# A guide to safely and efficiently test code that uses repositories

https://blog.malt.engineering/a-guide-to-safely-and-efficiently-test-code-that-uses-repositories-c91effb41dd0

With this article, I’d like to share a set of techniques that we use together at Malt to write modules making use of repositories, and to unit-test them. These techniques give us more business oriented logic and tests, while allowing for easy refactoring and quick tests.

Given that every part of what follows is already well documented and shared in various blog posts and conferences, why write yet another article about it, you might ask?  
Well, although we’ve used those techniques for long, we don’t use them as broadly as we would like. Then, some newcomers to Malt — as well as external developers — discovered them during our discussions and have asked for a simple demonstration of their use together as one comprehensive technique. So this article will attempt to be that demonstration, and should serve as an internal “how to”.

**Some definitions first**

As sharing a common language is key to good understanding, let’s first ensure we agree on a vocabulary.

**The many flavors of test doubles**

This article will mainly refer to two specific kinds of test doubles: Fake Objects (or just “Fakes” for short), and Mock Objects (“Mocks”). Three other kinds of test doubles are usually distinguished: Stubs, Spies, and Dummies. We developers often call all test doubles “mocks” indifferently, either because we’re lazy (I am), or because we just don’t know about all those flavors of test doubles. If that is your case, please do yourself a favor and take 2 minutes to read one of the following reference articles: [xUnit Patterns’ Test Double](http://xunitpatterns.com/Test%20Double.html" \t "_blank) or [Martin Fowler’s Test Double,](https://www.martinfowler.com/bliki/TestDouble.html) or [Uncle Bob’s Little Mocker](https://blog.cleancoder.com/uncle-bob/2014/05/14/TheLittleMocker.html). I’ll wait.

**Repository**

The second concept we’ll refer to is that of a Repository. Again, it’s likely that all readers are familiar with repositories, so let’s give a very simple definition of it and move on: a repository encapsulates the logic needed to fetch or save data into a store. The most common implementations will hide code accessing a database and using SQL queries, but it need not be. Again, you may want to read about repositories as a [Pattern of Enterprise Application Architecture](https://www.martinfowler.com/eaaCatalog/repository.html) or as a [Domain Driven Design’s building block](https://www.infoq.com/minibooks/domain-driven-design-quickly/).

As we’ll see, this article demonstrates a repository speaking the language of the domain, defined as a port of an Hexagonal Architecture, with several adapters. To be clear, that code organization isn’t a prerequisite to the techniques discussed next.

**The context: problems and solutions**

**Requirements for good unit tests**

Let’s first list what qualities we want unit tests to have.

1. Obviously, they must be useful. We want tests to cover the features of our system and prevent regressions.
2. Unit tests must run fast. A fast feedback is invaluable whereas long tests just get in the way of the development experience.

We don’t want tests to become the new compiling phase, as in [this classic XKCD comic](https://xkcd.com/303/).

3. Unit tests must be independent, as much as possible. One test’s setup must not impact another test.

4. Tests must be readable, because they document our system, and because we have to maintain them.

5. And last but clearly not the least: tests should help us build our system (especially when doing TDD) and mustn’t get in the way of refactoring.

**The problem with repositories and other infrastructure dependencies**

Infrastructure dependencies — and therefore databases — may get in the way of the requirements listed above:

* they are usually complex to set up, especially on developer workstations,
* starting the required middleware takes time, especially if done for each test,
* sharing them for several tests may cause conflicts.

In recent years, the rise of containers alleviated some of those points, but tests still still take time to run when using them. 15 seconds for a test is too long if all tests take that time. At Malt, we use [Testcontainers](https://www.testcontainers.org/" \t "_blank) with a trick to reuse containers instead of starting new ones every time, but it still takes up to 5 seconds per test.

Therefore, a good tradeoff (that we apply at Malt) might be to:

* use containers to test interactions with our infrastructure dependencies, but only once, i.e. to test our repositories,
* use test doubles in place of the “production” repositories when writing other tests.

And more often that not, using test doubles means using a mocking library.

