1. Preamble

Developing enterprise software applications with security requirements can be a cumbersome and error-prone process. In such applications often the security is spread all over the application code. This makes it difficult to understand how things work and hard to maintain security in such code.

With model-driven development business logic should be expressed in the objects of your domain model. As for now there is no convenient way to express access control requirements through your domain model. Hence access control-code is normally written into the service layer or data-access layer.

JPA Security offers a way to express access control through your domain model and additionally supplies a solution to completely separate access control logic from business logic, improving performance and maintainability. JPA Security provides an interface to define security requirements of your domain model via configuration (Annotations or XML).

Due to it's smooth integration into current Java standards,it may be integrated into every layer of your application. It enables you to configure rules defining access control for your Entity Beans and Embeddables based on the current security context (i.e. the currently authenticated user and/or its roles in the application).

JPA Security uses the notion of security unit to refer to a set of Java bean classes and their corresponding access rules. In the current version of JPA Security a security unit directly corresponds to a persistence unit of JPA. The content of a security unit is defined at deployment time and may not change at runtime.

In contrast to the security unit , the security context starts at runtime with the authentication of a user. As of version 0.4.0 of JPA Security the content of this context is completely customizable and you can do so by implementing the SecurityContext interface. The security context contains information like i.e. the current user, its roles and/or the current tenant, but any kind of context information that is needed to specify the authorization of the current user to access beans may be made available through the security context.

JPA Security smoothly integrates with existing authentication solutions like specified in the servlet specification, with EJB or Spring Security. JPA Security may be configured to take the current user and its roles provided by this technologies and fill the security context with it.

2. Basics - Rules, Users and Roles

Authorization is the process of giving someone permission to do or see something. It is used to determine what the authenticated user is allowed to do in the application and what data he is allowed to see.

2.1. Organizing Authorization

There are different ways of organizing authorization. Widely used approaches are role-based authorization or access control lists.

- In role-based Authorization users are assigned to roles. Rights to do or get something are granted to these roles.
- In the concept of "Access Control Lists "an access control list is assigned to every object that has to be accessed. An access control list is a list of access control entries. Every of this entries holds a reference to a user or a role it is valid for, as well as the access rights that the referenced user or role has on the object.

2.2. Access Control: Authorization in Action

Access Control is the process of allowing and forbidding access to resources based on the authorization of the current security context. There are different levels of access control.

- Presentation left access control - Access control is implemented in the presentation tier of the application. Only data that the user is allowed to see is displayed and only buttons for actions that the user is allowed to do are displayed and enabled. Although it may be necessary to provide different views to users with different roles, handling access control only at the presentation layer can be a big problem, when it involves loading and transferring masses of data to the presentation layer, where it gets filtered out, because the current user is not allowed to see it.
- Service level access control - Access control is implemented in the service tier. Often this is realized using method-based access control. For every method in the service tier it is defined which users and/or roles are allowed to call it. Although this kind of security is widely supported by current security solutions like JavaEE Security and Spring Security it is often not the correct location of handling access control because access control usually depends more on the data used than on the actions executed. This leads to code that checks the current user and/or its roles to decide which data access tier method to invoke or this kind of checks are done by the code invoking the service methods. However manual handling of authorization informations leads to spreading the security all over the code. This makes changes to security requirements hard to handle.

- Class level access control - Using class level access control on the entity beans of an application leads to a more fine-graded access control mechanism. If you are able to define which entity types may be used by which user or role (and you can do this in EJB 3 with the @RolesAllowed annotation) you have a clean way of defining which user may see and/or change which data without manual handling of authorization information. However this level is not detailed enough in many cases.
- Instance level access control - With instance level access control for every user and/or role you can define permissions on an entity basis. Access Control Lists are widely used for this approach, but JPA Security provides a more flexible way to accomplish this, like we see later.
- Property level access control - A step further from instance level access control is property level access control, where you can define access restrictions on a property base. JPA Security currently only supports property level access control for properties of type @Embeddable by providing instance level access control for embeddables.

2.3. Access Control in JPA Security

In JPA Security the access to entities and embeddables is defined by access rules. There is one set of access rules per security unit. This set of access rules applies to every JPA query and every entity and embeddable you get out of an EntityManager of the persistence unit that corresponds to that security unit.

