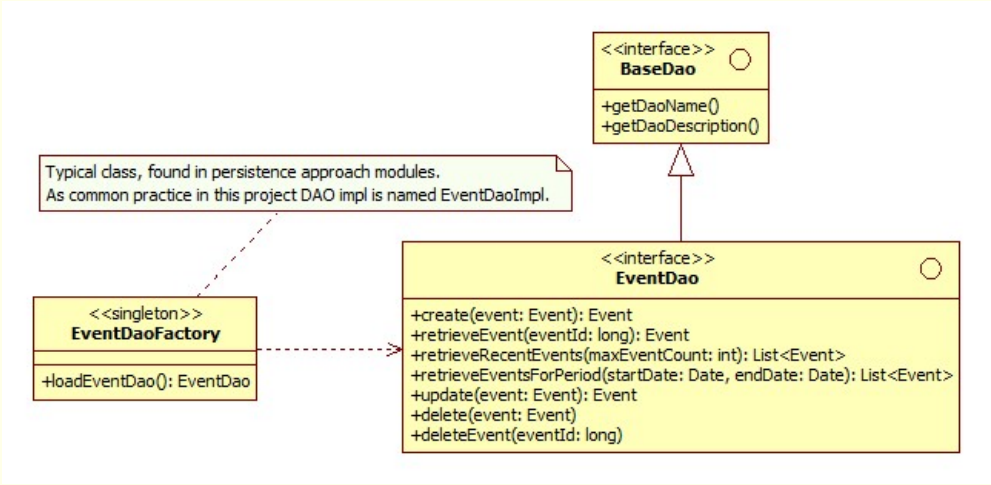
Main idea of design is to create easily 'buildable' mini projects frmk which allows to quickly add and test frmk-s combination. To simplify at least one aspect of system entities are planned to be used exactly the same and considered as "frozen".

### 2.1. Class high level structure

Domain and DAO classes:




DAO interfaces - more details:




#### 2.1.1. POJOs

Main POJO classes to play with are:

```
Event(id: long, category: Event.Category, severityLevel: enum(INFO/WARN/ERROR/FATAL)
      , source: string, processId: string, title: short_string, created: date_time).
```

Supplement entity is dict of event categories:

```
Event.Category(code: string, name: string).
```

These classes are suitable for lists and eager relationships mapping.
For more detailed info subclasses are to be used(likely in the next iteration):

```
DetailedEvent extends Event (info: long_string).
DetailedEvent.Category extends Event.Category(desc: long_string).
```

They are expected to be used only for individual entity editing. Sample:

```
Event(id: 123439887656, categoryCode: "START", severityLevel: "INFO", source: "server1"
      , processId: "PRC2387613", title: "System 123 get started", created: "2014-01-13H03:31:12.349Z").
Event.Category(code: "START", name: "Start")
Event.Category(code: "INTERACT", name: "Interaction")
Event.Category(code: "STOP", name: "Stop")

DetailedEvent(id: 123439887656, categoryCode: "START", severityLevel: "INFO", processId: "PRC2387613"
          ,  title: "System 123 get started" , created: "2014-01-13H03:31:12.349", info: "E.g. XML with details of start.").

DetailedEvent.Category(code: "START", name: "Start", desc: "Activity started.")
DetailedEvent.Category(code: "INTERACT", name: "Interaction", desc: "Interaction performed.")
DetailedEvent.Category(code: "STOP", name: "Stop", desc: "Activity started", desc: "Activity stopped.")
```

Practicality of this entity set in everyday life is doubtful, but at least some interesting use cases can be imagined in future iterations. For instance JMS queue can receive log message from

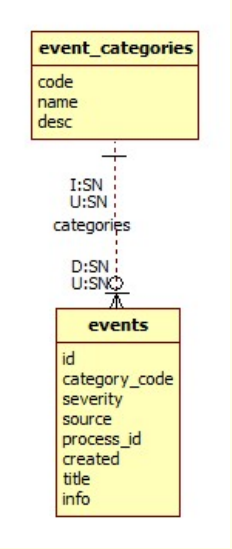
some remote system and transactionally store in DB(XA 2 phase commit). It's easy to imagine millions of records for events to check performance(reading/writing), as another case. Another "typical" case is workflow. A field ${processId} can be used to group events for some business process. A field ${source} can be process name. CEP rules can be implemented somewhere. Sample rule: suspended case after series of 'INTERACT' events. Warning about this sleeping process can be sent (e.g. as email, sms, or new event with severityLevel = 'WARN').