**The problem when using mocking libraries to write test doubles**

Mocking libraries offer very clever and convenient APIs to write test doubles. For instance [Mockito](https://site.mockito.org/" \t "_blank) (for Java) and [Mockk](https://mockk.io/" \t "_blank) (for Kotlin) are very nice tools that come handy in many situations. They are invaluable to test interactions with external code.

But as is often the case, overusing them may do more harm than good.

As the saying goes: “when all you have is a hammer…” ([credits](https://unsplash.com/photos/yYRAvcPrWms))

Overusing mocking libraries leads to several drawbacks that I’ll list below.

**1. Less readable tests.**

Compare (typical Java/Mockito code):

when(userRepository.find(userId)).thenReturn(Optional.of(testUser));

or (typical Kotlin/Mockk code):

every { userRepository.find(userId) } returns testUser

to:

userRepository.register(testUser)

Compared to the former examples, the intention of the latter is 100% clear and looks the same as “production” code, without any boiler plate code.

Of course, one may improve the situation by extracting any of the first samples in a well-named utility method, so that the calling code looks like the following:

givenAnExistingUser(testUser)

Still, that kind of code tends to be repeated in every test, where using the actual repository contract would lead to the same simple code everywhere.

Again, one could improve the situation by extracting the mocking logic in shared test fixtures. But not all code interact with the repository in the same way: one call site may use userRepository.findById(userId) while another one will use userRepository.findByEmail(emailAddress), which makes it more challenging to write shared test fixtures with mocking libraries. This also brings to the next point.

**2. Interaction locking and refactoring**

Because tests written using mocking libraries focus on individual interactions with the stubbed/mocked/… objects rather that on general expectations for the overall feature (it is the *raison d’être* of those libraries), they tend to lock these interactions and therefore get in the way of refactoring our code.

In our previous example, should the business logic change to use findByEmail instead of findById, then tests using mocking libraries would break because that interaction wasn’t the one specified. If testing that this specific interaction must occur wasn’t the*intent*of the test but all we wanted was for the mock object to satisfy a precondition of the test, then it’s too bad. Tests using a “real” repository wouldn’t care about such a change, as long as the appropriate user has been registered beforehand.

And the following point shows a variation of what’s essentially the same problem.

**3. Artificial restriction on what is the “unit” under test**

As it’s impractical to reason in terms of interactions for more than a handful of classes or methods, when doing so it’s easier to consider a specific method or class as the unit under test. While I can’t really tell why most of us developers insist on testing methods or classes one by one, I feel like maybe the overuse of mocking libraries is one of the reasons.

Anyway, when considering methods or classes as the unit to test, a common pattern is that classes (let’s consider classes A, B and C in that example) each come with a corresponding test class (ATest, BTest and CTest). Should A call methods of B which itself calls methods C, then ATest will “mock” B and BTest will mock C.  
Now, if one ever wants to merge B and C or to introduce new classes D and E somewhere in the middle of those interactions, our tests will get in the way of that change because they focus on interactions between our components, and not on the external behavior of the whole module.

On the other hand, when using a “real” repository one can build [whatever unit makes sense](https://martinfowler.com/bliki/UnitTest.html#SolitaryOrSociable) for observing the behavior of the software. Tens of classes could be involved and that would be OK.

**4. A solution**

To be clear, with discipline one could technically overcome all these drawbacks while still using mocking libraries. But sharing a lot of mocking setups tends to be challenging, and I would say that our effort would be better invested in using the techniques described next.

Regarding the issues discussed above, does that mean that we should use our actual “production” repository within our tests? Of course not, it would bring us back to square one. While we do want test doubles for our repositories, we would better have ones that act like the production ones.

And this precisely what Fake Objects are about: working implementations that take a few shortcuts to be more suitable for testing (but less so for production). For a repository, typical shortcuts are that the data only live in memory and are lost when the test process is over, or that there is no support for transactions (which is OK as long as our tests are single-threaded).

Therefore, we’ll build a fake repository. But how could we ensure that our fake repository mirrors the behavior of the real one in a faithful way? Granted, we often don’t ask ourselves that questions when using a mocking library, but as we’re going to use a single fake implementation for all our tests, we better have to be sure it does its job well!

