tapestry-hibernate-seedentity guide

Version status: 0.1.3 release stable, in production
Don't let the version number fool you. This is code that has been used for years but was refactored into its own module when we founded Tynamo. The earlier release, 0.0.1 is missing SeedEntityUpdater and support for @NaturalId. See release notes for 0.1.0, 0.1.2, 0.1.3

No more generating and running SQL seed scripts! tapestry-hibernate-seedentity module is a simple, object-oriented way to seed your database either for development or production environment or both. This works for any Tapestry5 application using Hibernate and is independent of the rest of Tynamo modules.

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

```
<dependency>
  <groupId>org.tynamo</groupId>
  <artifactId>tapestry-hibernate-seedentity</artifactId>
  <version>0.1.3</version>
</dependency>
```

Simply contribute your entities in the right order (starting from the leaf node) and you are done! Take a look at the following example:

```
@Contribute(SeedEntity.class)
 public static void addSeedEntities(OrderedConfiguration<Object> configuration) {
  User admin = new User();
  admin.setUsername("admin");
  admin.setPassword("admin");
  Set<User.Role> adminRoles = new HashSet<User.Role>();
  adminRoles.add(User.Role.admin);
  admin.setRoles(adminRoles);
  configuration.add("admin", admin);

  if (productionMode) return;

  // TEST entities
  Game game = new Game();
  game.setName("Texas Hold'em of the Month");
  Calendar calendar = GregorianCalendar.getInstance(TimeZone.getTimeZone("UTC"));
  game.setStartDate(calendar.getTime());
  game.add(GregorianCalendar.DAY_OF_YEAR, 31);
  game.setEndDate(calendar.getTime());
  configuration.add("game1", game);

  User user = new User();
  user.setUsername("user");
  user.setPassword("user");
  Set<Game> gamesParticipated = new HashSet<Game>();
  gamesParticipated.add(game);
  user.setGames(gamesParticipated);
  configuration.add("testuser1", user);
 }
```

The module uses standard Hibernate annotations

```
@NaturalId,
@Column(unique = true) or
@Table(uniqueConstraints = { @UniqueConstraint(columnNames = {...}) })
```

to identify seed entities. An entity that is found (based on the unique properties) will not be re-seeded if an existing entity is already found in the database. In case there's no uniquely identifying property (or properties) the entity is seeded (inserted) every time the application is run. Sometimes this is wanted, but more often you want only a few seed entities even if you don't have any uniquely identifying properties. In that case, you can wrap your entity in SeedEntityIdentifier. Consider the following example: if properties *start* date and *name* only **together** identify a Game uniquely, but for your testing you want only one game to exist, you can write the line ***configuration.add("game1", game);*** as ***configuration.add("game1", new SeedEntityIdentifier(game, "name"));*** You can also declare a unique property for the whole type (the entity class) for seeding purposes. Contribute ***configuration.add("gameType", new SeedEntityIdentifier(Game.class, "name"));*** and name property will be used as an identifier for each seed entity of type Game contributed that follows.

With more complex entity relations, it's sometimes difficult or impossible to save the objects "all at once", i.e. in a single transaction. For birectional relationships (the typical owner/part relationship or many-to-many with an inverse association), this is not a problem as long as you save the relationship from the "right end", but for unidirectional two-way associations, you cannot save the entity in a single transaction. A simple example of a case where unidirectional relationships are required is when you need to keep two lists of items, say a User has a prioritized list of unfinished Work and a list of finished Work (both OneToMany), and Work is assigned to a User (ManyToOne). You have to create and save a User first, then the Work and finally update User to add this new Work to his queue of items to work on. To achieve this, you can use SeedEntityUpdater. You create a User object first - you don't have to populate more than just the unique properties for it - and contribute it and then the Work object. Then you create another User object with the same unique properties, add Work to User's list and contribute it using ***configuration.add("updatedUser", new SeedEntityUpdater(user, updatedUser));***. The current transaction is committed and the user is updated as part of the new transaction. Similarly to not seeding existing entities, the entity is only updated if it is newly created (during the same invocation of SeedEntity service). If you want to update the entity on **every** invocation of seedentity, you can use *SeedEntityUpdater(Object originalEntity, Object updatedEntity, boolean forceUpdate)*.