The Practical Test Pyramid

The "Test Pyramid" is a metaphor that tells us to group software tests into buckets of different granularity. It also gives an idea of how many tests we should have in each of these groups. Although the concept of the Test Pyramid has been around for a while, teams still struggle to put it into practice properly. This article revisits the original concept of the Test Pyramid and shows how you can put this into practice. It shows which kinds of tests you should be looking for in the different levels of the pyramid and gives practical examples on how these can be implemented.

26 February 2018

Ham Vocke

Ham is a software developer and consultant at ThoughtWorks in Germany. Being tired of deploying software manually at 3 a.m., he added continuous delivery and diligent automation to his toolbox and set out to help teams deliver high-quality software reliably and efficiently. He makes up for the time gained by annoying people with his antics.

TESTING

TRANSLATIONS: Chinese

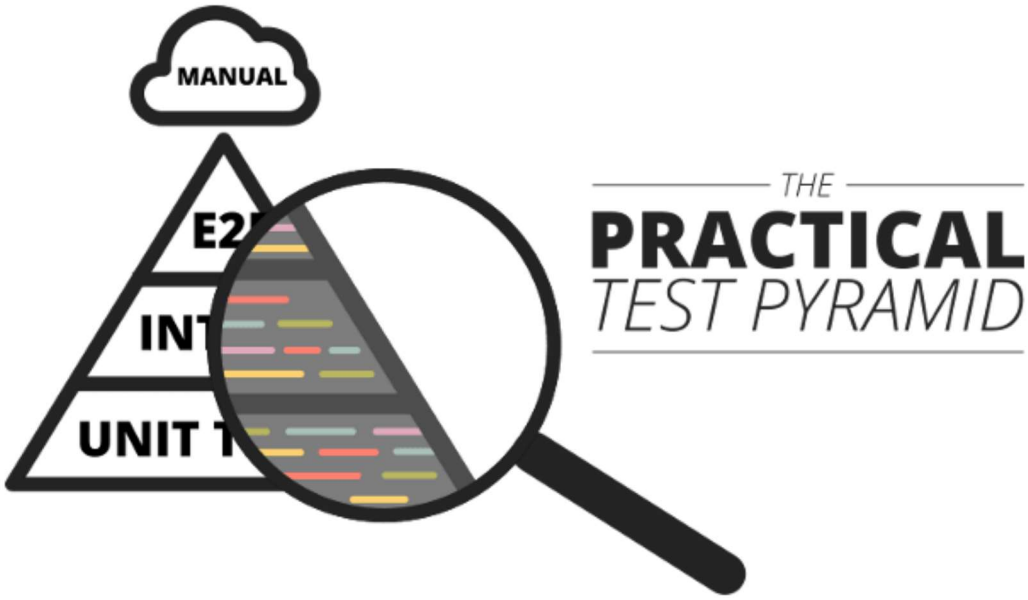
CONTENTS

The Importance of (Test) Automation

The Test Pyramid
Tools and Libraries We'll Look at
The Sample Application
  Functionality
  High-level Structure
  Internal Architecture
Unit tests
  What's a Unit?
  Sociable and Solitary
  Mocking and Stubbing
  What to Test?
  Test Structure
  Implementing a Unit Test
Integration Tests
  Database Integration
  Integration With Separate Services
Contract Tests
  Consumer Test (our team)
  Provider Test (the other team)
  Provider Test (our team)
UI Tests
End-to-End Tests
  User Interface End-to-End Test
  REST API End-to-End Test
Acceptance Tests — Do Your Features Work Correctly?
Exploratory Testing
The Confusion About Testing Terminology
Putting Tests Into Your Deployment Pipeline
Avoid Test Duplication
Writing Clean Test Code
Conclusion

SIDEBARS

But I Really Need to Test This Private Method
Specialised Test Helpers



Production-ready software requires testing before it goes into production. As the discipline of software development matured, software testing approaches have matured too. Instead of having myriads of manual software testers, development teams have moved towards automating the biggest portion of their testing efforts. Automating their tests allows teams to know whether their software is broken in a matter of seconds and minutes instead of days and weeks.

The drastically shortened feedback loop fuelled by automated tests goes hand in hand with agile development practices, continuous delivery and DevOps culture. Having an effective software testing approach allows teams to move fast and with confidence.

This article explores what a well-rounded test portfolio should look like to be responsive, reliable and maintainable - regardless of whether you're building a microservices architecture, mobile apps or IoT ecosystems. We'll also get into the details of building effective and readable automated tests.

The Importance of (Test) Automation

Software has become an essential part of the world we live in. It has outgrown its early sole purpose of making businesses more efficient. Today companies try to find ways to become first-class digital companies. As users everyone of us interacts with an ever-increasing amount of software every day. The wheels of innovation are turning faster.

If you want to keep pace you'll have to look into ways to deliver your software faster without sacrificing its quality. **Continuous delivery**, a practice where you automatically ensure that your software can be released into production any time, can help you with that. With continuous delivery you use a **build pipeline** to automatically test your software and deploy it to your testing and production environments.

Building, testing and deploying an ever-increasing amount of software manually soon becomes impossible — unless you want to spend all your time with manual, repetitive work instead of delivering working software. Automating everything — from build to tests, deployment and infrastructure — is your only way forward.

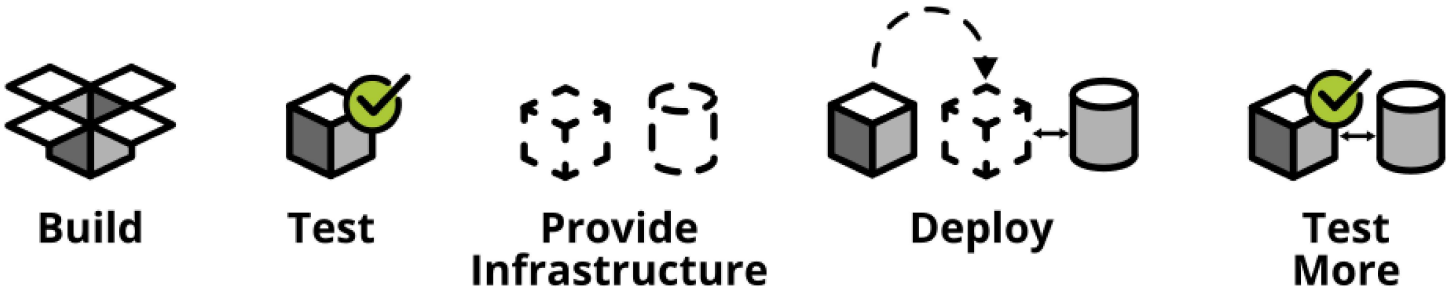


Figure 1: Use build pipelines to automatically and reliably get your software into production

Traditionally software testing was overly manual work done by deploying your application to a test environment and then performing some black-box style testing e.g. by clicking through your user interface to see if anything's broken. Often these tests would be specified by test scripts to ensure the testers would do consistent checking.

It's obvious that testing all changes manually is time-consuming, repetitive and tedious. Repetitive is boring, boring leads to mistakes and makes you look for a different job by the end of the week.

Luckily there's a remedy for repetitive tasks: *automation*.

Automating your repetitive tests can be a big game changer in your life as a software developer. Automate these tests and you no longer have to mindlessly follow click protocols in order to check if your software still works correctly. Automate your tests and you can change your codebase without batting an eye. If you've ever tried doing a large-scale refactoring without a proper test suite I bet you know what a terrifying experience this can be. How would you know if you accidentally broke stuff along the way? Well, you click through all your manual test cases, that's how. But let's be honest: do you really enjoy that? How about making even large-scale changes and knowing whether you broke stuff within seconds while taking a nice sip of coffee? Sounds more enjoyable if you ask me.

