Unit Testing

Overview

Documents the approach to unit testing and how such tests are organized.

Coverage

There are four important layers to the architecture of the application for which we could consider unit testing

- Domain model (JPA Objects)
- Service layer (JPA Services)
- REST layer (REST services)
- Application (this is covered more in Javascript Unit Testing)

Service and REST layers lend themselves better to integration testing as whether they function properly can only be determined in the context of an execution environment with data loaded that is known (or with an extensive mocking framework).

This leaves as the main candidate for unit testing the domain model objects. As these are mostly simple bean containers annotated to interact with the various frameworks, there are limits to how much can be usefully tested. In particular, we see the following areas as fruitful for unit testing of the domain model

- Getter/Setter testing (prove that get returns the same value as set)
- Copy constructors (prove that data elements are correctly copied, especially in "keepId" scenarios)
- Equality testing (prove that the equals method works)
- Hashcode testing (prove that the hascode method works and respects the equals contract).

Unit testing would also be appropriate for the service layer helper classes and the implementations of the "project specific algorithm handler".

Beyond this, the majority of the "business" logic of the application is written into either the REST service layers and so are not good candidates for unit testing approach.

Coverage Definition

NOTE: there is no formal coverage definition for unit testing at this time.

Organization

Unit tests should be kept with the classes they are intended to test in the standard src/test/java Maven structure. While the details above are not comprehensively implemented, you will see some examples of unit testing in the code.

References/Links

- http://en.wikipedia.org/wiki/Unit_testing