#### 2.1.2. DAOs (or Mappers in ORMs)

Core DAO is **EventDao**. Event categories can have own DAO if time allows.

### 2.2. Data high-level structure

To maximize simplicity for ORM inheritance hierarchy '*1 table per class hierarchy*' approach is selected. Tables can look like:




```
Table event_categories(code varchar(16) {PK}, name varchar(100) {UNQ}, description longvarchar).
Table events(id long {PK}, category_code varchar(16) {FK}, severity varchar(5)
       , source varchar(100), process_id varchar(100), title varchar(100), created timestamp, details longvarchar).
```

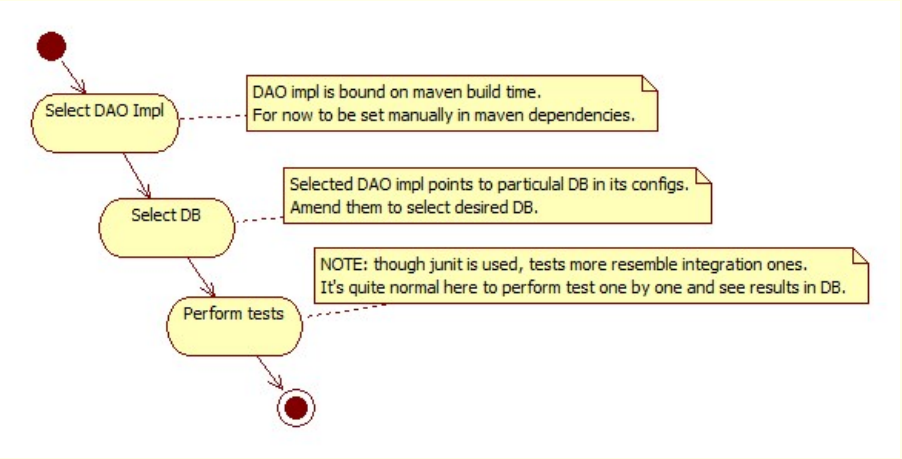
### 2.3. Project structure

Project is organised as directory structure with hierarchy of selected frmk names and approaches. For instance if we need to checkout how 'Java SE 7' + Spring JDBC templates, path to impl module could be: */java_se7_sql/approaches/spring*. Root dirs e.g. 'java_se7_sql' can be considered as "autonomous" group of modules. They can contain subtrees of modules with some reusable members (e.g. commons, api).

NOTE: If tree looks too "deep", flattening can be applied. Sample - 'java_se7_sql' (it could be e.g. 'lang/java/se/7/sql').

### 2.4. Build

For now main build tool is **maven** v3+, but gradle is in considerations too.
Typical command is '**mvn -e clean test**'. OS specific scripts will be provided for win7+(and maybe cygwin) and linux(sh) for now. They should be consider as hints/tips and main artefacts are DDL/DML files. For DBs basic setup instructions can be found in corresponding **configs** subdirs. DDLs and DMLs with initial data used in tests were added as e.g. SQL scripts. For example, if you need to create ETR tables in Postgresql, check **configs/sql_db_samples/pgsql/sql/create_tables.sql**.




## 3. Persistence DAO approaches overview

Java persistence has handful major paths and a lot of impl frmk-s. The lowest level is JDBC. It has 4 driver types. This project will embrace type 4 (pure Java driver). One step above of abstraction is JPA spec. It gradually replaced JEE entity beans. You'll see a few impls here. ORM allows to move abstraction up and developers can (theoretically) focus in Java dev rather than diving in SQL. In reality many business DB structures, especially legacy ones, have so "ORM incompatible" features (de-normalization, absence of OO inheritance support), enterprise dev-s still need to dive on SQL (and/or stored procedures) level.
Because there are tons of docs about persistence around, only short descriptions of approaches are provided below. Key config artefacts, classes, sample code snippets for one particular DAO operations (e.g. retrieve event) are provided. Code, context specific readme files and configs are considered here as detailed actionable docs.