The Test Pyramid

If you want to get serious about automated tests for your software there is one key concept you should know about: the **test pyramid**. Mike Cohn came up with this concept in his book *Succeeding with Agile*. It's a great visual metaphor telling you to think about different layers of testing. It also tells you how much testing to do on each layer.

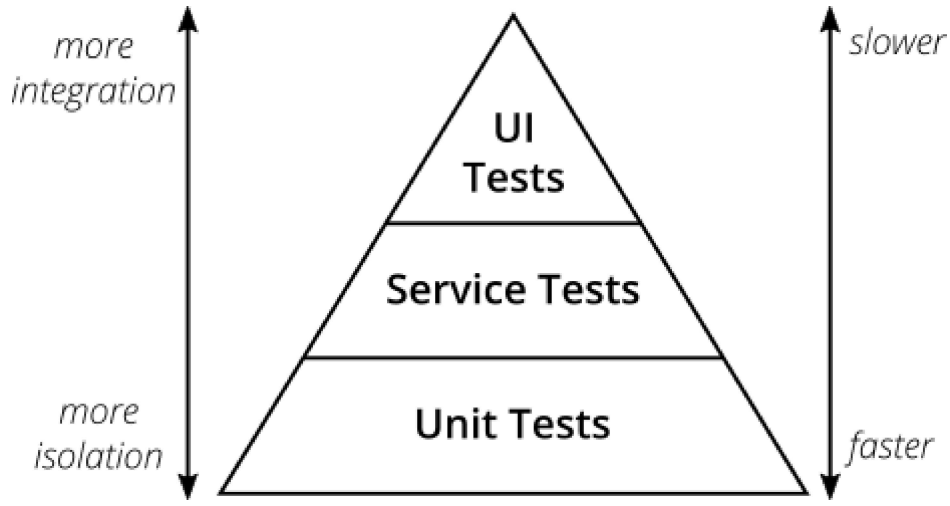


Figure 2: The Test Pyramid

Mike Cohn's original test pyramid consists of three layers that your test suite should consist of (bottom to top):

1. Unit Tests
2. Service Tests
3. User Interface Tests

Unfortunately the concept of the test pyramid falls a little short if you take a closer look. Some argue that either the naming or some conceptual aspects of Mike Cohn's test pyramid are not ideal, and I have to agree. From a modern point of view the test pyramid seems overly simplistic and can therefore be misleading.

Still, due to its simplicity the essence of the test pyramid serves as a good rule of thumb when it comes to establishing your own test suite. Your best bet is to remember two things from Cohn's original test pyramid:

1. Write tests with different granularity
2. The more high-level you get the fewer tests you should have

Stick to the pyramid shape to come up with a healthy, fast and maintainable test suite: Write *lots* of small and fast *unit tests*. Write *some* more coarse-grained tests and *very few* high-level tests that test your application from end to end. Watch out that you don't end up with a test ice-cream cone that will be a nightmare to maintain and takes way too long to run.

Don't become too attached to the names of the individual layers in Cohn's test pyramid. In fact they can be quite misleading: *service test* is a term that is hard to grasp (Cohn himself talks about the observation that a lot of developers completely ignore this layer). In the days of single page application frameworks like react, angular, ember.js and others it becomes apparent that *UI tests* don't have to be on the highest level of your pyramid - you're perfectly able to unit test your UI in all of these frameworks.

Given the shortcomings of the original names it's totally okay to come up with other names for your test layers, as long as you keep it consistent within your codebase and your team's discussions.

Tools and Libraries We'll Look at

- JUnit: our test runner
- Mockito: for mocking dependencies
- Wiremock: for stubbing out external services
- Pact: for writing CDC tests
- Selenium: for writing UI-driven end-to-end tests
- REST-assured: for writing REST API-driven end-to-end tests

The Sample Application

I've written a simple microservice including a test suite with tests for the different layers of the test pyramid.

The sample application shows traits of a typical microservice. It provides a REST interface, talks to a database and fetches information from a third-party REST service. It's implemented in Spring Boot and should be understandable even if you've never worked with Spring Boot before.

Make sure to check out the code on Github. The readme contains instructions you need to run the application and its automated tests on your machine.

Functionality

The application's functionality is simple. It provides a REST interface with three endpoints:

`GET /hello`	Returns "*Hello World*". Always.
`GET /hello/{lastname}`	Looks up the person with the provided last name. If the person is known, returns "*Hello {Firstname} {Lastname}*".
`GET /weather`	Returns the current weather conditions for *Hamburg, Germany*.

High-level Structure

On a high-level the system has the following structure:

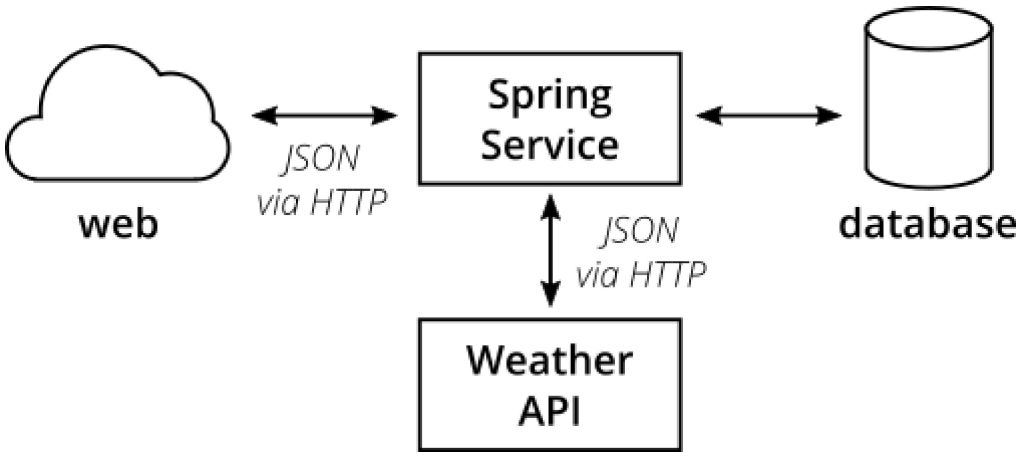


Figure 3: the high level structure of our microservice system

Our microservice provides a REST interface that can be called via HTTP. For some endpoints the service will fetch information from a database. In other cases the service will call an external weather API via HTTP to fetch and display current weather conditions.

