Tripal Test Suite Documentation

Release 1.4.0

Abdullah Almsaeed, Bradford Condon

Aug 24, 2018

Contents:

TripalTestSuite is a composer package that handles common test practices such as bootstrapping Drupal before running the tests, creating test file, and creating and managing database seeders (files that seed the database with data for use in testing).

CHAPTER 1

Support

Please visit our issues queue on Github for any questions, issues or contribution.

CHAPTER 2

License

TripalTestSuite is licensed under GPLv3.

2.1 Installation

Within your Drupal module path (e,g `sites/all/modules/my_module`), run the following.

```
composer require statonlab/tripal-test-suite --dev
```

2.1.1 Automatic Set Up

This module will automatically configure your tests directory, PHPUnit bootstrap files, and travis continuous integration file as well as provide an example test and an example database seeder to get you started.

From your module's directory, execute:

```
# You may specify the module name or leave it blank.
# When left blank, the name of the current directory will be used as the module name.
./vendor/bin/tripaltest init [MODULE_NAME]
```

This will - Set up the testing framework by creating the tests directory, phpunit.xml and tests/bootstrap.php - Create an example test in tests/ExampleTest.php - Create a DatabaseSeeders folder and an example seeder in tests/DatabaseSeeders/UsersTableSeeder.php - Create DevSeedSeeder.php in DatabaseSeers. See the [DevSeed section] to learn more about automatically populating the database with biological data. - Create an example `.env` file. - Create `.travis.yml` configured to use a tripal3 docker container to run your tests

You can now write tests in your `tests` folder. To enable continuous integration testing, push your module to github and enable Travis CI.

2.1.2 Forcing initialization

To force replacing files that tripaltest have perviously generated, you can use the `--force` flag. You will need to confirm this flag by typing `y` and hitting `enter`.

```
./vendor/bin/tripaltest init --force
```

2.2 Creating Tests

Using `tripaltest`, you can create test files pre-populated with all the requirements. To create a new test, run the following command from your module's root directory:

```
# Creates a test file called ExampleTest.php in the tests folder
./vendor/bin/tripaltest make:test ExampleTest

# Creates a test file called ExampleTest.php in tests/Features/Entities
# This will automatically detect and configure the namespace of your script
./vendor/bin/tripaltest make:test Features/Entities/ExampleTest
```

Warning: You should not include `tests/` in your path, nor should you specify a file extension.

Warning: Test names should end with `Test` for phpunit to recognize them.

2.3 Running Tests

Tripal Test Suite auto installs PHPunit as part of it's dependencies in composer.json. Therefore, running tests in Tripal Test Suite is done via phpunit as such:

```
./vendor/bin/phpunit
```

The command above, will read your `phpunit.xml` and runs the tests accordingly.

2.4 TripalTestCase

Test classes should extend the TripalTestCase class. Once extended, bootstrapping Drupal and reading your `.env` file is done automatically when the first test is run.

```
namespace Tests;

use StatonLab\TripalTestSuite\TripalTestCase;

class MyTest extends TripalTestCase {
}
```

Attention: If you define a `setUp` method within a test class, be sure to call `parent::setUp`!

2.5 Database Seeders

Database seeders are also supported in TripalTestSuite. They give you the ability to create reusable seeders that can be run using the `tripaltest` command line tool.

2.5.1 Creating Database Seeders

DB seeders can also be created automatically using `tripaltest`:

```
./vendor/bin/tripaltest make:seeder ExampleTableSeeder
```

The above command will create `ExampleTableSeeder.php` in `tests/DatabaseSeeders/` pre-populated with the necessary namespace, methods and properties.

2.5.2 Using Database Seeders

DB seeders support two important methods, `up()` and `down()`. The `up()` method is used to insert data into the database while the `down()` method is used to clean up the inserted data. The following is an example of a Seeder class.

```php
<?php

namespace Tests\DatabaseSeeders;

use StatonLab\TripalTestSuite\Database\Seeder;

class UsersTableSeeder extends Seeder
{
        /**
         * Seeds the database with users.
         */
        public function up()
        {
                $new_user = [
                        'name' => 'test user',
                        'pass' => 'secret',
                        'mail' => 'test@example.com',
                        'status' => 1,
                        'init' => 'Email',
                        'roles' => [
                                DRUPAL_AUTHENTICATED_RID => 'authenticated user',
                        ],
                ];

                // The first parameter is sent blank so a new user is created.
                user_save(new \stdClass(), $new_user);
        }
}
```

2.5.3 Running Seeders

You can also run the seeder manually by using the static `seed()` method. For example, within a test class, you can run `$seeder = UsersTableSeeder::seed()` which runs the `up()` method and returns an initialized seeder object. If you are using the `DBTransaction` trait, the data will be automatically rolled at the end of each test function.

