Is it bad practice to run tests on a database instead of on fake repositories?

Asked 15 years, 9 months ago Modified 7 years, 11 months ago

Viewed 4k times

14

I know what the advantages are and I use fake data when I am working with more complex systems.

What if I am developing something simple and I can easily set up my environment in a real database and the data being accessed is so small that the access time is not a factor, and I am only running a few tests.

Is it still important to create fake data or can I forget the extra coding and skip right to the real thing?

When I said real database I do not mean a production database, I mean a test database, but using a real live DBMS and the same schema as the real database.

tdd domain-driven-design data-access

Share

Improve this question

Follow

edited Dec 31, 2016 at 23:11

Peter Mortensen

31.6k ● 22 ● 109 ● 133

asked Mar 3, 2009 at 18:49

Sruly

10.5k ● 6 ● 35 ● 41

"a real DB" you mean a production DB - or a DB using the same DBMS ? – philant Mar 4, 2009 at 6:45

1 I mean the same DBMS and schema, but on a test DB – Sruly Mar 4, 2009 at 10:52

Thnx, then I confirm my answer. – philant Mar 5, 2009 at 6:52

11 Answers

Sorted by: Highest score (default)

24

The reasons to use fake data instead of a real DB are:

1. Speed. If your tests are slow you aren't going to run them. Mocking the DB can make your tests run much faster than they otherwise might.
2. Control. Your tests need to be the sole source of your test data. When you use fake data, your tests choose which fakes you will be using. So there is no chance that your tests are spoiled because someone left the DB in an unfamiliar state.
3. Order Independence. We want our tests to be runnable in any order at all. The input of one test should not depend on the output of another. When your tests control the test data, the tests can be independent of each other.

4. Environment Independence. Your tests should be runnable in any environment. You should be able to run them while on the train, or in a plane, or at home, or at work. They should not depend on external services. When you use fake data, you don't need an external DB.

Now, if you are building a small little application, and by using a real DB (like MySQL) you can achieve the above goals, then by all means use the DB. I do. But make no mistake, as your application grows you will eventually be faced with the need to mock out the DB. That's OK, do it when you need to. YAGNI. Just make sure you DO do it WHEN you need to. If you let it go, you'll pay.

Share Improve this answer

Follow

edited Dec 31, 2016 at 23:12

Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Mar 5, 2009 at 20:46

Uncle Bob

4,750 ● 30 ● 19

2 Speed is the only issue that I can see there being with using a real DB. Control, Order Independence, and Environment Independence are all solvable with real databases. E.g., ensure you drop the database before your run the first test, run all tests under transactions, don't commit the transactions. It's much more work to create a fake implementation of the database layer which behaves enough like the real one to write useful tests than to just point your DAL at a test-specific database. – binki Jun 29, 2018 at 18:29

2

It sort of depends what you want to test. Often you want to test the actual logic in your code not the data in the database, so setting up a complete database just to run your tests is a waste of time.

Also consider the amount of work that goes into maintaining your tests and testdatabase. Testing your code with a database often means your are testing your application as a whole instead of the different parts in isolation. This often result in a lot of work keeping both the database and tests in sync.

And the last problem is that the test should run in isolation so each test should either run on its own version of the database or leave it in exactly the same state as it was before the test ran. This includes the state after a failed test.

Having said that, if you really want to test on your database you can. There are tools that help setting up and tearing down a database, like dbunit.

I've seen people trying to create unit test like this, but almost always it turns out to be much more work then it is actually worth. Most abandoned it halfway during the project, most abandoning ttd completely during the project, thinking the experience transfer to unit testing in general.

So I would recommend keeping tests simple and isolated and encapsulate your code good enough it becomes possible to test your code in isolation.

Share Improve this answer

Follow

answered Mar 3, 2009 at 19:09

DefLog

1,009 ● 7 ● 13

2

As far as the Real DB does not get in your way, and you can go faster that way, I would be pragmatic and go for it.

In unit-test, the "test" is more important than the "unit".

Share Improve this answer

Follow

edited Mar 3, 2009 at 19:33

answered Mar 3, 2009 at 19:25

philant

35.7k ● 11 ● 73 ● 113

I'm afraid I must disagree. A test that not tests a unit will be harder to maintain and this can lead to loss of will to write tests. I've also arguged that unit testing will lead to a cleaner design, because there need to be units to be tested.
– DefLog Mar 3, 2009 at 20:11

I meant it's better to have a test that covers several units than no test at all. If he unit-tests with the real DBMS (not the production one though), what would it change the overall design compared to unit testing against a fake DB ? – philant Mar 4, 2009 at 6:51

I think it depends on whether your queries are fixed inside the repository (the better option, IMO), or whether the repository exposes composable queries; for example - if you have a repository method:

```
IQueryable<Customer> GetCustomers() {...}
```

Then your UI could request:

```
var foo =
GetCustomers().Where(x=>SomeUnmappedFunction(x));

bool SomeUnmappedFunction(Customer customer) {
   return customer.RegionId == 12345 &&
customer.Name.StartsWith("foo");
}
```

This will pass for an object-based fake repo, but will fail for *actual* db implementations. Of course, you can nullify this by having the repository handle all queries internally (no external composition); for example:

```
Customer[] GetCustomers(int? regionId, string
nameStartsWith, ...) {...}
```

Because this can't be composed, you can check the DB and the UI independently. With composable queries, you are forced to use integration tests throughout if you want it to be useful.

 answered Mar 3, 2009 at 19:38

Follow

Marc Gravell
1.1m ● 273 ● 2.6k ● 3k

1

It rather depends on whether the DB is automatically set up by the test, also whether the database is isolated from other developers.

