tapestry-security-jpa guide

Version status: 0.0.4 beta

Overview

Most security implementations support RBAC, or Role-Based Access Control. With roles, it's easy and intuitive to secure access to types, or classes, of data and operations. However, all but the most simplest systems tend to also need a secure access to specific instances of data, such as user profiles or account data. For example, each user may have an access to his or her "own" data, but not to anybody else's. Security frameworks typically require programmatic checks for instance-level access control, which is error prone and time consuming to implement. An *Entity-Relationship Based Access Control* (**ERBAC**) secures data instances based on their association with the currently executing subject. Tapestry-security-jpa is a JPA specific implementation of an ERBAC security system for instance level access control, and builds on top of tapestry-security module for Apache Tapestry 5.

Configuration

To use the feature, you need to add the following dependency to your pom.xml:

```
<dependency>
  <groupId>org.tynamo.security</groupId>
  <artifactId>tapestry-security-jpa</artifactId>
  <version>0.0.4</version>
</dependency>
```

The module doesn't require mandatory configuration. If you don't explicitly specify the realm and the type of principal to use with relationship-based security rules, the implementation will use the primary principal of the subject. You can explicitly configure the principal to use in the security check:

```
@Contribute(SymbolProvider.class)
@ApplicationDefaults
public static void provideSymbols(MappedConfiguration<String, String> configuration) {
    configuration.add(JpaSecuritySymbols.ASSOCIATED_REALM, "localdb");
    configuration.add(JpaSecuritySymbols.ASSOCIATED_PRINCIPALTYPE,
Integer.class.getName());
}
```

If you explicitly specify the principal, the subject is required to have one.

Using ERBAC in JPA context

Tapestry-security-jpa allows you to secure JPA entities with two simple annotations, @RequiresRole and @RequiresAssociation. You can either secure all persistence operations, limit the scope to a specific one (INSERT, UPDATE, DELETE, READ) or target any WRITE operations. The module implements a thin, transparent SecureEntityManager wrapper protecting any find(), persist(), merge() and remove() operations on the injected EntityManager when a Subject is present (i.e. bound to the currently executing thread). No other EntityManager operations are modified, so all create*Query() operations are unsecured. Semantically, persist() and merge() don't map directly to INSERT and UPDATE operations (both can be called either for new or existing entities, but persist() executes DELETE and subsequent INSERT when persisting an existing entity) but the implementation requires INSERT permission for persist and UPDATE for merge(). You specify the association to the currently executing Subject using the value attribute of the annotation. So, for example, to securely access an entity Thing, you would declare:

```
@Entity
@RequiresAssociation("owner")
public class Thing {

  @OneToOne
  private User owner;
}
```

The module uses either the configured principal or the primary principal (if not explicitly stated) of the current Subject as the @Id attribute of the associated entity to perform the security check. For find() operations, this means that the SecureEntityManager adds an "id equals" criteria to the find() query. If the required relationship doesn't exist, null is returned as if the entity never existed. The added benefit of @RequiresAssociation is that you can find entities secured with @RequiresAssociation by passing a null id, only relying on the association (*note that this is against the EntityManager specification, which states that IllegalArgumentException should be thrown if the id parameter is null!*). However, for write operations, a EntitySecurityException is thrown before the operation is executed if the required association doesn't exist. From security perspective, explicit errors make sense for write operations, since the user specifies the entity relationship whereas for read operations, the information is hidden if the implicitly required relationship doesn't exist; in other words errors are handled the same way as you would typically handle them in REST-based write and read operations.

@RequiresRole annotation takes precedence over @RequiresAssociation, both because the former operates on types of entities rather than instances and because role-type security checks are computationally cheaper than subject-instance checks. You can limit the scope of @RequiresRole to a specific operation the same way as with @RequiresAssociation. Note that if you want to secure WRITEs by associations, you should also use @RequireRole unless you really want to allow READ operations for all users (including anonymous) - i.e. specifying WRITE doesn't imply READ.

EntityManager doesn't provide a find(...) operation for returning all entities of a particular type. However, @RequiresAssociation works equally well for securing access to entities with @ManyToOne association the the owning entity. The module provides an AssociatedEntities service and its operation *List<?> findAll(EntityManager em, Class<?> entityClass)* for returning all associated entities.

If there's no SecurityManager bound to the thread (e.g. when executing batch operations), SecureEntityManager simply delegates back to the original SecurityManager, i.e. all entity-based access control is turned off.