The other option is to run it using `tripaltest` as follows

```
# run all available seeders
tripaltest db:seed

# Run a specific seeder by providing the class name
tripaltest db:seed ExampleSeeder
```

Attention: Running the seeder manually in a test function with `DBTransaction` enabled, means that the data is available only to that function and nothing else. However, running it using `tripaltest` makes it always available unless explicitly deleted.

2.5.4 Retrieving Seeder Data

If your seeder returns any data, you can obtain the returned record by manually running the seeder in your test. See below for an example:

```
<?php
// Seeder Class
class MySeeder extends Seeder {
        public function up() {
                // Generate some data.
                $data = db_query(...);

                return $data;
        }
}

// Test Class
class MyTest extends TripalTestCase {
        public function testExample() {
                $seeder = new MySeeder();
                $data = $seeder->up();

                // Run some tests using the generated data
                // ...
        }
}
```

2.5.5 Using DevSeed for Quick Biological Data Seeding

Tripal Test Suite ships with a default seeder called `DevSeedSeeder`. This seeder provides a quick and automated way of seeding your database with biological data such as organisms, mRNAs, BLAST annotations and InterProScan annotations. The data in the default seeder is obtained from Tripal DevSeed, which is a developer mini-set of biological data.

DevSeed uses factories and is therefore **only appropriate for testing and development** and should not be run on a production site.

> **Attention:** DevSeedSeeder.php becomes available after running `tripaltest init`. The `init` command will not override existing files unless you specify the `--force` flag so it it's safe to run it to get only the DevSeeder.

By default, the DevSeed comes with all sub-loaders disabled. To run the DevSeed seeder, you first have to configure it by uncommenting the type of data you want seeded. Then, you can run the seeder using `tripaltest db:seed DevSeedSeeder`.

1. Open `DatabaseSeeders/DevSeedSeeder.php`
2. You'll notice a few commented properties in the top of the file.
3. Uncomment and modify the properties to your need.
4. Carefully follow the instructions in this section. All loaders require an organism as well, but some are dependent on previous loaders.
5. Next, run `tripaltest db:seed DevSeedSeeder`
6. If the seeder runs successfully, you'll be able to see all the records in your Chado database.

The records provided by DevSeed are not published to your site as entities. You can do that by adding `$this->publish('CHADO_TABLE')` at the end of the `up()` method of the `DevSeedSeeder`. Replace `CHADO_TABLE` with the name of the table such as `feature` for mRNAs and `analysis` for analyses. Or, if you prefer, you can use the Tripal admin interface to publish the records.

2.6 Factories

DB factories provide a method to populate the database with fake data. Using factories, you won't have to run SQL queries to populate the Database in every test. Since they are reusable, you can define one factory for each table and use them across all tests. Usage example:

```
# Generates 100 controlled vocabularies.
# @return an array of vocabularies
$controlledVocabs = factory('chado.cv', 100)->create()
```

Factories should **only be used for testing and development purposes**.

2.6.1 Defining Factories

Factories live in `tests/DataFactory.php`. If you don't have that file, create it. Note that this file is auto created with `tripaltest init`.

Example DataFactory file:

```
<?php

use StatonLab\TripalTestSuite\Database\Factory;

Factory::define('chado.cv', function (Faker\Generator $faker) {
        return [
                'name' => $faker->name,
```

(continues on next page)

(continued from previous page)

```
                'definition' => $faker->text,
        ];
});
```

As shown in the example above, using `Factory::define()`, we can define new factories. The define method takes the following parameters:

Parameter	Type	Description	Example
$table	`string`	The table name preceded with the schema name if the schema is not public	`chado.cv` or `node`
$callback	`callable`	The function that generates the array. A `Faker\Generator` instance is automatically passed to the callable	see above for example
$primary_key	`string`	**OPTIONAL** The primary key for the given table. Primary keys auto discovered for CHADO tables only. If the factory wasn't able to find the primary key, an `Exception` will be thrown	`nid` or `cv_id`

2.6.2 Using Factories

Once defined, factories can be used in test files directly or in database seeders. Usage:

```
# Create a single CV record
$cv = factory('chado.cv')->create();
echo "$cv->name\n";

# Create 100 CV records
$cvs = factory('chado.cv', 100)->create();

foreach ($cvs as $cv) {
  echo "$cv->name\n";
}
```

2.6.3 Overriding Defaults

Sometimes you need to override a column to be a static predictable value. The `create()` method accepts an array of values to override the faker data with. Example:

```
# Let's make sure the cvterm has a specific cv id
$cv = factory('chado.cv')->create();
$cv_term = factory('chado.cvterm', 100)->create([
        'cv_id' => $cv->cv_id,
])
```

The above example creates 100 cv terms that have the same cv_id.