Internal Architecture

Internally, the Spring Service has a Spring-typical architecture:

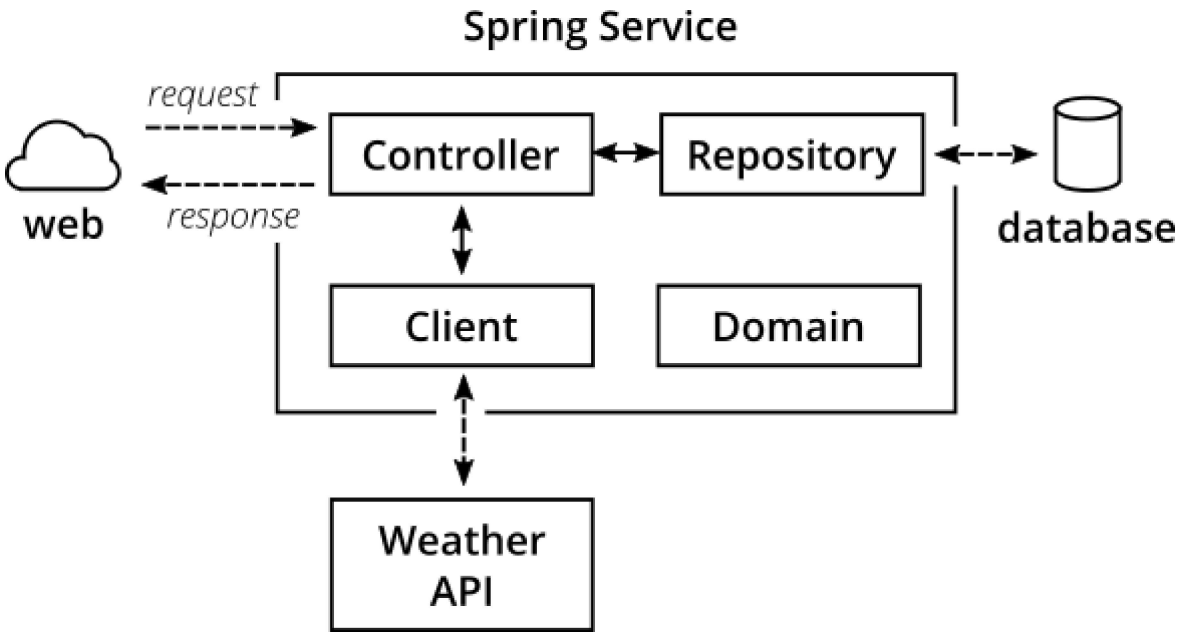


Figure 4: the internal structure of our microservice

- `Controller` classes provide *REST* endpoints and deal with *HTTP* requests and responses

- `Repository` classes interface with the *database* and take care of writing and reading data to/from persistent storage
- `Client` classes talk to other APIs, in our case it fetches *JSON* via *HTTPS* from the darksky.net weather API
- `Domain` classes capture our domain model including the domain logic (which, to be fair, is quite trivial in our case).

Experienced Spring developers might notice that a frequently used layer is missing here: Inspired by Domain-Driven Design a lot of developers build a *service layer* consisting of *service* classes. I decided not to include a service layer in this application. One reason is that our application is simple enough, a service layer would have been an unnecessary level of indirection. The other one is that I think people overdo it with service layers. I often encounter codebases where the entire business logic is captured within service classes. The domain model becomes merely a layer for data, not for behaviour (an Anemic Domain Model). For every non-trivial application this wastes a lot of potential to keep your code well-structured and testable and does not fully utilise the power of object orientation.

Our repositories are straightforward and provide simple CRUD functionality. To keep the code simple I used Spring Data. Spring Data gives us a simple and generic CRUD repository implementation that we can use instead of rolling our own. It also takes care of spinning up an in-memory database for our tests instead of using a real PostgreSQL database as it would in production.

Take a look at the codebase and make yourself familiar with the internal structure. It will be useful for our next step: Testing the application!

Unit tests

The foundation of your test suite will be made up of unit tests. Your unit tests make sure that a certain unit (your *subject under test*) of your codebase works as intended. Unit tests have the narrowest scope of all the tests in your test suite. The number of unit tests in your test suite will largely outnumber any other type of test.

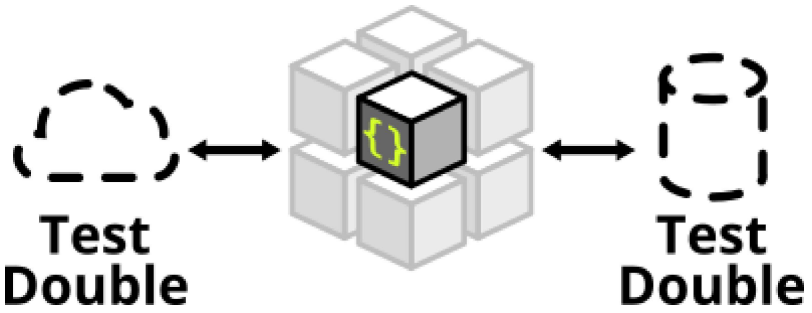


Figure 5: A unit test typically replaces external collaborators with test doubles

What's a Unit?

If you ask three different people what "*unit*" means in the context of unit tests, you'll probably receive four different, slightly nuanced answers. To a certain extent it's a matter of your own definition and it's okay to have no canonical answer.

If you're working in a functional language a *unit* will most likely be a single function. Your unit tests will call a function with different parameters and ensure that it returns the expected values. In an object-oriented language a unit can range from a single method to an entire class.

Sociable and Solitary

Some argue that all collaborators (e.g. other classes that are called by your class under test) of your subject under test should be substituted with *mocks* or *stubs*

to come up with perfect isolation and to avoid side-effects and a complicated test setup. Others argue that only collaborators that are slow or have bigger side effects (e.g. classes that access databases or make network calls) should be stubbed or mocked.

Occasionally people label these two sorts of tests as **solitary unit tests** for tests that stub all collaborators and **sociable unit tests** for tests that allow talking to real collaborators (Jay Fields' Working Effectively with Unit Tests coined these terms). If you have some spare time you can go down the rabbit hole and read more about the pros and cons of the different schools of thought.

At the end of the day it's not important to decide if you go for solitary or sociable unit tests. Writing automated tests is what's important. Personally, I find myself using both approaches all the time. If it becomes awkward to use real collaborators I will use mocks and stubs generously. If I feel like involving the real collaborator gives me more confidence in a test I'll only stub the outermost parts of my service.

Mocking and Stubbing

Mocks and Stubs are two different kinds of Test Doubles (there are more than these two). A lot of people use the terms Mock and Stub interchangeably. I think it's good to be precise and keep their specific properties in mind. You can use test doubles to replace objects you'd use in production with an implementation that helps you with testing.

In plain words it means that you replace a real thing (e.g. a class, module or function) with a fake version of that thing. The fake version looks and acts like the real thing (answers to the same method calls) but answers with canned responses that you define yourself at the beginning of your unit test.

Using test doubles is not specific to unit testing. More elaborate test doubles can be used to simulate entire parts of your system in a controlled way. However, in

unit testing you're most likely to encounter a lot of mocks and stubs (depending of whether you're the sociable or solitary kind of developer), simply because lots of modern languages and libraries make it easy and comfortable to set up mocks and stubs.

Regardless of your technology choice, there's a good chance that either your language's standard library or some popular third-party library will provide you with elegant ways to set up mocks. And even writing your own mocks from scratch is only a matter of writing a fake class/module/function with the same signature as the real one and setting up the fake in your test.

Your unit tests will run very fast. On a decent machine you can expect to run thousands of unit tests within a few minutes. Test small pieces of your codebase in isolation and avoid hitting databases, the filesystem or firing HTTP queries (by using mocks and stubs for these parts) to keep your tests fast.

Once you got a hang of writing unit tests you will become more and more fluent in writing them. Stub out external collaborators, set up some input data, call your subject under test and check that the returned value is what you expected. Look into Test-Driven Development and let your unit tests guide your development; if applied correctly it can help you get into a great flow and come up with a good and maintainable design while automatically producing a comprehensive and fully automated test suite. Still, it's no silver bullet. Go ahead, give it a real chance and see if it feels right for you.

But I *Really* Need to Test This Private Method

If you ever find yourself in a situation where you *really really* need to test a private method you should take a step back and ask yourself why.

I'm pretty sure this is more of a design problem than a scoping problem. Most likely you feel the need to test a private method because it's complex and testing this method through the public interface of the class requires a lot of awkward setup.

Whenever I find myself in this situation I usually come to the conclusion that the class I'm testing is already too complex. It's doing too much and violates the *single responsibility* principle - the *S*

of the five SOLID principles.

