Code Testing Guide

As of DSpace 7 (and above), all new code MUST come with corresponding unit or integration tests.

This guide provides an overview of how to write good unit and integration tests both for the Java (REST API) backend and the Angular (UI) frontend.

Backend (Java / REST) Code Tests

Overview of Test Framework

Technologies used

- JUnit 4 for writing tests
- Mockito for mocking objects or responses within tests
- H2 In-Memory Database for the database backend within test environment
- Spring Boot Test for testing the Server Webapp / REST API, which itself uses Hamcrest (matchers), JsonPath (for JSON parsing) and other testing tools.
- GitHub Actions - This is our continuous integration (CI) system of choice. It automatically runs all code style checks & unit/integration tests for all Pull Requests. Our configuration for GitHub Actions can be found in `[src]/.github/workflows/build.yml`
- Coveralls - GitHub Actions runs test code coverage checks which it passes to Coveralls.io for easier display/analysis. If Coveralls finds code coverage has decreased significantly (which means tests were not implemented for new code), it will place a warning on a Pull Request.

These three modules encompass most of the current test framework:

- *Parent module* (pom.xml): This POM builds our testEnvironment.zip (see "generate-test-env" profile). This test environment consists of a dspace install folder (including all configs and required subdirectories). It is unzipped and used by all other modules for testing. Modules can also override the default test environment configurations by adding their own "/test/data/dspaceFolder"
- *dspace-api* (Java API): This layer has some basic unit & integration tests for the Java API / DAOs. (Unfortunately, at this time, this module does not have as many tests as it could/should)
- *dspace-server-webapp* (Server Webapp / REST API): This layer is primarily integration tests for the REST API and other web interfaces (SWORD, OAI-PMH, etc).

Checklist for building good tests

- [ ] **Integration Tests** must be written for any methods/classes which *require* database access to test. We feel integration tests are more important than unit tests (and you'll see this in our codebase).
- [ ] **Unit Tests** must be written for any (public or private) methods/classes which *don't require* database-level access to test. For example, a utility method that parses a string should have a unit test that proves the string parsing works as expected.
- [ ] **Include tests for different user types** to prove access permissions are working. This includes testing as (1) an Anonymous user, (2) an Authenticated User (non-Admin), (3) an Administrator and/or Community/Collection Admin (as necessary).
- [ ] **Include tests for known error scenarios & error codes**. If the code throws an exception or returns an 4xx response, then a test should prove that is working.
- [ ] Bug fix PRs should **include a test that reproduces the bug** to prove it is fixed. For clarity, it may be useful to provide the test in a separate commit from the bug fix.
- [ ] Every test method *must **cleanup any test data created***. See guidelines below for "Cleaning up test data".
- [ ] Use "`context.turnOffAuthorisationSystem()`" sparingly, and *always follow-up* with a "`context.restoreAuthSystemState()`" *as soon as possible (and in the same test method)*. As turning off the authorization system can affect the behavior of tests, only use these methods when you need to create or delete test data.
- [ ] If a test needs to temporarily modify a configuration property's value (in any *.cfg file), see the guidelines below for "Changing configuration properties in tests"

Writing Integration Tests

A few quick guidelines on writing DSpace Integration Tests

- All integration test classes *must end in "IT".* For example: "ItemRestRepositoryIT.java" or "StructBuilderIT.java"
- All integration test classes should extend one of the Abstract classes provided. These include:
  - For dspace-server-webapp, two Abstract classes exist, based on the type of integration test
    - `org.dspace.app.rest.test.AbstractControllerIntegrationTest` : This style of Integration Test is for testing any class which is an `@Controller` (which is the majority of classes in the REST API / Server Webapp). This type of integration test uses Spring's built in caching and `MockMVC` to speed up the integration tests. One example is the ItemRestRepositoryIT (which tests the "/api/core/items" endpoints in the REST API).
    - `org.dspace.app.rest.test.AbstractWebClientIntegrationTest`: This style of Integration Test is for testing classes which require a full webserver to run. As such, it will run slower, and is primarily used for non-REST API classes, such as testing SWORD or OAI-PMH. Those each require a full webserver to be started, as they are not Spring Controllers (and are not built on Spring Boot). One example is OAIpmhIT (which tests the dspace-oai module)
  - For dspace-api, all Integration Tests should extend org.dspace.AbstractIntegrationTest
- As Integration Tests generate a lot of test data, all Integration tests must be sure to follow our guidelines for "Cleaning up test data" (see below).
- Many example Integration Tests can be found in the dspace-server-webapp (see org.dspace.app.rest.*) and dspace-api modules. We recommend looking at them for example code, etc.