So, why not write a contract for our repository, much like we write [contract tests](https://martinfowler.com/bliki/ContractTest.html) for external services?

**What are we talking about, in the end?**

To sum-up, here is our checklist:

1. hide data access logic behind a simple contract, that of the repository
2. define the static contract of the repository in an interface
3. write tests against that interface to specify the dynamic/behavioral contract of the repository
4. write an implementation of the repository that is suitable for production and passes all the tests
5. write an implementation of the repository that is suitable for tests and passes all the tests

We’ll also demonstrate how to use [Testcontainers](https://www.testcontainers.org/" \t "_blank) to test the production repository against a real DB.

Additionally, while not mandatory it’s advised that the repository speak the language of the surrounding domain, when possible. That means that whatever tools and types are used under hood to manipulate data, inputs and outputs of repositories should only use types of the domain, and methods should have names that have meaning for the domain as well (e.g. a persisting method could be named “store” or “register” rather than “save” or “insert”).

**Enough talk! A demonstration please!**

The following sections will each focus on a part of a sample codebase accessible on Github: [Maltcommunity/repository-test-example](https://github.com/Maltcommunity/repository-test-example" \t "_blank).  
It features a simple application for users to manage their “tasks”. A task is a very simple concept here: it’s composed of a summary and a description, and it belongs to an owner. That trivial domain is enough for our demonstration.

As we’ll only have a glimpse at it, feel free to browse the code for more details.

**Repository interface (i.e. static contract)**

The task-domain module defines [the TaskRepository interface](https://github.com/Maltcommunity/repository-test-example/blob/main/task-domain/src/main/kotlin/com/malt/task/TaskRepository.kt). It’s the contract the domain wishes to use for persisting and fetching tasks:

interface TaskRepository {  
 fun save(task: Task)  
 fun find(taskId: TaskId): Task?  
 fun find(specification: TaskSpecification): Stream<Task>  
 fun delete(taskId: TaskId)  
}

Note: here Task, TaskId, and TaskSpecification aren’t technical concepts but are part of the domain. Maybe an implementation of that repository will use an ORM under the hood with another TaskDTO class and JPA annotations, but the domain doesn’t need to know it. Again, while not mandatory it makes for a cleaner separation of concerns and each side may solely focus on its job.

**Tests for the repository (i.e. dynamic/behavioral contract)**

In its test sources, the task-domain module also defines (and shares) a “contract” (a test class really) with which TaskRepository’s implementations must comply.  
That class is abstract and requires its concrete children to provide an instance of the TaskRepository implementation to test.

(In case you’re wondering, “sut” is a common short name meaning “System Under Test”. Therefore, in the following snippet it could as well have been named taskRepository. All code snippets in this article use this convention.)

abstract class TaskRepositoryContract { abstract fun buildRepositoryUnderTest(): TaskRepository private val sut by lazy { buildRepositoryUnderTest() } @Test  
 fun `should find a saved task using its ID`() {  
 // given  
 val unsavedTask = TaskBuilder().build()  
 expectThat(sut.find(unsavedTask.id)).isNull() // when  
 sut.save(unsavedTask) // then  
 assertThat(sut.find(unsavedTask.id)).isEqualTo(unsavedTask)  
 } // ... other tests omitted for brevity...  
}

Notes:

* A contract test may also ask for other dependencies needed by the test, when those dependencies are also required to build the system under test.  
  For instance, to test a repository which depends on the current time, the contract (abstract) test class would require the concrete test class to build the repository with a clock stub and give it that stub, so that the test can control the time seen by the repository under test:

abstract fun getClockStubUsedByRepository(): ClockStub  
abstract fun buildRepositoryUnderTest(): TaskRepositoryprivate val clock by lazy { getClockStubUsedByRepository() }  
private val sut by lazy { buildRepositoryUnderTest() }@Test fun someTest() {  
 clock.setTime(/\* ... \*/)  
 // ...  
}

* A contract test may require its concrete children to implement actions needed by the test but which depend on the implementation under test, for instance a clearData() method.