The solution that often works for me is to split the original class into two classes. It often only takes one or two minutes of thinking to find a good way to cut the one big class into two smaller classes with individual responsibility. I move the private method (that I urgently want to test) to the new class and let the old class call the new method. Voilà, my awkward-to-test private method is now public and can be tested easily. On top of that I have improved the structure of my code by adhering to the single responsibility principle.

What to Test?

The good thing about unit tests is that you can write them for all your production code classes, regardless of their functionality or which layer in your internal structure they belong to. You can unit tests controllers just like you can unit test repositories, domain classes or file readers. Simply stick to the **one test class per production class** rule of thumb and you're off to a good start.

A unit test class should at least **test the *public* interface of the class**. Private methods can't be tested anyways since you simply can't call them from a different test class. *Protected* or *package-private* are accessible from a test class (given the package structure of your test class is the same as with the production class) but testing these methods could already go too far.

There's a fine line when it comes to writing unit tests: They should ensure that all your non-trivial code paths are tested (including happy path and edge cases). At the same time they shouldn't be tied to your implementation too closely.

Why's that?

Tests that are too close to the production code quickly become annoying. As soon as you refactor your production code (quick recap: refactoring means changing the internal structure of your code without changing the externally visible behaviour) your unit tests will break.

This way you lose one big benefit of unit tests: acting as a safety net for code changes. You rather become fed up with those stupid tests failing every time you refactor, causing more work than being helpful; and whose idea was this stupid testing stuff anyways?

What do you do instead? Don't reflect your internal code structure within your unit tests. Test for observable behaviour instead. Think about

if I enter values x and y, will the result be z?

instead of

if I enter x and y, will the method call class A first, then call class B and then return the result of class A plus the result of class B?

Private methods should generally be considered an implementation detail. That's why you shouldn't even have the urge to test them.

I often hear opponents of unit testing (or TDD) arguing that writing unit tests becomes pointless work where you have to test all your methods in order to come up with a high test coverage. They often cite scenarios where an overly eager team lead forced them to write unit tests for getters and setters and all other sorts of trivial code in order to come up with 100% test coverage.

There's so much wrong with that.

Yes, you should *test the public interface*. More importantly, however, you **don't test trivial code**. Don't worry, Kent Beck said it's ok. You won't gain anything from testing simple *getters* or *setters* or other trivial implementations (e.g. without any conditional logic). Save the time, that's one more meeting you can attend, hooray!

Test Structure

A good structure for all your tests (this is not limited to unit tests) is this one:

1. Set up the test data
2. Call your method under test
3. Assert that the expected results are returned

There's a nice mnemonic to remember this structure: "*Arrange, Act, Assert*". Another one that you can use takes inspiration from BDD . It's the "*given*", "*when*", "*then*" triad, where *given* reflects the setup, *when* the method call and *then* the assertion part.

This pattern can be applied to other, more high-level tests as well. In every case they ensure that your tests remain easy and consistent to read. On top of that tests written with this structure in mind tend to be shorter and more expressive.

> **Specialised Test Helpers**
>
> It's a thing of beauty that you can write unit tests for your entire codebase, regardless of what layer of your application's architecture you're on. The example shows a simple unit test for a controller. Unfortunately, when it comes to Spring's controllers there's a downside to this approach: Spring MVC's controller make heavy use of annotations to declare which paths they're listening on, which HTTP verbs to use, which parameters they parse from the URL path or query params and so on. Simply invoking a controller's method within your unit tests won't test all of these crucial things. Luckily, the Spring folks came up with a nice test helper you can use to write better controller tests. Make sure to check out MockMVC. It gives you a nice DSL you can use to fire fake requests against your controller and check that everything's cool. I've included an example in the sample codebase. A lot of frameworks offer test helpers to make testing specific aspects of your codebase more pleasant. Check out the documentation of your framework of choice and see if it offers any useful helpers for your automated tests.

Implementing a Unit Test

Now that we know what to test and how to structure our unit tests we can finally see a real example.

Let's take a simplified version of the `ExampleController` class:

```
@RestController
public class ExampleController {

    private final PersonRepository personRepo;

    @Autowired
    public ExampleController(final PersonRepository personRepo) {
        this.personRepo = personRepo;
    }

    @GetMapping("/hello/{lastName}")
    public String hello(@PathVariable final String lastName) {
        Optional<Person> foundPerson = personRepo.findByLastName(lastName);

        return foundPerson
                .map(person -> String.format("Hello %s %s!",
                        person.getFirstName(),
                        person.getLastName()))
                .orElse(String.format("Who is this '%s' you're talking about?",
                        lastName));
    }
}
```

A unit test for the `hello(lastname)` method could look like this:

```
public class ExampleControllerTest {

    private ExampleController subject;

    @Mock
    private PersonRepository personRepo;

    @Before
    public void setUp() throws Exception {
        initMocks(this);
        subject = new ExampleController(personRepo);
    }

    @Test
    public void shouldReturnFullNameOfAPerson() throws Exception {
        Person peter = new Person("Peter", "Pan");
```

```
        given(personRepo.findByLastName("Pan"))
            .willReturn(Optional.of(peter));

        String greeting = subject.hello("Pan");

        assertThat(greeting, is("Hello Peter Pan!"));
    }

    @Test
    public void shouldTellIfPersonIsUnknown() throws Exception {
        given(personRepo.findByLastName(anyString()))
            .willReturn(Optional.empty());

        String greeting = subject.hello("Pan");

        assertThat(greeting, is("Who is this 'Pan' you're talking about?"));
    }
}
```

We're writing the unit tests using JUnit, the de-facto standard testing framework for Java. We use Mockito to replace the real `PersonRepository` class with a stub for our test. This stub allows us to define canned responses the stubbed method should return in this test. Stubbing makes our test more simple, predictable and allows us to easily setup test data.

Following the *arrange*, *act*, *assert* structure, we write two unit tests - a positive case and a case where the searched person cannot be found. The first, positive test case creates a new person object and tells the mocked repository to return this object when it's called with "*Pan*" as the value for the `lastName` parameter. The test then goes on to call the method that should be tested. Finally it asserts that the response is equal to the expected response.

The second test works similarly but tests the scenario where the tested method does not find a person for the given parameter.

Integration Tests

All non-trivial applications will integrate with some other parts (databases, filesystems, network calls to other applications). When writing unit tests these are usually the parts you leave out in order to come up with better isolation and faster tests. Still, your application will interact with other parts and this needs to be tested. **Integration Tests** are there to help. They test the integration of your application with all the parts that live outside of your application.

For your automated tests this means you don't just need to run your own application but also the component you're integrating with. If you're testing the integration with a database you need to run a database when running your tests. For testing that you can read files from a disk you need to save a file to your disk and load it in your integration test.

I mentioned before that "unit tests" is a vague term, this is even more true for "integration tests". For some people integration testing means to test through the entire stack of your application connected to other applications within your system. I like to treat integration testing more narrowly and test one integration point at a time by replacing separate services and databases with test doubles. Together with contract testing and running contract tests against test doubles as well as the real implementations you can come up with integration tests that are faster, more independent and usually easier to reason about.