Writing Unit Tests

A few quick guidelines on writing DSpace Unit Tests

- All unit test classes *must end in "Test".* For example: "ItemTest.java" or "DiscoverQueryBuilderTest"
- For dspace-api, all Unit Tests should extend either AbstractUnitTest or AbstractObjectTest. For example, see ItemTest
- For dspace-server-webapp, Unit Tests need not extend any class and just need to be named ending with Test. For example, see DiscoverQueryBuilderTest
- Many example Unit Tests can be found in the dspace-api, and a few in the dspace-server-webapp. We recommend looking at them for example code, etc.

Cleaning up test data

Integration Tests necessarily have to create test data to verify the code they are testing is working properly. But, just as importantly, they *must cleanup any test data they create.* **Integration tests which do not cleanup after themselves often result in random or odd CI failures.** These odd failures in CI builds may occur anytime the CI environment runs tests *in a different order* than your local machine and test data from an earlier test directly affects the results of a later test. Keep in mind, JUnit has no defined *order of execution* for tests. So, if you are seeing tests succeed on your system, but fail in CI (or another system), then it's almost certainly because tests are running in a different order on the two systems...and one order is succeeding while another is failing (likely cause of test data not being cleaned up in prior tests).

Here are three ways to ensure your test data is cleaned up properly in any Integration tests you create. They are roughly prioritized in terms of preference.

1. **Use Builders for automatic cleanup**: Whenever possible, use the Builder test classes (see `org.dspace.app.rest.builder.*`) in "dpace-server-webapp", as these Builder classes *automatically cleanup after themselves*!

   ```
   // Example of creating a test Item via the ItemBuilder class
   // As soon as the method using this "testItem" completes, the "testItem" will be automatically deleted
   // by the AbstractBuilderCleanupUtil (which is called @After every test)
   context.turnOffAuthorisationSystem();
   Item testItem = ItemBuilder.createItem(context, collection);
   context.restoreAuthSystemState();
   ```

2. **Cleanup after test POST or file upload**: If you are testing a `POST` command (or file upload), you MUST cleanup the POSTed data by parsing the ID out of the response and using a Builder class for cleanup. Our best practice is to use the following code logic. Further examples can be found in the codebase.

```
import static com.jayway.jsonpath.JsonPath.read;
import java.util.concurrent.atomic.AtomicReference;

AtomicReference<UUID> idRef = new AtomicReference<>();
try {
    // Example of a POST command to create a new Collection while logged in as the user with the given
"authToken"
    getClient(authToken).perform(post("/api/core/collections")
                        ...[various params and data sent via POST]...

                        // Check the POST was successful, which means we created test content & need to
clean it up!
                        .andExpect(status().isCreated())

                        // From the JSON response, read the "id" field, parse it as a UUID, and save to
"idRef" local variable.
                        .andDo(result -> idRef.set(UUID.fromString(read(result.getResponse().
getContentAsString(), "$.id"))));
} finally {
    // Using the CollectionBuilder, delete the Collection with UUID equal to value of idRef
    CollectionBuilder.deleteCollection(idRef.get());
}
```

3. *(If neither of the above are possible)* **"Manually" delete created data**: If you are creating test data in either the "dspace-api" or "dspace-server-webapp", you should *use the Builder method (see above).* This manual test cleanup may no longer be necessary (in v7.x and above) & should be avoided wherever possible. That said, here's an example of manual cleanup:

```
// Create test data by temporarily turning off authorization
context.turnOffAuthorisationSystem();
WorkspaceItem workspaceItem = workspaceItemService.create(context, collection, true);
Item item = installItemService.installItem(context, workspaceItem);
context.restoreAuthSystemState();

[perform various tests]

// Delete the test Item created, again by temporarily turning off authorization
context.turnOffAuthorisationSystem();
itemService.delete(context, item);
context.restoreAuthSystemState();
```

Changing configuration properties in tests

The DSpace ConfigurationService makes this very easy to do!

All you have to do is call `setProperty` in your test to change the value to whatever value your test needs or expects, for example:

```
// Change the value of the "dspace.ui.url" to be "http://mydspace.edu"
// NOTE: there are no special permissions required to change values in tests. So, all you need is this one line.
configurationService.setProperty("dspace.ui.url", "http://mydspace.edu");

// Any tests after the above call will see "dspace.ui.url = http://mydspace.edu"
// NOTE: once the test method completes, our test environment will automatically reset "dspace.ui.url" back to
the default value.
```

NOTE: You do NOT need to reset the property value back to the default setting. After every test runs, the ConfigurationService reloads the defaults from the `dspace.cfg` and the `local.cfg` used by our test environment.