2.3.1. Example: Access Rule

The following rule restricts the read access to accounts, where the owner is equal to the current user. In other words: every user can read only its own accounts.

```
        GRANT READ ACCESS TO Account account WHERE account.owner =
CURRENT_PRINCIPAL
```

In the previous example the CURRENT_PRINCIPAL is provided by the currently active security context. When the access rule is evaluated, the CURRENT_PRINCIPAL is received from the security context and the rule is evaluated against this principal

2.3.2. Access Rules in JPA Security

JPA Security allows the definition of rules that grant create, read, update and/or delete access. Currently there is no explicit way to deny access. Although this is not needed, since every entity or embeddable to which no access is granted may not be accessed. An exception from this behavior are classes for which no access rule exists at all. Objects of such class may be accessed without any restriction.

2.3.3. The syntax of access rules

The general syntax of an access rule of JPA Security looks like GRANT [CREATE] [READ] [UPDATE] [DELETE] ACCESS TO entity_name alias [where_clause], where entity_name must be an entity or embeddable of the persistence unit (according to the JPA Specification, defaults to the class name if not otherwise specified) and the alias is an alias for that entity or embeddable that may be used in the where clause.

The syntax of the where_clause is derived from the syntax of WHERE clauses of JPQL, the query language of JPA. Within the clause any alias may be used that is defined by your current security context. The build-in security contexts define two aliases, which are CURRENT_PRINCIPAL and CURRENT_ROLES . The CURRENT_PRINCIPAL alias will be evaluated to the currently authenticated principal during runtime and the CURRENT_ROLES alias will be expanded to a list of roles that the current principal belongs to. No input parameters may be used in the WHERE clause of access rules. If you need more aliases to be defined (i.e. CURRENT_TENANT), you will have to implement your own security context like described later.

2.3.4. Providing access rules

With JPA Security there are two predefined ways to provide access rules: via XML configuration or via Annotations. In a later chapter we will see how to implement your own way of providing access rules.

2.3.4.1. Access rules via XML

One predefined way to provide access rules in JPA Security is via a file called security.xml , which is located in the META-INF directory of your application. Below is an example of the structure of such file:

```
<
security xmlns="http://jpasecurity.sf.net/xml/ns/security"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://jpasecurity.sf.net/xml/ns/security
```

```
                              http://jpasecurity.sf.net/xml/ns/security/security_1_0.xsd"
          version="1.0"
>
<
persistence-unit name="..."
>
<
access-rule
>
...
<
/access-rule
>

    ...

<
/persistence-unit
>
<
/security
>
```

2.3.4.2. Access rules via Annotations

The other predefined way to provide access rules in JPA Security is via Annotations. You may annotate your entity classes with one of the following two annotations: javax.annotation.security.RolesAllowed and net.sf.jpasecurity.security.rules.Permit .

Note that the semantics of the @RolesAllowed annotation slightly differs between the EJB Specification and JPA Security: If you annotate a class with the @RolesAllowed annotation this means for EJB any access to any method of an instance of that class will cause a SecurityException , if the current user is not in one of the roles allowed. JPA Security goes a step further: The current user will not retrieve this object from database if he is not in one of the roles allowed. JPA Security does not support the @RolesAllowed annotation at method-level.

The @Permit annotation has two optional parameters:

- With the parameter access you can specify which kind of access shall be granted by this rule. The value is an array of the AccessType enum. Possible values of an AccessType are CREATE , READ , UPDATe and DELETE .
- With the parameter rule you can specify a rule to restict the access for entities of the annotated class. The syntax of this rule is derived from the where -clause of JPQL (see previous section), where the special keyword this serves as an alias for the annotated entity or embeddable that may be used in the rule.

Example:

```
@Permit(access = AccessType.READ, rule = "owner = CURRENT_PRINCIPAL")
public class Account {
    ...
}
```

2.3.4.3. Applying of access rules

Read-access rules are applied to every entity or embeddable that is accessed via a JPA-Security-enabled EntityManager or via object-navigation through objects obtained from such EntityManager . When the object is accessed via JPQL, the access rules are applied directly to the JPQL-query, allowing the filtering to take place within the database. When the object is accessed via object-navigation, JPA Security tries to avoid database-calls and evaluates the rules in memory. When in-memory-evaluation is not applicable and the entity-manager is still open, a query is performed to evaluate the query. In the next section you can read, in which cases in-memory-evaluation is applicable and where not. When in-memory-evaluation is not applicable and the entity-manager is already closed, a SecurityException will be thrown.