Narrow integration tests live at the boundary of your service. Conceptually they're always about triggering an action that leads to integrating with the outside part (filesystem, database, separate service). A database integration test would look like this:

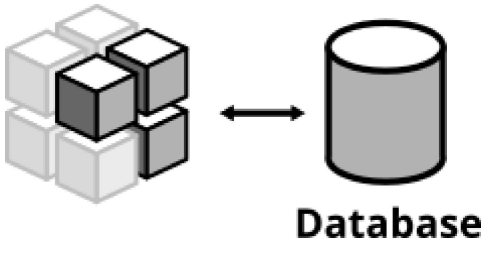


Figure 6: A database integration test integrates your code with a real database

1. start a database
2. connect your application to the database
3. trigger a function within your code that writes data to the database
4. check that the expected data has been written to the database by reading the data from the database

Another example, testing that your service integrates with a separate service via a REST API could look like this:

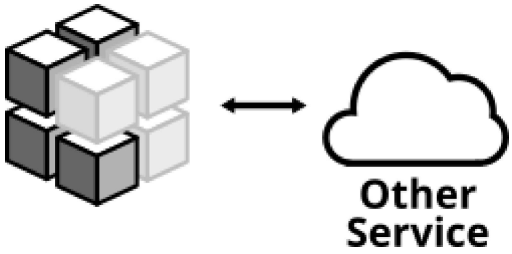


Figure 7: This kind of integration test checks that your application can communicate with a separate service correctly

1. start your application
2. start an instance of the separate service (or a test double with the same interface)
3. trigger a function within your code that reads from the separate service's API
4. check that your application can parse the response correctly

Your integration tests - like unit tests - can be fairly whitebox. Some frameworks allow you to start your application while still being able to mock some other parts of your application so that you can check that the correct interactions have happened.

Write integration tests for all pieces of code where you either *serialize* or *deserialize* data. This happens more often than you might think. Think about:

- Calls to your services' REST API
- Reading from and writing to databases
- Calling other application's APIs

- Reading from and writing to queues
- Writing to the filesystem

Writing integration tests around these boundaries ensures that writing data to and reading data from these external collaborators works fine.

When writing *narrow integration tests* you should aim to run your external dependencies locally: spin up a local MySQL database, test against a local ext4 filesystem. If you're integrating with a separate service either run an instance of that service locally or build and run a fake version that mimics the behaviour of the real service.

If there's no way to run a third-party service locally you should opt for running a dedicated test instance and point at this test instance when running your integration tests. Avoid integrating with the real production system in your automated tests. Blasting thousands of test requests against a production system is a surefire way to get people angry because you're cluttering their logs (in the best case) or even DoS 'ing their service (in the worst case). Integrating with a service over the network is a typical characteristic of a *broad integration test* and makes your tests slower and usually harder to write.

With regards to the test pyramid, integration tests are on a higher level than your unit tests. Integrating slow parts like filesystems and databases tends to be much slower than running unit tests with these parts stubbed out. They can also be harder to write than small and isolated unit tests, after all you have to take care of spinning up an external part as part of your tests. Still, they have the advantage of giving you the confidence that your application can correctly work with all the external parts it needs to talk to. Unit tests can't help you with that.

Database Integration

The `PersonRepository` is the only repository class in the codebase. It relies on *Spring Data* and has no actual implementation. It just extends the `CrudRepository`

interface and provides a single method header. The rest is Spring magic.

```
public interface PersonRepository extends CrudRepository<Person, String> {
    Optional<Person> findByLastName(String lastName);
}
```

With the `CrudRepository` interface Spring Boot offers a fully functional CRUD repository with `findOne`, `findAll`, `save`, `update` and `delete` methods. Our custom method definition (`findByLastName()`) extends this basic functionality and gives us a way to fetch `Persons` by their last name. Spring Data analyses the return type of the method and its method name and checks the method name against a naming convention to figure out what it should do.

Although Spring Data does the heavy lifting of implementing database repositories I still wrote a database integration test. You might argue that this is *testing the framework* and something that I should avoid as it's not our code that we're testing. Still, I believe having at least one integration test here is crucial. First it tests that our custom `findByLastName` method actually behaves as expected. Secondly it proves that our repository used Spring's wiring correctly and can connect to the database.

To make it easier for you to run the tests on your machine (without having to install a PostgreSQL database) our test connects to an in-memory *H2* database.

I've defined H2 as a test dependency in the `build.gradle` file. The `application.properties` in the test directory doesn't define any `spring.datasource` properties. This tells Spring Data to use an in-memory database. As it finds H2 on the classpath it simply uses H2 when running our tests.

When running the real application with the `int` profile (e.g. by setting `SPRING_PROFILES_ACTIVE=int` as environment variable) it connects to a PostgreSQL database as defined in the `application-int.properties`.

I know, that's an awful lot of Spring specifics to know and understand. To get there, you'll have to sift through a lot of documentation. The resulting code is easy on the eye but hard to understand if you don't know the fine details of Spring.

On top of that going with an in-memory database is risky business. After all, our integration tests run against a different type of database than they would in production. Go ahead and decide for yourself if you prefer Spring magic and simple code over an explicit yet more verbose implementation.

Enough explanation already, here's a simple integration test that saves a Person to the database and finds it by its last name:

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class PersonRepositoryIntegrationTest {
    @Autowired
    private PersonRepository subject;

    @After
    public void tearDown() throws Exception {
        subject.deleteAll();
    }

    @Test
    public void shouldSaveAndFetchPerson() throws Exception {
        Person peter = new Person("Peter", "Pan");
        subject.save(peter);

        Optional<Person> maybePeter = subject.findByLastName("Pan");

        assertThat(maybePeter, is(Optional.of(peter)));
    }
}
```

You can see that our integration test follows the same *arrange*, *act*, *assert* structure as the unit tests. Told you that this was a universal concept!

Integration With Separate Services

Our microservice talks to darksky.net, a weather REST API. Of course we want to ensure that our service sends requests and parses the responses correctly.

We want to avoid hitting the real *darksky* servers when running automated tests. Quota limits of our free plan are only part of the reason. The real reason is *decoupling*. Our tests should run independently of whatever the lovely people at darksky.net are doing. Even when your machine can't access the *darksky* servers or the darksky servers are down for maintenance.

We can avoid hitting the real *darksky* servers by running our own, fake *darksky* server while running our integration tests. This might sound like a huge task. Thanks to tools like Wiremock it's easy peasy. Watch this:

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class WeatherClientIntegrationTest {

    @Autowired
    private WeatherClient subject;

    @Rule
    public WireMockRule wireMockRule = new WireMockRule(8089);

    @Test
    public void shouldCallWeatherService() throws Exception {
        wireMockRule.stubFor(get(urlPathEqualTo("/some-test-api-key/53.5511,9.9937")
                .willReturn(aResponse()
                        .withBody(FileLoader.read("classpath:weatherApiResponse.json
                        .withHeader(CONTENT_TYPE, MediaType.APPLICATION_JSON_VALUE)
                        .withStatus(200)));

        Optional<WeatherResponse> weatherResponse = subject.fetchWeather();

        Optional<WeatherResponse> expectedResponse = Optional.of(new WeatherResponse
        assertThat(weatherResponse, is(expectedResponse));
    }
}
```

To use Wiremock we instantiate a `WireMockRule` on a fixed port (`8089`). Using the DSL we can set up the Wiremock server, define the endpoints it should listen on and set canned responses it should respond with.

Next we call the method we want to test, the one that calls the third-party service and check if the result is parsed correctly.

It's important to understand how the test knows that it should call the fake Wiremock server instead of the real *darksky* API. The secret is in our `application.properties` file contained in `src/test/resources`. This is the properties file Spring loads when running tests. In this file we override configuration like API keys and URLs with values that are suitable for our testing purposes, e.g. calling the fake Wiremock server instead of the real one:

```
weather.url = http://localhost:8089
```

Note that the port defined here has to be the same we define when instantiating the `WireMockRule` in our test. Replacing the real weather API's URL with a fake one in our tests is made possible by injecting the URL in our `WeatherClient` class' constructor:

```
@Autowired
public WeatherClient(final RestTemplate restTemplate,
                     @Value("${weather.url}") final String weatherServiceUrl,
                     @Value("${weather.api_key}") final String weatherServiceApiKey)
    this.restTemplate = restTemplate;
    this.weatherServiceUrl = weatherServiceUrl;
    this.weatherServiceApiKey = weatherServiceApiKey;
}
```

This way we tell our `WeatherClient` to read the `weatherUrl` parameter's value from the `weather.url` property we define in our application properties.