Factories should **only be used for testing and development purposes**. This is because they are **recursive** and create the records linked via foreign key. They do this **even if you override the default** for the linked record.

2.7 Using DB Transactions to Automatically Rollback Database Changes

Using DB transactions cleans up the database after every test by rolling back the database to the original state before the test started. Therefore, anything added to the database in one test function will not be available for the next function. If you'd like data to be available for all of the tests, see database seeders above.

To activate DB Transactions, simply add the DBTransaction trait to your test class:

```
namespace Tests;

use StatonLab\TripalTestSuite\TripalTestCase;
use StatonLab\TripalTestSuite\DBTransaction;

class MyTest extends TripalTestCase {
        use DBTransaction;
}
```

The trait will automatically activate DB transactions and rollback the database when the test is finished.

Warning: If the code you are testing requires a transaction, Postgres will fail since it does not support nested transactions.

2.8 Publishing Tripal Entities

We provide an easy way to convert your chado records into entities. This is the equivalent of publishing Tripal content using the GUI.

Publishing records is possible in both database seeders and directly in the test class.

The following publishes all features in `chado.feature` if they have not been published yet.

```
// Get the cvterm id of mRNA
$cvterm = chado_select_record('cvterm', ['cvterm_id'], ['name' => 'mRNA'])[0];

// Create 100 mRNA records
$features = factory('feature', 100)->create(['type_id' => $cvterm->cvterm_id]);

// Publish all features in chado.feature
$this->publish('feature');
```

The following publishes only the given feature ids:

```
// Get the cvterm id of mRNA
$cvterm = chado_select_record('cvterm', ['cvterm_id'], ['name' => 'mRNA'])[0];

// Create 100 mRNA records
$features = factory('feature', 100)->create(['type_id' => $cvterm->cvterm_id]);

// Get the ids of our new features
$feature_ids = [];
foreach ($features as $feature) {
        $feature_ids[] = $feature->feature_id;
```

(continues on next page)

(continued from previous page)

```
}

// Publish only the given features
$this->publish('feature', $feature_ids);
```

The previous examples create mRNA entities.

Attention: An mRNA bundle must already be available before running this script.

2.9 Testing HTTP Requests

TripalTestSuite provides a comprehensive HTTP testing methods. It allows you to call site urls and check that your Drupal menu items are working as expected.

For example, the following tests that the homepage is accessible and that the name of the website is present in the response.

```
public function testHomePage() {
  // Send a GET request
  $response = $this->get('/')

  // Verify the HTTP response code is "200 OK" and that the site name is visible
  $response->assertStatus(200)
                   ->assertSee('My Site');
}
```

2.9.1 Available HTTP Testing Methods

The following table describes all available HTTP methods in any test class that extends TripalTestSuite:

name	parameters	Description	Return
`$this->get()`	**$url** `string` The url to calln**$params** `array` Query parameters.n**$headers** `array` Additional HTTP headers	Sends a GET request	`TestResponse`
`$this->post()`	**$url** `string` The url to calln**$params** `array` Form request parameters.n**$headers** `array` Additional HTTP headers	Sends a POST request	`TestResponse`
`$this->put()`	**$url** `string` The url to calln**$params** `array` Query parameters.n**$headers** `array` Additional HTTP headers	Sends a PUT request	`TestResponse`
`$this->patch()`	**$url** `string` The url to calln**$params** `array` Query parameters.n**$headers** `array` Additional HTTP headers	Sends a PATCH request	`TestResponse`
`$this->delete()`	**$url** `string` The url to calln**$params** `array` Query parameters.n**$headers** `array` Additional HTTP headers	Sends a DELETE request	`TestResponse`

The `TestResponse` returned from the HTTP requests, provide the following set of assertion methods:

name	Parameters	Description			
`$response->assertStatus()`	**$code** `int`	Verify the returned HTTP status code is equal to `$code`			
`$response->assertSeeContent()`	**$content** `string`	Verify the given string is present in the returned response body (i	e HTML	JSON	etc)
`$response->assertJsonStructure()`	**$structure** `array`	Verifies that the returned JSON matches the given structure (see below for example)			
`$response->assertSuccessful()`	none	Verify the returned HTTP status code is between 200 and 299	which are HTTP's successful response codes		

2.10 User Authentication

Authenticating a user with TripalTestSuite is very simple using the `actingAs` method. When authenticating a user with TripalTestSuite, the user is automatically signed out by the end of each test method, which guarantees that your other tests are using the anonymous user unless you specifically tell it otherwise.

```
public function testExample() {
  // Authenticate the superuser who has an id 1
  $this->actingAs(1);

  // Verify that the user is the admin user
  global $user;
  $this->assertTrue(1 === $user->uid);
}
```

Attention: The `actingAs` method can take a user id to authenticate or a Drupal user object.