Update-access rules are applied on flush() or commit() . Again the default-behaviour is in-memory-evaluation then, falling back to a query like described above.

Create-access rules and delete-access rules are applied when the appropriate action is performed with the entity-manager (either direct or by cascading). In-memory-evaluation applies like described above.

For all cases JPA Security is clever enough to apply the appropriate access rules for sub- and superclasses, too.

2.3.4.4. In-Memory-Evaluation

Every access rule that does not contain any sub-select can be evaluated in memory. For queries that contain sub-selects it depends on the kind of the sub-select and the content of the (first-level) entity-manager-cache of JPA Security. Sub-selects where all aliases from within the sub-select can directly replaced with an alias from outside can be evaluated in memory. Thus the following access rule can be evaluated in memory:

```
 GRAND ACCESS TO TestBean bean WHERE EXISTS (SELECT b FROM TestBean b
WHERE b = bean AND b.accessControlList = CURRENT_PRINCIPAL)
```

Since the alias from within the sub-select cannot directly replaced by the alias from outside, the following query cannot be evaluated in memory:

```
 GRAND ACCESS TO TestBean bean WHERE EXISTS (SELECT entry FROM
AccessControlListEntry entry WHERE bean.accessControlList = entry.accessControlList AND
entry IN (CURRENT_ROLES)
```

Although there can be no guarantee in general that the last query can be evaluated in memory, in-memory-evaluation can still be achieved by ensuring that the entities that are needed to evaluate the sub-select are contained in the (first-level) entity-manager-cache. For example the last rule can be evaluated in memory if there exists an AccessContolListEntry in the cache that meets the specified where-clause. Remember, that on persist the check will be done before the persist-operation is cascaded. So when you want to persist a TestBean that contains an accessControlList with entries that match the where-clause, you have to persist the entries before you persist the TestBean to ensure the entries are in the cache.

Authentication is the process of determining and verifying the identity of someone or something. In multi-user applications, the process of authentication for an application is needed to get knowledge about the person that is currently using the application. The widely used process for authentication is a login process during which the user is asked for his username and password. A user that knows one of this username/password-tuples is assumend to be the person belonging to that username. Besides username/password authentication there are other methods like public-key-authentication with a digital certificate, to name just one.

2.4. Authentication in JPA Security

JPA Security uses an implementation of the net.sf.jpasecurity.configuration.SecurityContext interface to be aware of the currently authenticated user and other related information (like its roles, tenant, ...) at runtime.

2.4.1. Default-Configuration for Authentication

By default JPA Security will try to auto-detect your security context. This is done via the indirection of an authentication provider. An authentication provider provides access to the current authenticated user and its roles. The detection follows the following rules (The first matching rule is taken):

1. When spring-security is in the classpath, the SpringAuthenticationProvider is used.
2. When java:comp/EJBContext is available in the JNDI-context, the EjbAuthenticationProvider is used.
3. When JSF is available in the classpath, the JsfAuthenticationProvider is used.
4. Otherwise the DefaultAuthenticationProvider is used.

2.4.2. Customizing Authentication

When the described auto-detection strategy does not work for your environment, you may specify the class name of any implementation of the net.sf.jpasecurity.configuration.SecurityContext interface as value of the persistence-property net.sf.jpasecurity.security.context in your persistence.xml . To provide backward compatibility to JPA Security 0.3 net.sf.jpasecurity.security.authentication.provider is also valid, if you specify a class name of an implementation of the net.sf.jpasecurity.configuration.AuthenticationProvider interface, but any specification of the net.sf.jpasecurity.security.context property will take precedence.

3. Getting started with JPA Security

In order to use JPA Security you have to enable it for your persistence unit. This can be done by modifying your "persistence.xml "to point to JPA Security. Additionally you have to configure JPA Security to use your original persistence provider.