Writing *narrow integration tests* for a separate service is quite easy with tools like Wiremock. Unfortunately there's a downside to this approach: How can we ensure that the fake server we set up behaves like the real server? With the current implementation, the separate service could change its API and our tests would still pass. Right now we're merely testing that our `WeatherClient` can parse the responses that the fake server sends. That's a start but it's very brittle. Using *end-to-end tests* and running the tests against a test instance of the real service instead of using a fake service would solve this problem but would make us reliant on the availability of the test service. Fortunately, there's a better solution to this dilemma: Running contract tests against the fake and the real server ensures that the fake we use in our integration tests is a faithful test double. Let's see how this works next.

Contract Tests

More modern software development organisations have found ways of scaling their development efforts by spreading the development of a system across different teams. Individual teams build individual, loosely coupled services without stepping on each others toes and integrate these services into a big, cohesive system. The more recent buzz around microservices focuses on exactly that.

Splitting your system into many small services often means that these services need to communicate with each other via certain (hopefully well-defined, sometimes accidentally grown) interfaces.

Interfaces between different applications can come in different shapes and technologies. Common ones are

- REST and JSON via HTTPS
- RPC using something like gRPC
- building an event-driven architecture using queues

For each interface there are two parties involved: the provider and the consumer. The **provider** serves data to consumers. The **consumer** processes data obtained from a provider. In a REST world a provider builds a REST API with all required endpoints; a consumer makes calls to this REST API to fetch data or trigger changes in the other service. In an asynchronous, event-driven world, a provider (often rather called **publisher**) publishes data to a queue; a consumer (often called **subscriber**) subscribes to these queues and reads and processes data.

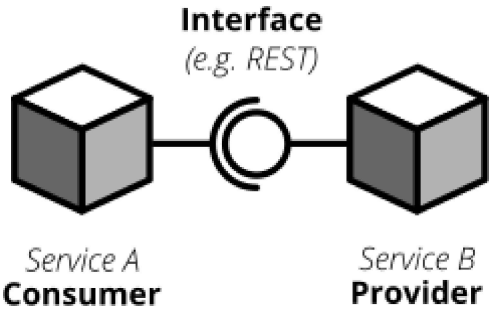


Figure 8: Each interface has a providing (or publishing) and a consuming (or subscribing) party. The specification of an interface can be considered a contract.

As you often spread the consuming and providing services across different teams you find yourself in the situation where you have to clearly specify the interface between these services (the so called **contract**). Traditionally companies have approached this problem in the following way:

- Write a long and detailed interface specification (the *contract*)
- Implement the providing service according to the defined contract
- Throw the interface specification over the fence to the consuming team
- Wait until they implement their part of consuming the interface
- Run some large-scale manual system test to see if everything works

- Hope that both teams stick to the interface definition forever and don't screw up

More modern software development teams have replaced steps 5. and 6. with something more automated: Automated contract tests make sure that the implementations on the consumer and provider side still stick to the defined contract. They serve as a good regression test suite and make sure that deviations from the contract will be noticed early.

In a more agile organisation you should take the more efficient and less wasteful route. You build your applications within the same organisation. It really shouldn't be too hard to talk to the developers of the other services directly instead of throwing overly detailed documentation over the fence. After all they're your co-workers and not a third-party vendor that you could only talk to via customer support or legally bulletproof contracts.

Consumer-Driven Contract tests (CDC tests) let the consumers drive the implementation of a contract. Using CDC, consumers of an interface write tests that check the interface for all data they need from that interface. The consuming team then publishes these tests so that the publishing team can fetch and execute these tests easily. The providing team can now develop their API by running the CDC tests. Once all tests pass they know they have implemented everything the consuming team needs.

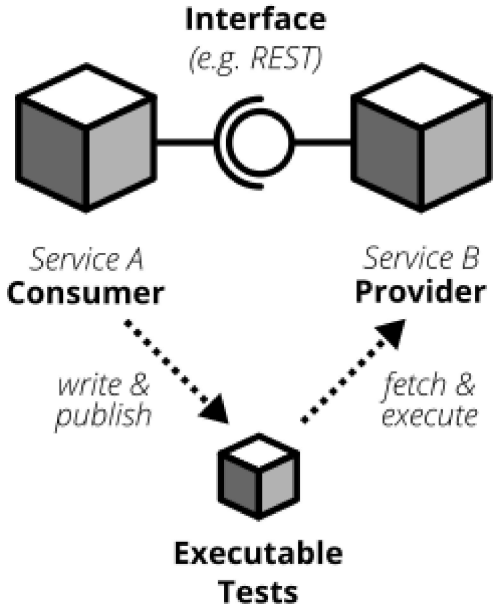


Figure 9: Contract tests ensure that the provider and all consumers of an interface stick to the defined interface contract. With CDC tests consumers of an interface publish their requirements in the form of automated tests; the providers fetch and execute these tests continuously

This approach allows the providing team to implement only what's really necessary (keeping things simple, YAGNI and all that). The team providing the interface should fetch and run these CDC tests continuously (in their build pipeline) to spot any breaking changes immediately. If they break the interface their CDC tests will fail, preventing breaking changes to go live. As long as the tests stay green the team can make any changes they like without having to worry about other teams. The Consumer-Driven Contract approach would leave you with a process looking like this:

- The consuming team writes automated tests with all consumer expectations
- They publish the tests for the providing team
- The providing team runs the CDC tests continuously and keeps them green
- Both teams talk to each other once the CDC tests break

If your organisation adopts a microservices approach, having CDC tests is a big step towards establishing autonomous teams. CDC tests are an automated way to foster team communication. They ensure that interfaces between teams are working at any time. Failing CDC tests are a good indicator that you should walk

over to the affected team, have a chat about any upcoming API changes and figure out how you want to move forward.

A naive implementation of CDC tests can be as simple as firing requests against an API and assert that the responses contain everything you need. You then package these tests as an executable (.gem, .jar, .sh) and upload it somewhere the other team can fetch it (e.g. an artifact repository like Artifactory).

Over the last couple of years the CDC approach has become more and more popular and several tools been build to make writing and exchanging them easier.

Pact is probably the most prominent one these days. It has a sophisticated approach of writing tests for the consumer and the provider side, gives you stubs for separate services out of the box and allows you to exchange CDC tests with other teams. Pact has been ported to a lot of platforms and can be used with JVM languages, Ruby, .NET, JavaScript and many more.

If you want to get started with CDCs and don't know how, Pact can be a sane choice. The documentation can be overwhelming at first. Be patient and work through it. It helps to get a firm understanding for CDCs which in turn makes it easier for you to advocate for the use of CDCs when working with other teams.

Consumer-Driven Contract tests can be a real game changer to establish autonomous teams that can move fast and with confidence. Do yourself a favor, read up on that concept and give it a try. A solid suite of CDC tests is invaluable for being able to move fast without breaking other services and cause a lot of frustration with other teams.

Consumer Test (our team)

Our microservice consumes the weather API. So it's our responsibility to write a **consumer test** that defines our expectations for the contract (the API) between our microservice and the weather service.