**Actual implementation using PostgreSQL and Spring’s JdbcTemplate**

While the task-domain module doesn’t depend on any framework, we’re building a Spring application so [our PostgresqlTaskRepository adapter](https://github.com/Maltcommunity/repository-test-example/blob/main/adapters/task-persistence/src/main/kotlin/com/malt/task/PostgresqlTaskRepository.kt) will happily use the power of it:

@Transactional(propagation = MANDATORY)  
@Repository  
internal class PostgresqlTaskRepository(  
 private val jdbcTemplate: NamedParameterJdbcTemplate  
) : TaskRepository { private val taskMapper = TaskMapper() override fun save(task: Task) {  
 jdbcTemplate.update("""  
 INSERT INTO ${Table.name} (  
 ${Cols.id}, ${Cols.creationDate}, ${Cols.ownerId},  
 ${Cols.summary}, ${Cols.description}   
 ) VALUES (  
 :id, :creationDate, :ownerId, :summary, :description  
 ) ON CONFLICT (${Cols.id}) DO UPDATE SET  
 ${Cols.summary} = :summary,  
 ${Cols.description} = :description  
 """.trimIndent(), mapOf(  
 "id" to task.id.value,  
 "creationDate" to task.creationDate,  
 "ownerId" to task.ownerId.value,  
 "summary" to task.summary,  
 "description" to task.description,  
 ))  
 } override fun find(taskId: TaskId): Task? { /\* ... \*/ } // ... other methods omitted for brevity...  
}

Note: at Malt, we generally use [Jooq](https://www.jooq.org/" \t "_blank) rather than JdbcTemplate. This gives us some additional type safety, and we’ve built some tooling to easily consume result sets when many-to-many relationships are involved.

**Fake implementation using an in-memory list**

The task-domain module doesn’t only specify the contract of a TaskRepository, it also provides [a Fake — in-memory — implementation of it](https://github.com/Maltcommunity/repository-test-example/blob/main/task-domain/src/test/kotlin/com/malt/task/test/InMemoryTaskRepository.kt), as a courtesy to other modules (and because it itself uses it):

class InMemoryTaskRepository : TaskRepository { private val tasks = mutableListOf<Task>() override fun save(task: Task) {  
 delete(task.id)  
 tasks.add(task)  
 } override fun find(taskId: TaskId) =  
 tasks.firstOrNull { it.id == taskId } override fun find(specification: TaskSpecification) =  
 tasks.stream().filter { specification.isSatisfiedBy(it) } override fun delete(taskId: TaskId) {  
 tasks.removeIf { it.id == taskId }  
 } fun clear() { // not part of the repository contract  
 tasks.clear()  
 }  
}

You read it well, that is all there is to it.

To be honest, as the sample codebase also demonstrates the use of Specifications, InMemoryTaskRepository is a bit shorter than it would if defining all of its behavior by itself. But even considering all the code making it work, it’s shorter than the production implementation and that difference would only increase with the addition of more operations.

**Running the contract tests against the actual implementation**

It’s time to show how we ensure that PostgresqlTaskRepository complies with the contract of a TaskRepository.  
[PostgresqlTaskRepositoryTest](https://github.com/Maltcommunity/repository-test-example/blob/main/adapters/task-persistence/src/test/kotlin/com/malt/task/PostgresqlTaskRepositoryTest.kt) extends the contract class and uses Spring and Testcontainers to build the repository:

@JdbcTemplatePostgreqlTest  
@Import(PostgresqlTaskRepository::class)  
@Sql("/task-schema.sql")  
class PostgresqlTaskRepositoryTest : TaskRepositoryContract() { @Inject  
 lateinit var sut: PostgresqlTaskRepository override fun buildRepositoryUnderTest() = sut  
}

Admittedly, there’s a lot of “magic” happening there. The thing to know is that JdbcTemplatePostgreqlTest is a meta-annotation that tells both JUnit and Spring what to do in order to: 1) spawn a PostgreSQL container using Testcontainer, 2) create a unique — throw-away — database, 3) start a Spring context with a datasource configured to use that DB, 4) inject our repository into the test class, 5) clean everything after the test.  
As it would be inconvenient to put all the code here, please have a look at [the test utilities](https://github.com/Maltcommunity/repository-test-example/tree/main/common/test-utils/src/main/java/com/malt/test/postgres) for more details.