At the moment it may not be a problem (e.g. only one developer). However (for manual database setup) setting up the database is an extra impediment for running tests, and this is a very bad thing.

Share Improve this answer

Follow

answered Mar 3, 2009 at 18:57

Douglas Leeder
53.3k ● 9 ● 99 ● 138

1

If you're just writing a simple one-off application that you absolutely know will not grow, I think a lot of "best practices" just go right out the window.

You don't *need* to use DI/IOC or have unit tests or mock out your db access if all you're writing is a simple "Contact Us" form. However, where to draw the line between a "simple" app and a "complex" one is difficult.

In other words, use your best judgment as there is no hard-and-set answer to this.

Share Improve this answer

Follow

answered Mar 3, 2009 at 18:59

Kevin Pang
41.4k ● 38 ● 122 ● 173

1

It is ok to do that for the scenario, as long as you don't see them as "unit" tests. Those would be integration tests. You also want to consider if you will be manually testing through the UI again and again, as you might just automated your smoke tests instead. Given that, you might even consider not doing the integration tests at all, and just work at the functional/ui tests level (as they will already be covering the integration).

As others as pointed out, it is hard to draw the line on complex/non complex, and you would usually now when it is too late :(. If you are already used to doing them, I am sure you won't get much overhead. If that were not the case, you could learn from it :)

Share Improve this answer

Follow

answered Mar 3, 2009 at 19:19

eglasius

36k ● 5 ● 68 ● 110

1

Assuming that you want to automate this, the most important thing is that you can programmatically generate your initial condition. It sounds like that's the case, and even better you're testing real world data.

However, there are a few drawbacks:

Your real database might not cover certain conditions in your code. If you have fake data, you cause that behavior to happen.

And as you point out, you have a simple application; when it becomes less simple, you'll want to have tests that you can categorize as unit tests and system tests. The unit tests should target a simple piece of functionality, which will be much easier to do with fake data.

Share Improve this answer

Follow

edited Dec 31, 2016 at 23:13

Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Mar 3, 2009 at 18:59

Jared Oberhaus

14.6k ● 5 ● 58 ● 55

0

One advantage of fake repositories is that your regression / unit testing is consistent since you can expect the same results for the same queries. This makes it easier to build certain unit tests.

There are several disadvantages if your code (if not read-query only) modifies data: - If you have an error in your code (which is probably why you're testing), you could end up breaking the production database. Even if you didn't break it. - if the production database changes over time and especially while your code is executing, you may lose track of the test materials that you added and have a hard time later cleaning it out of the database. - Production queries from other systems accessing the database may treat your test data as real data and this can corrupt results of important business processes somewhere down the road. For example, even if you

marked your data with a certain flag or prefix, can you assure that anyone accessing the database will adhere to this schema?

Also, some databases are regulated by privacy laws, so depending on your contract and who owns the main DB, you may or may not be legally allowed to access real data.

If you need to run on a production database, I would recommend running on a copy which you can easily create during of-peak hours.

Share Improve this answer

Follow

answered Mar 3, 2009 at 18:57

Uri

89.6k ● 51 ● 226 ● 322

Of course I will not test on a production DB. I want to test on a test DB isnted of on mock data. – Sruly Mar 3, 2009 at 19:52

That was really not clear from your question, since you talked about the "real DB" which sounded like the production DB. – Uri Mar 3, 2009 at 20:52

0

It's a really simple application, and you can't see it growing, I see no problem running your tests on a real DB. If, however, you think this application will grow, it's important that you account for that in your tests.

Keep everything as simple as you can, and if you require more flexible testing later on, make it so. Plan ahead

though, because you don't want to have a huge application in 3 years that relies on old and hacky (for a large application) tests.

Share Improve this answer

Follow

edited Mar 3, 2009 at 18:59

GEOCHET

21.3k ● 15 ● 77 ● 99

answered Mar 3, 2009 at 18:55

Alex Fort

18.8k ● 5 ● 44 ● 51

0

The downsides to running tests against your database is lack of speed and the complexity for setting up your database state before running tests.

If you have control over this there is no problem in running the tests directly against the database; it's actually a good approach because it simulates your final product better than running against fake data. The key is to have a pragmatic approach and see best practice as guidelines and not rules.

Share Improve this answer

Follow

edited Dec 31, 2016 at 23:14

Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Mar 3, 2009 at 19:00

terjetyl

9,565 ● 5 ● 56 ● 73