JDBC connection config note: pretty often you can find **db.properties** file in resources folder. It has sample connection config info e.g. for derby:

```
# JDBC connection:
jdbc.driver=org.apache.derby.jdbc.ClientDriver
jdbc.url=jdbc:derby://localhost:1527/eventium
jdbc.username=etracker
jdbc.password=some_password
```

For other DBs you only need to copy/paste their specific values from similar db* files.

OK, let's get started.

### 3.1. JDBC

Class **EventDaoImpl** uses **DbConnectionFactoryI**.It, in turn, goes for DB settings to **db.properties** via **DbConnectionSettingsHolder**. For 1 time category loading **EventCategoryCache** is used. For now no 2nd level caching is used to make things simple.
Code fragment for **retrieveEvent**:

```
String queryString = "select id, title, category_code, severity, source, process_id, created from events where id = ?";
try (PreparedStatement statement = connection.prepareStatement(queryString,
  ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY)) {
 statement.setLong(1, eventId);
 ResultSet resultSet = statement.executeQuery();
 boolean firstExist = resultSet.first();
```

Notice Java SE7 style for resource allocation.

### 3.2. MyBatis

MyBatis (former iBatis) is ORM suitable to deal with complex DB structures (especially legacy). Dev-s need to know SQL. But it means more power and freedom in finding balanced OO-DB mappings.

#### 3.2.1. MyBatis XML

XML is classical config approach. File **mybatis-config.xml** has link to DB properties and mappers. Fragment:

```
<configuration>
    <properties resource="com/meriosol/etr/dao/db.properties" />
    <mappers>
        <mapper resource="com/meriosol/etr/dao/mapper/EventMapper.xml"/>
    </mappers>
```

File **EventMapper.xml** stores SQL expressions e.g. for event retrieval:

```
<mapper namespace="com.meriosol.etr.dao.mapper.EventMapper">
 <sql id="eventsTableName">events</sql>
 <sql id="eventCategoriesTableName">event_categories</sql>
 <sql id="baseEventFields">title, category_code, severity , source , process_id, created</sql>

 <sql id="joinedSelectBaseForEvents">select e.id as "event_id", e.title as "event_title", ec.code as "category_code",
  ec.name as "category_name",
  e.severity as "event_severity", e.source as "event_source",e.process_id as "event_process_id", e.created as
  "event_created"
  from
  <include refid="eventsTableName"/>
  e left join
  <include refid="eventCategoriesTableName"/>
  ec on e.category_code = ec.code
 </sql>
 ...
 <select id="retrieveEvent" resultMap="eventResultMap">
  <include refid="joinedSelectBaseForEvents"/> where id = #{id}
 </select>
 <resultMap id="eventResultMap" type="Event">
  <id property="id" column="event_id"/>
  <result property="title" column="event_title"/>
 ...
```

Interface **EventMapper** has corresponding method **Event retrieveEvent(Long eventId)**. MyBatis uses this method name to look through in mapper XML for SQL. Once query get executed, resultset is mapped to Event POJO. File **ehcache.xml** has terracota ehcache config. Fragment:

```
<ehcache>
    <diskStore path="/tmp/etr/ehcache" />

    <defaultCache maxElementsInMemory="10000" eternal="false"
                  timeToIdleSeconds="120" timeToLiveSeconds="120" overflowToDisk="true"
                  diskSpoolBufferSizeMB="30" maxElementsOnDisk="10000000"
                  diskPersistent="false" diskExpiryThreadIntervalSeconds="120"
                  memoryStoreEvictionPolicy="LRU"/>
```

In order to load MyBatis classes DAO impl uses:

```
    ... inputStream = Resources.getResourceAsStream(MYBATIS_RESOURCE_CONFIG);
        this.sessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
```