Also, that code lies a bit in that it should be annotated with @Transactional in order for Spring to automatically open a transaction before each test, and rollback it after the test (which helps in having fast independent tests). Because of how Spring works, that annotation has been put on the contract class, which is a sad but acceptable tradeoff, [as explained here](https://github.com/Maltcommunity/repository-test-example/blob/main/task-domain/src/test/kotlin/com/malt/task/TaskRepositoryContract.kt#L15).

**Running the contract tests against the fake implementation**

Proving that the “fake” — in-memory — implementation works is a no-brainer. Here is all the code of [InMemoryTaskRepositoryTest](https://github.com/Maltcommunity/repository-test-example/blob/main/task-domain/src/test/kotlin/com/malt/task/test/InMemoryTaskRepositoryTest.kt" \t "_blank):

class InMemoryTaskRepositoryTest : TaskRepositoryContract() {  
 override fun buildRepositoryUnderTest() = InMemoryTaskRepository()  
}

**Writing tests making use of the fake implementation**

We can finally observe the benefits of our work!

All tests defined in the task-shell-app module as well as some of the task-domain module use InMemoryTaskRepository. I’ll only show extracts of them, slightly reworked so as to highlight how the fake repository is used.

Here is a test specifying the behavior of an application service dealing with tasks of the current user. The repository is queried to observe the result of adding a task, in the same way as production code would call it:

val fixtures = TaskCommandsFixtures()  
val clock = fixtures.clock  
val repository = fixtures.taskRepository  
val sut = fixtures.currentUserTaskService  
  
@Test  
fun `should save task and return it when adding a new task`() {  
 // given  
 val taskSummary = "Some task"  
 val taskDescription = "the aim of it being to do something"  
 val expectedCreationDate = clock.stop().offsetDateTime // when  
 val createdTaskReturned = sut.addTaskForUser(  
 summary = taskSummary, description = taskDescription) // then a task has been saved  
 val userTasks = repository.find(TaskOwnerIdIs(idOfCurrentUser))  
 expectThat(userTasks) hasSize 1 // and that task is the one that has been returned  
 // with expected attributes  
 val createdTaskFetchedFromRepository = userTasks[0]  
 expectThat(createdTaskFetchedFromRepository) {  
 isEqualToComparingProperties(createdTaskReturned)  
 get { ownerId } isEqualTo idOfCurrentUser  
 get { creationDate } isEqualTo expectedCreationDate  
 get { summary } isEqualTo taskSummary  
 get { description } isEqualTo taskDescription  
 }  
}

As pointed out by my colleague [Jean Helou](https://www.malt.fr/profile/jeanhelou) when reviewing this article, in some cases it might be possible (and desirable) to verify that the right data have been persisted without querying the repository at all, when the system provides an indirect mean to observe those data. It’s indeed possible for the above test: one could check the result of calling sut.findAllUserTasks(), and that would make our system under test a complete black box.

Now, here is another example of a test specifying the behavior of the highest-level component of our application. It saves tasks using the repository, and then verify the output of the shell command “merge-tasks”:

val fixtures = TaskCommandsFixtures()  
val repository = fixtures.taskRepository  
val sut = TaskCommands(fixtures.currentUserTaskService)@Test  
fun `should display task resulting from merging two other ones`() {  
 // given two tasks  
 val taskId1 = givenAnExistingTaskBelongingToCurrentUser(1).id  
 val taskId2 = givenAnExistingTaskBelongingToCurrentUser(2).id // when using the command to merge them together  
 val actualDisplay = sut.mergeTasks(taskId1.value, taskId2.value) // then a representation of the resulting task is displayed  
 expectThat(actualDisplay) isEqualTo """  
 Tasks task-id-1 and task-id-2 successfully merged:  
  
 task-id-2  
 summary of task 2 — summary of task 1  
 description of task 2  
  
 description of task 1  
 """.trimIndent()  
}fun givenAnExistingTaskBelongingToCurrentUser(id: Int): Task {  
 val task = TaskBuilder(  
 id = id,  
 ownerId = idOfCurrentUser,  
 summary = "summary of task $id",  
 description = "description of task $id"  
 ).build()  
 taskRepository.save(task)  
 return task  
}

In the above test many components work together, and that’s OK as long as they are all strongly involved in the observed behavior. This is not to say that each and every test should load all the components of the codebase, but why write complex and rigid tests full of stubs and mocks — with less confidence in the result — when one can simply observe the actual system?