Supposed you have an existing JPA application. Your persistence.xml may look similar to this:

```
<persistence 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"
             version="1.0">

  <persistence-unit name="your-persistence-unit-name" transaction-type="...">

    <provider>your.persistence.provider.ClassName</provider>

    <class>your.persistent.ClassName</class>
    <!-- More class-mappings go here -->
    ...

    <properties>
      <!-- persistence-provider-specific properties go here -->
      ...
    </properties>

  </persistence-unit>
</persistence>
```

After integrating JPA Security your "persistence.xml "may look like this:

```
<persistence 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"
             version="1.0">

  <persistence-unit name="your-persistence-unit-name" transaction-type="...">
```

```
    <provider>net.sf.jpasecurity.persistence.SecurePersistenceProvider</provider>

    <class>your.persistent.ClassName</class>
    <!-- More class-mappings go here -->
    ...

    <properties>
      <property name="net.sf.jpasecurity.persistence.provider"
                value="your.persistence.provider.ClassName" />
      <!-- persistence-provider-specific properties go here -->
      ...
    </properties>

  </persistence-unit>
</persistence>
```

We changed the <provider >tag to point to JPA Security's implementation of the PersistenceProvider interface and added the property net.sf.jpasecurity.persistence.provider to point to your original persistence provider. That is all you need to integrate JPA Security.

To use Java EE authentication information for JPA Security you may integrate JPA Security like described in the previous chapter and specify net.sf.jpasecurity.security.authentication.EjbAuthenticationProvider as authentication provider in your persistence properties. You need to define all the roles your application is using with the @DeclareRoles annotation at at least one of your entity beans in order to make the EjbAuthenticationProvider work correctly.

If you do not have a version of a spring-security.jar in your classpath usually you do not need to define your own authentication provider as the auto-detection mechanism of JPA Security should automatically install the EjbAuthenticationProvider .

To use Servlet authentication information for JPA Security within your JSF application you may integrate JPA Security like described in the previous chapter and specify net.sf.jpasecurity.jsf.authentication.JsfAuthenticationProvider as authentication provider in your persistence properties. You need to define all the roles your application is using with the @DeclareRoles annotation at at least one of your entity beans in order to make the JsfAuthenticationProvider work correctly.

If you do not have a version of a spring-security.jar in your classpath usually you do not need to define your own authentication provider as the auto-detection mechanism of JPA Security should automatically install the EjbAuthenticationProvider (if you are in a Java EE server) or the JsfAuthenticationProvider if you are using JSF outside a Java EE server, both using Servlet authentication information.

You need to have the jpasecurity-jsf module in your classpath in order to make the JsfAuthenticationProvider work.

If you are using the Spring-framework with Spring Security, you don't have to follow the chapter Getting started with JPA Security (also you can do and specify the net.sf.jpasecurity.security.authentication.SpringAuthenticationProvider in your persistence.properties).

The easiest way to integrate Spring-Security is to simply replace your LocalEntityManagerFactoryBean or LocalContainerEntityManagerFactoryBean in your Spring configuration with the appropriate counterpart in the net.sf.jpasecurity.spring.persistence package (which are SecureLocalEntityManagerFactoryBean and SecureContainerEntityManagerFactoryBean).

The autodetection mechanism of JPA Security will automatically detect that you are using Spring Security, but you can specify your security context or authentication provider as bean property of the Secure*EntityManagerFactoryBean in your spring configuration.

You need to have the jpasecurity-spring module in your classpath to use spring security.

If you neither use Java EE authentication nor Servlet authentication (via JSF) nor Spring-Security you either have to provide your own implementation of the net.sf.jpasecurity.configuration.SecurityContext interface or you have to use one of the build-in authentication providers, which are DefaultAuthenticationProvider and StaticAuthenticationProvider .

Both provide methods to authenticate users and roles and methods to apply runAs behavior. The DefaultAuthenticationProvider may be used in server-site applications where the authentication is on a per-thread-basis whereas the StaticAuthenticationProvider may be used on client-site applications where authentication per vm is intended.

You may take a look at the simple tutoral to see an example of using the StaticAuthenticationProvider . In the next chapter you will learn how to provide a custom security context or authentication provider.

4. Customization

JPA Security is configured via the persistence properties in your persistence.xml . Below is a list of persistence properties that are supported out of the box by JPA Security.

- net.sf.jpasecurity.persistence.provider - This is the only property that is required by JPA Security. It specifies the class name of the original persistence provider that shall be used by JPA Security to do the actual database access.
- net.sf.jpasecurity.security.authentication.provider - This property specifies the class name of the implementation of an authentication provider which may be any implementation of the interface net.sf.jpasecurity.configuration.AuthenticationProvider .
- net.sf.jpasecurity.security.context - This property specifies the class name of the implementation of a security context which may be any implementation of the interface net.sf.jpasecurity.configuration.SecurityContext .
- net.sf.jpasecurity.security.rules.provider - This property specifies the class name of the implementation of an access rules provider which may be any implementation of the interface net.sf.jpasecurity.configuration.AccessRulesProvider .