Retrieve event then looks like:

```
    public Event retrieveEvent(Long eventId) {
        Event event;
        try (SqlSession session = this.sessionFactory.openSession(TransactionIsolationLevel.SERIALIZABLE)) {
            EventMapper eventMapper = session.getMapper(EventMapper.class);
            event = eventMapper.retrieveEvent(eventId);
            session.commit();
        }
```

```
        return event;
    }
```

### 3.2.2. MyBatis Annotations

For annotation use config **mybatis-config.xml** references to POJO classes e.g.:

```
    <mappers>
        <mapper class="com.meriosol.etr.dao.mapper.EventMapper"/>
        <mapper class="com.meriosol.etr.dao.mapper.EventCategoryMapper"/>
    </mappers>
```

Our familiar retrieve event config moves to that mapper interface:

```
    @Select(DmlCommands.RETRIEVE_EVENT)
    @Options(useCache = true)
    @MapKey("id")
    @Results({
            @Result(property = "id", column = "id"),
            @Result(property = "category", column = "category_code", javaType = Event.Category.class
                    , one = @One(select = "com.meriosol.etr.dao.mapper.EventCategoryMapper.retrieveEventCategory")),
            @Result(property = "title", column = "title"),
            @Result(property = "severity", column = "severity"),
            @Result(property = "source", column = "source"),
            @Result(property = "processId", column = "process_id"),
            @Result(property = "created", column = "created")
    })
    Event retrieveEvent(Long eventId);
```

For caching annotation is used:

```
    @CacheNamespace(implementation = org.mybatis.caches.ehcache.EhcacheCache.class)
```

### 3.2.3. MyBatis Annotations with Spring DI

Spring can be used to simplify config. File **applicationContext.xml** has MyBatis config:

```
    <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="dataSource" ref="dataSource"/>
        <property name="typeAliasesPackage" value="com.meriosol.etr.domain"/>
    </bean>

    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
        <property name="basePackage" value="com.meriosol.etr.dao.mapper"/>
    </bean>
```

Mapper class is autowired:

```
    @Service("EventDaoService")
    public class EventDaoImpl implements EventDao {
        private static final Logger LOG = LoggerFactory.getLogger(EventDaoImpl.class);
        ...
        @Autowired
        private EventMapper eventMapper;
```

### 3.3. Spring JDBC templates

Spring context config fragment:

```
    <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>
    <context:component-scan base-package="com.meriosol.etr.dao.impl"/>
    <context:annotation-config/>
    <tx:annotation-driven/>
```

DAO impl is annotated as service. Spring finds it via component scanning. Dependencies:

```
@Service("EventDaoService")
public class EventDaoImpl implements EventDao {
    private NamedParameterJdbcTemplate namedParameterJdbcTemplate;
    private JdbcTemplate jdbcTemplate;
    private EventCategoryCache eventCategoryCache;
```

Class **NamedParameterJdbcTemplate** is useful for cases when named param map is preferable way to set params. Event load method uses this template:

```
    interface DmlCommands {
        String BASE_SELECT = "select id, category_code, title, severity , source , process_id, created " +
                "from events ";
```

```
        String RETRIEVE_EVENT = BASE_SELECT + " where id = :id";
....
    @Transactional
    public Event retrieveEvent(Long eventId) {
        Map namedParameters = Collections.singletonMap("id", eventId);
        try {
            return this.namedParameterJdbcTemplate.queryForObject(DmlCommands.RETRIEVE_EVENT
     , namedParameters, new EventMapper());
        } catch (EmptyResultDataAccessException e) {
            return null;
        }
    }
```

### 3.4. Hibernate

Hibernate ORM is immensely popular, de-facto standard for cases where you can manage DB structure, adjust it to domain class hierarchies. Ideas from it lead to Java EE JPA spec. SQL like lang - HQL is used to describe queries.