Frontend (Angular UI) Code Tests

Overview of Test Framework

As the frontend is an Angular.io application (which uses Angular CLI), we follow the best practices for Testing. *This includes concentrating our effort on Unit tests (specs) over Integration / end-to-end (e2e) tests per this section of the Angular testing guide.*

Technologies used

- Jasmine for writing unit/spec tests and Karma for running those tests
- Cypress for writing integration / end-to-end (e2e) tests
- GitHub Actions - This is our continuous integration (CI) system of choice. It automatically runs all code style checks & unit/integration tests for all Pull Requests. Our configuration for GitHub Actions can be found in `[src]/.github/workflows/build.yml`
    - Our GitHub Actions configuration also starts up a Docker-based REST API backend for running end-to-end (e2e) tests against.
- Coveralls - GitHub Actions CI runs test code coverage checks which it passes to Coveralls.io for easier display/analysis. If Coveralls finds code coverage has decreased significantly (which means tests were not implemented for new code), it will place a warning on a Pull Request.

Test exists in a few key places in the codebase:

- Unit tests (or specs) can be found throughout the codebase (under `/src/app`) alongside the code they test. Their filenames always end with ".spec.ts". For example, the `login-page.component.ts` has a corresponding `login-page.component.spec.ts` test file.
- End to End (e2e) tests are all in the `/cypress/integration` folder. At this this time we have very few of these.

Learning Resources

- https://testing-angular.com/ is an online E-Book which walks through creating Angular unit tests in Jasmine & Angular end-to-end tests in Cypress.

Checklist for building good tests

- [ ] **Unit Tests (i.e. specs)** must be written for *every Angular Component or Service.* In other words, every "`*.component.ts`" file must have a corresponding "`*.component.spec.ts`" file, and every "`*.service.ts`" file must have a corresponding "`*.service.spec.ts`" file.
- [ ] **Integration (e2e) Tests** are recommended for new features but not yet required.
- [ ] **Include tests for different user types** (if behaviors differ per user type). This includes testing as (1) an Anonymous user, (2) an Authenticated User (non-Admin), (3) an Administrator and/or Community/Collection Admin (as necessary).
- [ ] **Include tests for known error scenarios**. For example, tests should validate when errors/warnings are expected to appear, and/or validate when buttons are expected to be enabled/disabled.
- [ ] Bug fix PRs should **include a test that reproduces the bug** to prove it is fixed. For clarity, it may be useful to provide the test in a separate commit from the bug fix.

Writing Unit (spec) Tests

A few quick guidelines on writing Angular tests using Jasmine:

- Specs are **required** for all Angular Components, Services and other classes.
- All specs should be placed in the same directory as the Angular Component which they test. The filename should end in ".spec.ts". For example, the `login-page.component.ts` has a corresponding `login-page.component.spec.ts` test file.
- As much as possible/reasonable, follow the Angular Testing Guide, which provides detailed example tests. Jasmine also has good documentation /tutorials on learning to write tests.
- Always mock all providers. Doing so ensures that you're only testing your own code, so a change to the service won't break your test. It also ensures that successive tests can't influence each other by sharing data through an service
- When testing asynchronous code, verify that your expect is actually executed. The following test will always succeed, because the expect is only executed after the it function has already completed:

  ```
  it("should encounter an expect", () => {
    timer(1000).subscribe(() => {
      expect(true).toBe(false);
    })
  })
  ```

  The easiest way to fix this would be to use the callback function it provides:

  ```
  it("should encounter an expect", (done) => {
    timer(1000).subscribe(() => {
      expect(true).toBe(false);
      done();
    })
  })
  ```

  Now the test won't complete until the callback function done() is called, so the test above will fail. Other ways of testing asynchronous code include marbles, and fakeAsync.

  A quick way to verify that your expect is actually being used is to flip it and see if that causes the test to fail.

  Note that this applies to all functions in a test suite, not just it. Here's a commit that fixes a number of these issues in beforeEach and a few in it: 066e6cd.
- If you are working on debugging specific tests, you can add a "focus" on those tests (e.g. `fdescribe` instead of `describe`). However, be warned that you **must remove that focus** before the PR can be merged, as otherwise you'll see a large decrease in code coverage (i.e. all non-focused tests will be ignored)
- https://angular.io/guide/test-debugging is a guide for debugging Angular unit tests in Karma / Jasmine.

- https://testing-angular.com/ is an online E-Book which walks through creating Angular unit tests in Jasmine & Angular end-to-end tests in Cypress.

Writing Integration (e2e) Tests

A few quick guidelines on writing Angular end-to-end (e2e) tests using Cypress:

- All e2e tests should be in the `/cypress/integration/` directory, per the Angular CLI best practices.
- Tests should be named similar to the component or page they test.
    - For example "homepage.spec.ts" obviously tests the homepage, while "header.spec.ts" may test specific aspects of the header of the entire site.
- docs.cypress.io has great guides & documentation helping you learn more about writing/debugging e2e tests in Cypress.
- https://testing-angular.com/ is an online E-Book which walks through creating Angular unit tests in Jasmine & Angular end-to-end tests in Cypress.
- NOTE: be aware that when e2e tests run, they require using a REST API backend & test data. Therefore in GitHub Actions CI, our tests run against a Docker based REST API (preloaded with basic test data) defined in the docker-compose-ci.yml