As stated before you may implement your own way of providing access rules (i.e. via JDBC).You have to implement the interface net.sf.jpasecurity.security.rules.AccessRulesProvider and specify the property net.sf.jpasecurity.security.rules.provider in your persistence.xml with the classname of your implementation of theinterface net.sf.jpasecurity.security.rules.AccessRulesProvider . Take a look at its javadoc documentation for further reference.

4.1. Accessing persistence properties

Your custom access rules provider may need additional configuration parameters. You can define them via the persistence properties in your persistence.xml . All you have to do, is to implement the interface net.sf.jpasecurity.persistence.PersistenceInformationReceiver . Then you will have the persistence properties injected when your persistence provider is initialized.

4.2. Implementing an Access Rules Provider

When you take a look at the methods of net.sf.jpasecurity.rules.AccessRulesProvider , you may notice that you need to create objects of type net.sf.jpasecurity.security.AccessRule . This objects may be created using a net.sf.jpasecurity.jpql.parser.JpqlParser in conjunction with an net.sf.jpasecurity.security.rules.AccessRulesCompiler , but you may subclass net.sf.jpasecurity.security.rules.AbstractAccessRulesProvider to inherit this functionality. Then you have to override initializeAccessRules and call compileRules from within.

4.3. Implementing a JDBC access rules provider

The following code shows how to implement an access rules provider that reads its access rules from a database. You may specify the needed parameters in your persistence.xml .

```
 public class JdbcAccessRulesProvider extends AbstractAccessRulesProvider {

   public static final String ACCESS_RULES_JDBC_URL_PROPERTY =
"net.sf.jpasecurity.security.rules.jdbc.url";
   public static final String ACCESS_RULES_JDBC_USERNAME_PROPERTY =
"net.sf.jpasecurity.security.rules.jdbc.username";
   public static final String ACCESS_RULES_JDBC_PASSWORD_PROPERTY =
"net.sf.jpasecurity.security.rules.jdbc.password";
   public static final String ACCESS_RULES_JDBC_TABLE_PROPERTY =
"net.sf.jpasecurity.security.rules.jdbc.table";
   public static final String ACCESS_RULES_JDBC_COLUMN_PROPERTY =
"net.sf.jpasecurity.security.rules.jdbc.column";

   protected void initializeAccessRules() {
     Map<String, String> properties = getPersistenceProperties();
     String url = getPersistenceProperty(ACCESS_RULES_JDBC_URL_PROPERTY);
     String username =
getPersistenceProperty(ACCESS_RULES_JDBC_USERNAME_PROPERTY);
     String password =
getPersistenceProperty(ACCESS_RULES_JDBC_PASSWORD_PROPERTY);
     String table = getPersistenceProperty(ACCESS_RULES_JDBC_TABLE_PROPERTY);
     String column =
getPersistenceProperty(ACCESS_RULES_JDBC_COLUMN_PROPERTY);
     Connection connection = null;
     Statement statement = null;
     ResultSet resultSet = null;
     Collection<String> accessRules = new HashSet<String>();
     try {
       connection = DriverManager.getConnection(url, username, password);
       statement = connection.createStatement();
       resultSet = statement.executeQuery("SELECT " + column + " FROM " + table);
       while (resultSet.next()) {
         accessRules.add(resultSet.getString(1));
       }
       compileRules(accessRules);
     } catch (SQLException e) {
       throw new PersistenceException("Error reading access rules", e);
     } finally {
       close(resultSet);
       close(statement);
       close(connection);
     }
   }

   private String getPersistenceProperty(String propertyName) {
     String propertyValue = getPersistenceProperties().get(propertyName);
     if (propertyValue == null) {
       throw new PersistenceException("Error reading acces rules, property " + propertyName +
" must be set");
     }
     return propertyValue;
   }

   private void close(ResultSet resultSet) {
     if (resultSet != null) {
       try {
         resultSet.close();
       } catch (SQLException e) {
         //ignore
       }
     }
   }

   private void close(Statement statement) {
     if (statement != null) {
       try {
         statement.close();
       } catch (SQLException e) {
         //ignore
       }
     }
   }

   private void close(Connection connection) {
     if (connection != null) {
       try {
         connection.close();
       } catch (SQLException e) {
         //ignore
       }
     }
   }
 }
```