First we include a library for writing pact consumer tests in our `build.gradle`:

```
testCompile('au.com.dius:pact-jvm-consumer-junit_2.11:3.5.5')
```

Thanks to this library we can implement a consumer test and use pact's mock services:

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class WeatherClientConsumerTest {

    @Autowired
    private WeatherClient weatherClient;

    @Rule
    public PactProviderRuleMk2 weatherProvider =
            new PactProviderRuleMk2("weather_provider", "localhost", 8089, this);

    @Pact(consumer="test_consumer")
    public RequestResponsePact createPact(PactDslWithProvider builder) throws IOExce
        return builder
                .given("weather forecast data")
                .uponReceiving("a request for a weather request for Hamburg")
                    .path("/some-test-api-key/53.5511,9.9937")
                    .method("GET")
                .willRespondWith()
                    .status(200)
                    .body(FileLoader.read("classpath:weatherApiResponse.json"),
                            ContentType.APPLICATION_JSON)
                .toPact();
    }

    @Test
    @PactVerification("weather_provider")
    public void shouldFetchWeatherInformation() throws Exception {
        Optional<WeatherResponse> weatherResponse = weatherClient.fetchWeather();
        assertThat(weatherResponse.isPresent(), is(true));
        assertThat(weatherResponse.get().getSummary(), is("Rain"));
    }
}
```

If you look closely, you'll see that the `WeatherClientConsumerTest` is very similar to the `WeatherClientIntegrationTest`. Instead of using Wiremock for the server stub we use Pact this time. In fact the consumer test works exactly as the integration test, we replace the real third-party server with a stub, define the expected response and check that our client can parse the response correctly. In this sense the `WeatherClientConsumerTest` is a narrow integration test itself. The advantage over the wiremock-based test is that this test generates a *pact file* (found in `target/pacts/&pact-name>.json`) each time it runs. This pact file describes our expectations for the contract in a special JSON format. This pact file can then be used to verify that our stub server behaves like the real server. We can take the pact file and hand it to the team providing the interface. They take this pact file and write a provider test using the expectations defined in there. This way they test if their API fulfils all our expectations.

You see that this is where the *consumer-driven* part of CDC comes from. The consumer drives the implementation of the interface by describing their expectations. The provider has to make sure that they fulfil all expectations and they're done. No gold-plating, no YAGNI and stuff.

Getting the pact file to the providing team can happen in multiple ways. A simple one is to check them into version control and tell the provider team to always fetch the latest version of the pact file. A more advances one is to use an artifact repository, a service like Amazon's S3 or the pact broker. Start simple and grow as you need.

In your real-world application you don't need both, an *integration test* and a *consumer test* for a client class. The sample codebase contains both to show you how to use either one. If you want to write CDC tests using pact I recommend sticking to the latter. The effort of writing the tests is the same. Using pact has the benefit that you automatically get a pact file with the expectations to the contract that other teams can use to easily implement their provider tests. Of course this only makes sense if you can convince the other team to use pact as

well. If this doesn't work, using the *integration test* and Wiremock combination is a decent plan b.

Provider Test (the other team)

The provider test has to be implemented by the people providing the weather API. We're consuming a public API provided by darksky.net. In theory the darksky team would implement the provider test on their end to check that they're not breaking the contract between their application and our service.

Obviously they don't care about our meager sample application and won't implement a CDC test for us. That's the big difference between a public-facing API and an organisation adopting microservices. Public-facing APIs can't consider every single consumer out there or they'd become unable to move forward. Within your own organisation, you can — and should. Your app will most likely serve a handful, maybe a couple dozen of consumers max. You'll be fine writing provider tests for these interfaces in order to keep a stable system.

The providing team gets the pact file and runs it against their providing service. To do so they implement a provider test that reads the pact file, stubs out some test data and runs the expectations defined in the pact file against their service.

The pact folks have written several libraries for implementing provider tests. Their main GitHub repo gives you a nice overview which consumer and which provider libraries are available. Pick the one that best matches your tech stack.

For simplicity let's assume that the darksky API is implemented in Spring Boot as well. In this case they could use the Spring pact provider which hooks nicely into Spring's MockMVC mechanisms. A hypothetical provider test that the darksky.net team would implement could look like this:

```
@RunWith(RestPactRunner.class)
@Provider("weather_provider") // same as the "provider_name" in our clientConsumerTe
@PactFolder("target/pacts") // tells pact where to load the pact files from
```

```
public class WeatherProviderTest {
    @InjectMocks
    private ForecastController forecastController = new ForecastController();

    @Mock
    private ForecastService forecastService;

    @TestTarget
    public final MockMvcTarget target = new MockMvcTarget();

    @Before
    public void before() {
        initMocks(this);
        target.setControllers(forecastController);
    }

    @State("weather forecast data") // same as the "given()" in our clientConsumerTe
    public void weatherForecastData() {
        when(forecastService.fetchForecastFor(any(String.class), any(String.class)))
                .thenReturn(weatherForecast("Rain"));
    }
}
```

You see that all the provider test has to do is to load a pact file (e.g. by using the `@PactFolder` annotation to load previously downloaded pact files) and then define how test data for pre-defined states should be provided (e.g. using Mockito mocks). There's no custom test to be implemented. These are all derived from the pact file. It's important that the provider test has matching counterparts to the *provider name* and *state* declared in the consumer test.

Provider Test (our team)

We've seen how to test the contract between our service and the weather provider. With this interface our service acts as consumer, the weather service acts as provider. Thinking a little further we'll see that our service also acts as a

provider for others: We provide a REST API that offers a couple of endpoints ready to be consumed by others.

As we've just learned that contract tests are all the rage, we of course write a contract test for this contract as well. Luckily we're using consumer-driven contracts so there's all the consuming teams sending us their Pacts that we can use to implement our provider tests for our REST API.

Let's first add the Pact provider library for Spring to our project:

```
testCompile('au.com.dius:pact-jvm-provider-spring_2.12:3.5.5')
```

Implementing the provider test follows the same pattern as described before. For the sake of simplicity I simply checked the pact file from our simple consumer into our service's repository. This makes it easier for our purpose, in a real-life scenario you're probably going to use a more sophisticated mechanism to distribute your pact files.

```
@RunWith(RestPactRunner.class)
@Provider("person_provider")// same as in the "provider_name" part in our pact file
@PactFolder("target/pacts") // tells pact where to load the pact files from
public class ExampleProviderTest {

    @Mock
    private PersonRepository personRepository;

    @Mock
    private WeatherClient weatherClient;

    private ExampleController exampleController;

    @TestTarget
    public final MockMvcTarget target = new MockMvcTarget();

    @Before
    public void before() {
        initMocks(this);
        exampleController = new ExampleController(personRepository, weatherClient);
```

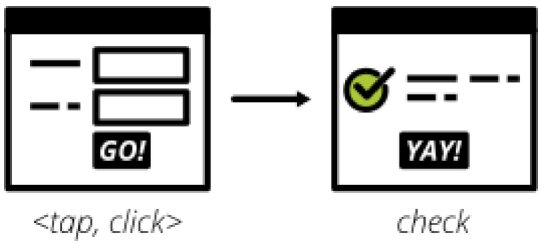
```
        target.setControllers(exampleController);
    }