#### 3.4.1. Hibernate XML

Main config file for Hibernate / XML approach is **hibernate.cfg.xml**. In its session factory element there are mapper config links:

```
<hibernate-configuration>
    <session-factory>
        <!-- Database connection settings -->
        <property name="connection.driver_class">org.apache.derby.jdbc.ClientDriver</property>
        <property name="connection.url">jdbc:derby://localhost:1527/eventium</property>
        <property name="connection.username">etracker</property>
        <property name="connection.password">some_password</property>
  ...

        <mapping resource="com/meriosol/etr/dao/hibernate/Event.hbm.xml"/>
        <mapping resource="com/meriosol/etr/dao/hibernate/EventCategory.hbm.xml"/>
    </session-factory>
```

Notice DB config settings are right there. I didn't find way to extract them into "traditional" **db.properties**.
Mapper file **Event.hbm.xml** fragment:

```
<hibernate-mapping package="com.meriosol.etr.domain">
    <class name="Event" table="events">
        <id name="id" type="long" column="id">
            <generator class="native"/>
        </id>

        <many-to-one name="category" class="Event$Category" column="category_code"
                     cascade="save-update" not-null="false" lazy="false" fetch="join"/>
        <property name="severity" column="severity">
            <type name="org.hibernate.type.EnumType">
                <param name="enumClass">com.meriosol.etr.domain.Event$Severity</param>
            </type>
        </property>
...
```

This mapper binds domain class **Event** with table **events**. For ID value generation native DB vendor feature is to be used.
Many-to-one relationship for field **category** is bound as **events.category_code** to **Event.Category** (Event$Category in config because '$' should be used for static nested classes) and **event_categories.code** DB field (as described in **EventCategory.hbm.xml**).
DAO impl class has reference to session factory that initialized like this:

```
Configuration configuration = new Configuration();
configuration.configure();
ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder().applySettings(configuration.getProperties()).build();
this.sessionFactory = configuration.buildSessionFactory(serviceRegistry);
```

Fragment of event retrieval:

```
session = this.sessionFactory.openSession();
transaction = session.beginTransaction();
event = (Event) session.get(Event.class, eventId);
transaction.commit();
```

#### 3.4.2. Hibernate JPA

Quite interesting combination. JPA spec was mainly based on ideas from different ORMs including Hibernate. So, what we have as a result (at least on basic level), let's see below.
Main config file **persistence.xml** has mappers(annotation approach here) and DB connection settings:

```
<persistence-unit name="com.meriosol.etr.dao.jpa">
        <class>com.meriosol.etr.dao.entity.EventEntity</class>
        <class>com.meriosol.etr.dao.entity.EventCategoryEntity</class>

        <properties>
            <property name="javax.persistence.jdbc.driver" value="org.apache.derby.jdbc.ClientDriver"/>
```

Because this time annotation approach is used and by system it's paramount to remain domain POJOs intact, artificial entities were created. Fragment of EventEntity:

```
@Entity
@Table(name = "events")
public class EventEntity {
    @Id
    @Column(name = "id", insertable = true, updatable = false)
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "category_code", nullable = true, insertable = true, updatable = true)
    private EventCategoryEntity category;

    @Column(name = "severity", insertable = true, nullable = true, updatable = true)
    @Enumerated(EnumType.STRING)
    private Event.Severity severity;
 ...
```

Some concerns regarding annotations:

- One of concerns is using changeable names in annotations. What if some of DBs has table name **mucho_eventos** for example?
- Another concern is how not to get lost in annotated POJOS in really big enterprise systems. Especially without IDE support.
- When configs are spread around classes it's a bit hard to troubleshoot / fix in non-trivial systems(especially with legacy DBs).

Factory **EventDaoFactory** init-s **javax.persistence.EntityManagerFactory**:

```
 ...
    private static final String PERSISTENT_UNIT_NAME = "com.meriosol.etr.dao.jpa";
 ...
    private EventDaoFactory() {
        this.entityManagerFactory = Persistence.createEntityManagerFactory(PERSISTENT_UNIT_NAME);
    }
```