In your persistence.xml you can specify the needed parameters like following.

```
 <persistence ...>

   <persistence-unit name="..." ...>

     ...

     <properties>
       <property name="net.sf.jpasecurity.security.rules.jdbc.url" value="jdbc:your.db.url" />
       <property name="net.sf.jpasecurity.security.rules.jdbc.username" value="your_username"
/>
       <property name="net.sf.jpasecurity.security.rules.jdbc.password" value="your_password"
/>
       <property name="net.sf.jpasecurity.security.rules.jdbc.table"
value="your_table_with_access_rules" />
       <property name="net.sf.jpasecurity.security.rules.jdbc.column"
value="your_column_containing_the_access_rules" />
     </properties>

   </persistence-unit>
 </persistence>
```

You can implement a custom security context in a similar way like the access rules provider. You only have to implement the interface net.sf.jpasecurity.configuration.SecurityContext and specify the property net.sf.jpasecurity.security.context in your persistence.xml with the classname of your implementation of theinterface net.sf.jpasecurity.configuration.SecurityContext . Take a look at its javadoc documentation for further reference.

4.4. Accessing persistence properties

As your access rules provider your custom security context may need additional configuration parameters, too. You also can define them via the persistence properties in your persistence.xml . Again you have to implement the interface net.sf.jpasecurity.persistence.PersistenceInformationReceiver to have the persistence properties injected when your security context is initialized.

4.5. Implementing a servlet-filter security context

The following code shows how to implement a security context that reads its authentication information from the HttpSession .

```
 public class SecurityContextFilter implements SecurityContext, Filter {

    private static final Alias PRINCIPAL_ALIAS = new Alias("principal");
    private static final Alias ROLES_ALIAS = new Alias("roles");
    private static final Alias TENANT_ALIAS = new Alias("tenant");
    private static final Collection<Alias> ALIASES
      = Collections.unmodifiableList(Arrays.asList(PRINCIPAL_ALIAS, ROLES_ALIAS,
TENANT_ALIAS));

    private static ThreadLocal<HttpSession> session = new ThreadLocal<HttpSession>();

    public Collection<Alias> getAliases() {
        return ALIASES;
    }

    public Object getAliasValue(Alias alias) {
        HttpSession session = SecurityContextFilter.session.get();
        if (session == null) {
            return null;
        }
        return session.getAttribute(alias.getName());
    }

    public Collection<?> getAliasValues(Alias alias) {
        Object aliasValue = getAliasValue(alias);
        if (aliasValue instanceof Collection) {
            return (Collection<?>)aliasValue;
        } else if (aliasValue == null) {
            return null;
        } else if (aliasValue.getClass().isArray()) {
            return Arrays.asList((Object[])aliasValue);
        } else {
            return Collections.singleton(aliasValue);
        }
    }

    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
throws IOException, ServletException {
        try {
            if (request instanceof HttpServletRequest) {
                HttpServletRequest httpRequest = (HttpServletRequest)request;
                AuthenticationFilter.session.set(httpRequest.getSession());
            }
            chain.doFilter(request, response);
        } finally {
            AuthenticationFilter.session.remove();
        }
    }

    public void init(FilterConfig config) throws ServletException {
    }

    public void destroy() {
    }
}
```

You now have to specify the class as web filter in your web.xml .

```
 <?xml version="1.0" encoding="ISO-8859-1"?>
 <web-app version="2.4"
         xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-
app_2_4.xsd">

     ...

     <filter>
         <filter-name>securityContextFilter</filter-name>
         <filter-class>your.package.SecurityContextFilter</filter-class>
     </filter>

     <filter-mapping>
       <filter-name>securityContextFilter</filter-name>
       <url-pattern>/*</url-pattern>
     </filter-mapping>

     ...

 </web-app>
```

Now your login-process may store the authentication information (the principal, roles and tenant) in the HttpSession and you are done.

5. One Step deeper

JPA Security intercepts your action with the EntityManager . Whenever you retrieve an entity from your EntityManager , it is subsidized by a proxy from JPA Security. Likewise whenever you perform a JPQL-query, it is modified with additional clauses and parameters to match your security rules.