2.11 Helper Methods

TripalTestSuite provides a set of helper methods to automate tedious aspects of testing.

2.11.1 Silently Testing Printed Output

Since tests should run "silently", i.e. without printing output to the screen, we'd have to create an output buffer to collect printed strings into a variable. In PHP, this can be done as such:

```
// Supress tripal errors
putenv("TRIPAL_SUPPRESS_ERRORS=TRUE");
ob_start();

// Run the call
echo "testing";
$output = ob_get_contents();
```

(continues on next page)

(continued from previous page)

```
// Clean the buffer and unset tripal errors suppression
ob_end_clean();
putenv("TRIPAL_SUPPRESS_ERRORS");
```

However, TripalTestSuite provides a `silent()` method that automates this process, provides helpful assertions and supports larger strings. Example usage:

```
$output = silent(function() {
  echo "testing";
});
$output->assertSee('testing'); // true!
```

Warning: This method has a maximum string size to avoid memory leaks. The size is set in PHP's ini file as `output_buffering`, which by default is set to 4KB. If you would like to collect larger strings, you must adjust your PHP settings.

2.11.2 Assertions and Methods

The silent method returns a SilentResponse which provides the following methods.

Method	Arguments	Description
`assertSee()`	`$value` mixed	Asserts that the given value is present in the suppressed printed output
`assertReturnEquals()`	`$value` mixed	Asserts that the given value equals the returned value from the called function
`assertJsonStructure()`	`$strcture` array ''$data'' array **Optional**	Asserts that the given stricture matches that of the suppressed printed output
`getContent()`	None	Get the suppressed printed content as a string
`getReturnValue()`	None	Get the returned value from the called function

Examples

```
$output = silent(function() {
        drupal_json_output(['key' => 'value']);
        return true;
});

$output->assertSee('value')
           ->assertJsonStructure(['key'])
           ->assertReturnEquals(true);
```

You can also call methods directly in the Callable function:

```
// Assume we have the following function
function tripal_print_message($message) {
  echo $message;
}

$output = silent(function() {
  tripal_print_message('tripal test suite');
});
$output->assertSee('test');

// Get the output as a string
$rawOtput = $output->getContent();
```

2.11.3 Access Private and Protected Properties and Methods of Objects

TripalTestSuite provides a `reflect()` method that accepts an object and makes all of the properties and methods public and available for testing. Assume we have the following class:

```
class PrivateClass
{
        private $private;

        public function __construct($private = 'private')
        {
                $this->private = $private;
        }

        protected function myProtected()
        {
                return 'protected';
        }

        private function privateWithArgs($one, $two)
        {
                return $one.' '.$two;
        }
}
```

Because of the functions and properties of the class are private or protected, we normally would not be able to access any of them. However, we can force access using the reflect helper. See below for an examples.

2.11.4 Accessing Private and Protected Methods

```
// Pass an initialized class to the reflect method
$myObject = new PrivateClass();
$privateClass = reflect($myObject);

// Accessing protected methods
$value = $privateClass->myProtected();
$this->assertEquals('protected', $value);

// Accessing private methods with arguments
$value = $privateClass->privateWithArgs('one', 'two');
$this->assertEquals('one two', $value);
```

2.11.5 Accessing Properties

```
// Pass an initialized class to the reflect method
$myObject = new PrivateClass();
$privateClass = reflect($myObject);

$this->assertEquals('private', $privateClass->private);
```

2.12 Environment Variables

You can specify the Drupal web root path in `tests/.env`.

```
# tests/.env
BASE_URL=http://localhost
DRUPAL_ROOT=/var/www/html
FAKER_LOCALE=en_US
```

This allows TripalTestSuite to bootstrap the entire Drupal framework and make it available in your tests.

2.13 Upgrading TripalTestSuite

Since we are using composer to manage releases, running `composer update` should update all your dependencies to the latest version. However, you need to be aware of how composer deals with versioning.

Upgrading to a major versions (e.g, from 1.5.0 to 2.0.0), will require that you change the specified version in your composer.json file. Upgrading minor version (e.g, 1.0.0 to 1.1.0) can be made automatic by specifying `1.*` as your `tripal-test-suite` version.