DAO impl then uses it e.g. for event loading:

```
 entityManager = this.entityManagerFactory.createEntityManager();
 transaction = entityManager.getTransaction();
 transaction.begin();

 CriteriaBuilder cb = entityManager.getCriteriaBuilder();
 CriteriaQuery cq = cb.createQuery(EventEntity.class);
 Root eventEntityRoot = cq.from(EventEntity.class);
 cq.select(eventEntityRoot);
 String pkFieldName = "id";
 String idParamName = "id";
 cq.where(cb.equal(eventEntityRoot.get(pkFieldName) , cb.parameter(Long.class, idParamName)));

 Query query = entityManager.createQuery(cq);
 query.setParameter(idParamName, eventId);

 try {
  eventEntity = (EventEntity) query.getSingleResult();
 } catch (NoResultException e) {
  LOG.info("No entity found for eventId='{}'. In some cases it can be normal", eventId);
 }

 transaction.commit();
```

Well, JPA criteria API is type safe, but a heck verbose, even for simple cases. I had hard time to figure out how to build queries for more advanced ones. Let's call Spring for help.

### 3.4.3. Hibernate Spring

Hibernate+Spring is popular de-facto combination for light-weight(vs entity beans / JavaEE) and often "container-less" variants.

#### 3.4.3.1. Hibernate Spring XML

In Hibernate samples above you could notice DB connection info had to be set in hibernate xml configs. Spring relieves this config "tension" and now **db.properties** get back to scene. Along with connection settings it has hibernate specific e.g.:

```
hibernate.hbm2ddl.auto = false
hibernate.dialect = org.hibernate.dialect.DerbyTenSevenDialect
hibernate.cache.provider_class = org.hibernate.cache.internal.NoCacheProvider
```

Spring **applicationContext.xml** points to this file via placeholder way:

```
    <context:property-placeholder location="com/meriosol/etr/dao/db.properties"/>
```

Hibernate session factory in "Spring" style:

```
<bean id="sessionFactory"
   class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
 <property name="dataSource" ref="dataSource"/>
    <property name="mappingResources">
        <list>
            <value>com/meriosol/etr/dao/hibernate/Event.hbm.xml</value>
            <value>com/meriosol/etr/dao/hibernate/EventCategory.hbm.xml</value>
        </list>
    </property>
```

```
    ....
```

DAO impl is loaded from app context:

```
    ApplicationContext context = new ClassPathXmlApplicationContext("com/meriosol/etr/dao/applicationContext.xml");
    return (EventDao) context.getBean("EventDaoService");
```

SessionFactory is autowired in this DAO impl:

```
@Service("EventDaoService")
public class EventDaoImpl implements EventDao {
...
    @Autowired
    private SessionFactory sessionFactory;
    ...
```

All this "jazz" makes event loading deadly simple:

```
    @Transactional
    public Event retrieveEvent(Long eventId) {
        Session session = this.sessionFactory.openSession();
        return (Event) session.get(Event.class, eventId);
    }
```

### 3.4.3.2. Hibernate Spring Annotations

For annotated classes one small amendment is needed in **applicationContext.xml** - package scanner:

```
<bean id="sessionFactory"
   class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
 <property name="dataSource" ref="dataSource"/>
 <property name="packagesToScan">
     <list>
           <value>com.meriosol.etr.dao.entity</value>
        </list>
    </property>
```

DAO impl is loaded the same way as in XML approach and SessionFactory is autowired the same way. Event load:

```
    @Transactional(readOnly = true, propagation = Propagation.REQUIRED)
    public Event retrieveEvent(Long eventId) {
        Session session = this.sessionFactory.getCurrentSession();
        EventEntity eventEntity = (EventEntity) session.get(EventEntity.class, eventId);
        return EventEntityTransformUtil.transform(eventEntity);
    }
```

**EventEntityTransformUtil** used to transform **EventEntity** to basic (annotation free) **Event**.

### 3.4.3.3. Hibernate Spring Annotations JPA

This combination is even more incredible than those above. It's where established de-facto meet specs and new additions.
To avoid repeating the same as above (see Hibernate/Spring) let's pay attention to differences. Major difference is using **EntityManagerFactory** in **applicationContext.xml**:

```
 ...
<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
 <property name="dataSource" ref="dataSource"/>
 <property name="persistenceUnitName" value="jpaTest"/>
 <property name="packagesToScan">
  <list>
   <value>com.meriosol.etr.dao.entity</value>
  </list>
 </property>
 <property name="jpaVendorAdapter">
  <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
  </bean>
 </property>

   <bean id="entityManager" class="org.springframework.orm.jpa.support.SharedEntityManagerBean">
        <property name="entityManagerFactory" ref="entityManagerFactory"/>
    </bean>
 ...
```