5.1. Modification of queries

JPA Security modifies the where-clause of your JPQL queries by adding restrictions according to your access rules. This behavior enforces security-rule-evaluation within the database. Only database rows resulting in entities that the user is allowed to read will be loaded. When using Hibernate as persistence provider, the Hibernate - WITH -clause is supported by JPA Security.

5.2. Secure entities

The proxies that are created around your entities are called SecureEntity (actually they implement an interface of the same name). This is how they behave:

- When an entity is accessed for the first time a check is performed whether the current user is allowed to read the entity. On the first access of the entity all one-to-one- and many-to-one-relations to other entities are replaced by relations to SecureEntities which screen the original entities Furthermore one-to-many- and many-to-many-relations are replaced by SecureCollections , which are explained later.
- Changes to the SecureEntity are buffered and only flushed to the entity when the active transaction is committed. This occurs when flush() is called on the EntityManager or a query is performed with flush-mode AUTO (which is the default). As a matter of course whenever a SecureEntity flushes its changes a check is performed whether the current user is allowed to update the entity or not. During the flush of a SecureEntity all relations to other SecureEntities or to SecureCollections are replaced by their corresponding original.

5.3. Secure collections

Collection relationships (i.e. one-to-many- and many-to-many-relations) are handled via SecureCollections . Secure collections are filtered in memory and the backing collection will contain every entity of the original relationship. The main difference is that when you access any method of a secure collection it will behave, as if it only contained those entities you are allowed to read. In addition write-access will only be possible if write-access is allowed to the owning entity. Furthermore, for performance reasons every modification to a secure collection is queued and will not be executed until a commit operation.

5.4. Other Operations

Every entity that was loaded over a secured EntityManager can be casted to SecureEntity . This interface provides methods to programmatically check accessibility, force read- and write-check (via refresh() and flush()) and check the state of the entity.

A secure EntityManager can be casted to AccessManager , which allows programmatic security-checks, too.

5.4.1. In-Memory evaluation

On every operation that does not result into a query to the database JPA Security tries to check the configured access rules in memory. That means, for normal create-, update- and delete-operations, no database interaction is needed for the access check .

In-memory evaluation works perfectly, when no subselect is contained in the access rules. When the access rules contain subselects, some constraints are placed on the definition of the access rules. The first restriction is, that access rules that contain subselects may only contain subselects within EXISTS-clauses and not within IN-clauses. This restriction is likely to change in the future.

For example the following works:

```
 GRANT ACCESS TO TestEntity entity WHERE EXISTS (SELECT e FROM TestEntity e
WHERE e = entity AND ...)
```

Whereas the following will not work for the current release:

```
 GRANT ACCESS TO TestEntity entity WHERE entity IN (SELECT e FROM TestEntity e
WHERE ...)
```

Every subselect that contains only references to pathes to properties of the checked entity will work.

For example the following works, since acl is a direct reference to a property of the checked entity (indicated by acl = entity.acl):

```
 GRANT ACCESS TO TestEntity entity WHERE EXISTS (SELECT acl FROM
AccessControlList acl WHERE acl = entity.acl AND ...)
```

Whereas the following will not work since there is no direct path from a property of the checked entity to e (Reverse navigation would take place from e.acl to e , which is currently not supported).

```
 GRANT ACCESS TO TestEntity entity WHERE EXISTS (SELECT e FROM AclEntry e
WHERE e.acl = entity.acl)
```

The access rule could be rewritten to work with in-memory evaluation:

```
 GRANT ACCESS TO TestEntity entity WHERE EXISTS (SELECT e FROM
AccessControlList acl JOIN acl.entries e WHERE acl = entity.acl)
```

When in-memory evaluation cannot take place within the previously defined constraints, there is another chance to evaluate a query in memory: When the entities that are needed for the specified evaluation are already loaded within the specific EntityManager , evaluation will take place based on that entities. That means, if every AclEntry of the specific entity was loaded into memory by previous operations, the following access rule can be evaluated:

```
 GRANT ACCESS TO TestEntity entity WHERE EXISTS (SELECT e FROM AclEntry e
WHERE e.acl = entity.acl)
```

When an access rule cannot be evaluated the access-check will return false . This behavior will change in the future since it is not deterministic since the evaluation depends on previously loaded entities .