You may also have noted that the repository is never directly instantiated, nor are most components used by the tests. Instead a “fixtures” class is in charge of instantiating and wiring our components (and it also provides some utility methods). This make our tests independent on the wiring: should any component gain or loose a dependency, only the fixtures class must change but tests remain unimpacted.

**When not to use these techniques**

While I wouldn’t say that there are situations where the techniques discussed in this article would be harmful, it is certain that they may introduce an unnecessary overhead for the simplest cases.

For instance, should you write a dumb CRUD API that directly delegates to a datastore, then you would better write the simplest code possible, without indirection, covered by a single integration test.

A good rule of thumb is: if a repository is used in many places (even indirectly) then these techniques are valuable. It happens that it should be the case of most business logic.

**What does a typical TDD flow look like with these techniques?**

I’m glad you asked! While everyone will come with it’s own way of working, here is a proposed flow that works well for us:

1. As usual, write a test for the feature at hand.
2. Should that test need to interact with a repository behavior that is undefined yet: add the requested method(s) to the repository interface.
3. Don’t attempt to write the production implementation right now (just make it compile and throw an exception if called), but instead write the minimal fake implementation that will make your test pass.
4. Finish writing whatever code will make your test pass.
5. Now you can write tests for the new behavior of the repository, and maybe improve your fake implementation to make it match all expected cases.
6. Finally, you’re free to concentrate on the production implementation of the repository when ready to do so.

The idea here is to avoid context switching by staying focused on your feature until it’s done. The implementation of the production repository is a detail that can wait. Also writing only the requested amount of fake implementation to make your test pass should take no more effort than using a mocking library, meaning you can still apply the principles of [the GOOS book](http://www.growing-object-oriented-software.com/).

**Conclusion**

Let’s recap! What did we obtain?

* More comprehensive tests, integrating more production code instead of “mocking” everything. Such tests increase our confidence in the overall system.
* Less rigid tests, allowing us to freely refactor our code. Again, this is allowed by a broader definition of a what a tested “unit” should be and by focusing less on interactions between our internal components.
* Fast tests for the business logic so that we can write as many as we want and still have instant feedback.
* A few slower tests to ensure our actual repository works as expected with a real database. Because there are so few (and still runs fast enough thanks to Testcontainers), they don’t impact us much negatively.
* A guarantee that our actual and fake repositories behave the same way.

Additionally, we’ve seen that working this way doesn’t get in the way of TDD (quite the contrary) and that it doesn’t require more effort. Indeed when we think about it, it just moves the effort from writing hundreds of mock setups to building a single richer fake object.

Once again, to be clear: mocking libraries may be useful to test specific kinds of interactions. Used blindly they can do more harm than good.  
Also, though this article only demonstrated how to code a fake repository, you might want to also code your own stubs, mocks or spies — and not only for repositories —and obtain similar benefits.

Finally, I’ve suggested earlier that there might be a relationship between what’s been demonstrated, and the use of Domain Driven Design or Hexagonal Architecture. It just so happens that both encourage us to focus on our business code and to move infrastructure code at the boundaries of our system, and therefore they push us towards such solutions. But you needn’t use those principles: applying the above techniques should give immediate benefits for any non-trivial code using repositories.

To conclude, I would say that these techniques foster good habits: you start following this path, and soon enough you realize that applying one of them automatically triggers the application of the other ones, much as reflex :-)

*Only (slightly reworked) parts of the sample codebase have been shown in the article. For more details, feel free to clone or browse the repository on Github: [Maltcommunity/repository-test-example](https://github.com/Maltcommunity/repository-test-example" \t "_blank).*