That entity manager is autowired in DAO impl:

```
@Service("EventDaoService")
public class EventDaoImpl implements EventDao {
    @Autowired
    private EntityManager entityManager;
```

Event load uses this EntityManager:

```
@Transactional(readOnly = true, propagation = Propagation.REQUIRED)
public Event retrieveEvent(Long eventId) {
 CriteriaBuilder cb = this.entityManager.getCriteriaBuilder();
 CriteriaQuery cq = cb.createQuery(EventEntity.class);
 Root eventEntityRoot = cq.from(EventEntity.class);
 cq.select(eventEntityRoot);
 String pkFieldName = "id";
 String idParamName = "id";
 cq.where(cb.equal(eventEntityRoot.get(pkFieldName), cb.parameter(Long.class, idParamName)));

 Query query = this.entityManager.createQuery(cq);
 query.setParameter(idParamName, eventId);

 EventEntity eventEntity = null;
 try {
  eventEntity = (EventEntity) query.getSingleResult();
 } catch (NoResultException e) {
  LOG.info("No entity found for eventId='{}'. In some cases it can be normal", eventId);
 }
```

### 3.5. NoSQL DBs

So far persistence and ORMs described access to relational(or "SQL") DBs. Because of growing popularity of alternative NoSQL world it would be strange not to touch this area. Cassandra and Solr were selected because I already used them in some projects. But likely other flavours will be added(e.g. MongoDB with Json storage, Neo4J for graphs).

#### 3.5.1. NoSQL Cassandra

Cassandra allows to distribute its keyspaces and "tables"(column families) across many machines with P2P replication with eventual consistency(so don't expect "real-time" one). Let's use 1 host for simplicity now. Familiar config **db.properties** has now cassandra connection specific properties:

```
cassandra.hosts=127.0.0.1
cassandra.keyspace=eventium
cassandra.username=etracker
cassandra.password=some_password
```

DAO impl has a few helpers:

```
public class EventDaoImpl implements EventDao {
    private DbSessionFactory dbSessionFactory;
    private EventCategoryCache eventCategoryCache;
    private EventByCreatedSliceCrud eventByCreatedSliceCrud;
```

**DbSessionFactory** loads Cassandra connection via cluster:

```
Session obtainNewSession(String hosts, String keyspace, String username, String password) {
 String[] hostsArray = hosts.split(",");
 Cluster cluster = Cluster.builder()
   .addContactPoints(hostsArray).withCredentials(username, password)
   .withRetryPolicy(DowngradingConsistencyRetryPolicy.INSTANCE)
   .withReconnectionPolicy(new ConstantReconnectionPolicy(100L))
   .build();
 Metadata metadata = cluster.getMetadata();
 return cluster.connect();
}
```

Event load code fragment shows how this session is used:

```
 ...
 String selector = "SELECT " + EVENT_COLUMNS
   + " FROM " + getFullEventTableName(keyspace) + " WHERE id = ? LIMIT 1;";
 PreparedStatement statement = session.prepare(selector);
 BoundStatement boundStatement = statement.bind(eventId);
 ResultSet results = session.execute(boundStatement);
 Row eventRow = null;
 if (results != null) {
  eventRow = results.one();
 }
```

Class **EventByCreatedSliceCrud** is used to load event IDs sorted according to rule "show me the latest first", which is typical for events.
Tricky thing about column families design is they should be adjusted to particular use cases. For instance in order to load events with sorted severity level, another "table" has to be created and own CRUD supported. I suspect it can make SQL DB dev-s crazy.

#### 3.5.2. NoSQL Solr

Solr is distributed reverse indexer of documents(e.g. articles/news). Lucent engine is used for search in indices (Solr cores, almost "tables").
Solr core has fields which can be indexed or used as (meta)data. Textual fields is only one data type sample supported by engine. As with the most previous approaches, we have config **db.properties**. But with Solr specific properties e.g.:

```
# Solr connection:
solr.url=http://localhost:8983/solr

# Solr cores:
solr.core.event categories=event categories
```

```
solr.core.events=events
```

DAO impl uses these helpers:

```
public class EventDaoImpl implements EventDao {
    private SolrServerFactory solrServerFactory;
    private EventCategoryCache eventCategoryCache;
```

Class **SolrServerFactory** "knows" how to get connection for particular Solr core. Inner class **SolrServerHolder** stores **Map<SolrCoreCode, SolrServer> servers**. Solr server is loaded by instantiation of **HttpSolrServer**:

```
    HttpSolrServer httpSolrServer = new HttpSolrServer(this.settings.getSolrCoreUrl(solrCoreCode));
```

Event solr document load code fragment:

```
    String queryString = "id:" + eventId;
    SolrQuery solrQuery = new SolrQuery();
    solrQuery.set("q", queryString);

    Event event = null;
    QueryResponse response = null;
    try {
        response = solrServer.query(solrQuery);
        if (response != null) {
            SolrDocumentList results = response.getResults();
            if (results != null && results.size() > 0) {
                if (results.size() > 1) {
                    throw new EtrException(String.format("Too many events found for ID='%s'!", eventId));
                }
                SolrDocument solrDocument = results.get(0);
                event = buildEventFromResultSet(solrDocument);
            }
        }
    } catch (SolrServerException e) {
        throw new EtrException(e);
    }
...
```

For now(April 2014) this is last approach to tackle. If time allows, more ones will be added.

### 3.6. Performance

Performance is quite big topic itself (along with non-functional such as security, maintainability/extensibility, availability, reliability/durability, flexibility etc). In current state of project minimal attention was paid to enhance performance. Only occasionally connection poolers are used, for instance. In some future statement execution latency, traffic throughput (size, speed) for statements needs to be explored and in case of bottlenecks fixed. But for beginning all these executions need to be measured.

TODO: Performance comparison would be nice to see(especially in multi-threaded scenarios).

## 4. Disclaimers

This section describes what this project is NOT about and what to be aware of while perusing its artefacts.

**Code and configs are for quick start in particular combination of frmk-s only. Production ready code definitely requires more polishing, e.g. "what if"s handling, NPE situation catching etc.**

**There is no goal to show all advanced features of used frmk-s. More specific projects to be created if time allows.**

**In production systems reusability is one of key non-functional requirements. This project follows this practice only partially as "side feature" because main goal is to just show working basic combination, without forcing learner to jump through all hoops/levels of dependencies. Code is expected to be repeated, with reasonable level of sanity. Consider subprojects as relatively independent modules.**

**Pragmatism(and a sort of laziness) is strong force to have *good enough* amount of frmk combinations. Missed combinations are hopefully can be figured out from provided ones.**

**This project assumes you have basic knowledge in back-end development area, familiar at least with Java, SQL.**

**At least from beginning only the most popular modern and stable frmk-s versions to be selected. Elder versions maybe get addressed when project itself comes to age. Funny part is: these now modern versions will be old soon.**

**Project is in status "pet PoC, to play around", can be removed at any time without warning.**

## 5. Abbreviations

| Abbrev | Definition |
|---|---|
| frmk | framework |
| config | configuration |
| CRUD | Create/Read/Update/Delete |
| NOP | Null Pointer (exception) |
| OSS | Open Source Software |
| dict | dictionary |
| DAO | Data Access Object |
| ORM | Object Relational Mapping |
| DB | database |
| DI | Dependency Injection |
| PROD | Production |

| | |
|---|---|
| DEV | Development |
| TX | Transaction |
| ACID | Atomic Consistent Isolated Durable (about TX) |
| PK | Primary Key |
| FK | Foreign Key |
| env | environment |
| param | parameter |
| dir | directory |
| desc | description |
| CEP | Complex Event Processing |

| Version: 0.9 | Updated: 2014-04-02 | © MerioSol, 2014 |