    @State("person data") // same as the "given()" part in our consumer test
    public void personData() {
        Person peterPan = new Person("Peter", "Pan");
        when(personRepository.findByLastName("Pan")).thenReturn(Optional.of
                (peterPan));
    }
}
```

The shown `ExampleProviderTest` needs to provide state according to the pact file we're given, that's it. Once we run the provider test, Pact will pick up the pact file and fire HTTP request against our service that then responds according to the state we've set up.

UI Tests

Most applications have some sort of user interface. Typically we're talking about a web interface in the context of web applications. People often forget that a REST API or a command line interface is as much of a user interface as a fancy web user interface.

UI tests test that the user interface of your application works correctly. User input should trigger the right actions, data should be presented to the user, the UI state should change as expected.



UI Tests and end-to-end tests are sometimes (as in Mike Cohn's case) said to be the same thing. For me this conflates two things that are rather orthogonal concepts.

Yes, testing your application end-to-end often means driving your tests through the user interface. The inverse, however, is not true.

Testing your user interface doesn't have to be done in an end-to-end fashion. Depending on the technology you use, testing your user interface can be as simple as writing some unit tests for your frontend javascript code with your backend stubbed out.

With traditional web applications testing the user interface can be achieved with tools like Selenium. If you consider a REST API to be your user interface you should have everything you need by writing proper integration tests around your API.

With web interfaces there's multiple aspects that you probably want to test around your UI: behaviour, layout, usability or adherence to your corporate design are only a few.

Fortunately, testing the *behaviour* of your user interface is pretty simple. You click here, enter data there and want the state of the user interface to change accordingly. Modern single page application frameworks (react, vue.js, Angular and the like) often come with their own tools and helpers that allow you to thoroughly test these interactions in a pretty low-level (unit test) fashion. Even if you roll your own frontend implementation using vanilla javascript you can use

your regular testing tools like Jasmine or Mocha. With a more traditional, server-side rendered application, Selenium-based tests will be your best choice.

Testing that your web application's *layout* remains intact is a little harder. Depending on your application and your users' needs you may want to make sure that code changes don't break the website's layout by accident.

The problem is that computers are notoriously bad at checking if something "looks good" (maybe some clever machine learning algorithm can change that in the future).

There are some tools to try if you want to automatically check your web application's design in your build pipeline. Most of these tools utilise Selenium to open your web application in different browsers and formats, take screenshots and compare these to previously taken screenshots. If the old and new screenshots differ in an unexpected way, the tool will let you know.

Galen is one of these tools. But even rolling your own solution isn't too hard if you have special requirements. Some teams I've worked with built lineup and its Java-based cousin jlineup to achieve something similar. Both tools take the same Selenium-based approach I described before.

Once you want to test for *usability* and a "looks good" factor you leave the realms of automated testing. This is the area where you should rely on exploratory testing, usability testing (this can even be as simple as hallway testing) and showcases with your users to see if they like using your product and can use all features without getting frustrated or annoyed.

End-to-End Tests

Testing your deployed application via its user interface is the most end-to-end way you could test your application. The previously described, webdriver driven UI tests are a good example of end-to-end tests.

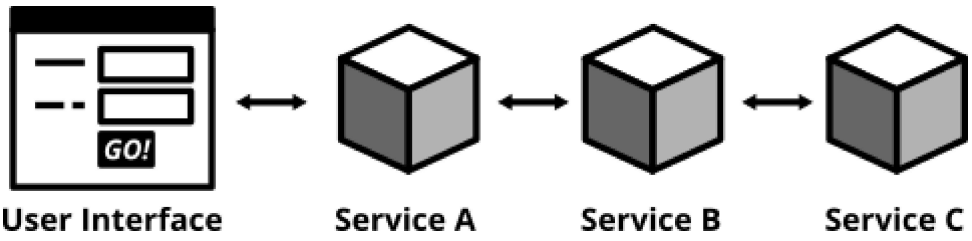


Figure 11: End-to-end tests test your entire, completely integrated system

End-to-end tests (also called Broad Stack Tests) give you the biggest confidence when you need to decide if your software is working or not. Selenium and the WebDriver Protocol allow you to automate your tests by automatically driving a (headless) browser against your deployed services, performing clicks, entering data and checking the state of your user interface. You can use Selenium directly or use tools that are build on top of it, Nightwatch being one of them.

End-to-End tests come with their own kind of problems. They are notoriously flaky and often fail for unexpected and unforeseeable reasons. Quite often their failure is a false positive. The more sophisticated your user interface, the more flaky the tests tend to become. Browser quirks, timing issues, animations and unexpected popup dialogs are only some of the reasons that got me spending more of my time with debugging than I'd like to admit.

In a microservices world there's also the big question of who's in charge of writing these tests. Since they span multiple services (your entire system) there's no single team responsible for writing end-to-end tests.

If you have a centralised *quality assurance* team they look like a good fit. Then again having a centralised QA team is a big anti-pattern and shouldn't have a place in a DevOps world where your teams are meant to be truly cross-functional. There's no easy answer who should own end-to-end tests. Maybe your organisation has a community of practice or a *quality guild* that can take care of these. Finding the correct answer highly depends on your organisation.

Furthermore, end-to-end tests require a lot of maintenance and run pretty slowly. Thinking about a landscape with more than a couple of microservices in place you won't even be able to run your end-to-end tests locally — as this would require to start all your microservices locally as well. Good luck spinning up hundreds of applications on your development machine without frying your RAM.

Due to their high maintenance cost you should aim to reduce the number of end-to-end tests to a bare minimum.

Think about the high-value interactions users will have with your application. Try to come up with user journeys that define the core value of your product and translate the most important steps of these user journeys into automated end-to-end tests.

If you're building an e-commerce site your most valuable customer journey could be a user searching for a product, putting it in the shopping basket and doing a checkout. That's it. As long as this journey still works you shouldn't be in too much trouble. Maybe you'll find one or two more crucial user journeys that you can translate into end-to-end tests. Everything more than that will likely be more painful than helpful.

Remember: you have lots of lower levels in your test pyramid where you already tested all sorts of edge cases and integrations with other parts of the system. There's no need to repeat these tests on a higher level. High maintenance effort and lots of false positives will slow you down and cause you to lose trust in your tests, sooner rather than later.

User Interface End-to-End Test

For end-to-end tests Selenium and the WebDriver protocol are the tool of choice for many developers. With Selenium you can pick a browser you like and let it

automatically call your website, click here and there, enter data and check that stuff changes in the user interface.

Selenium needs a browser that it can start and use for running its tests. There are multiple so-called '*drivers*' for different browsers that you could use. Pick one (or multiple) and add it to your `build.gradle`. Whatever browser you choose, you need to make sure that all devs in your team and your CI server have installed the correct version of the browser locally. This can be pretty painful to keep in sync. For Java, there's a nice little library called webdrivermanager that can automate downloading and setting up the correct version of the browser you want to use. Add these two dependencies to your `build.gradle` and you're good to go:

```
testCompile('org.seleniumhq.selenium:selenium-chrome-driver:2.53.1')
testCompile('io.github.bonigarcia:webdrivermanager:1.7.2')
```

Running a fully-fledged browser in your test suite can be a hassle. Especially when using continuous delivery the server running your pipeline might not be able to spin up a browser including a user interface (e.g. because there's no X-Server available). You can take a workaround for this problem by starting a virtual X-Server like xvfb.

A more recent approach is to use a *headless* browser (i.e. a browser that doesn't have a user interface) to run your webdriver tests. Until recently PhantomJS was the leading headless browser used for browser automation. Ever since both Chromium and Firefox announced that they've implemented a headless mode in their browsers PhantomJS all of a sudden became obsolete. After all it's better to test your website with a browser that your users actually use (like Firefox and Chrome) instead of using an artificial browser just because it's convenient for you as a developer.

Both, headless Firefox and Chrome, are brand new and yet to be widely adopted for implementing webdriver tests. We want to keep things simple. Instead of fiddling around to use the bleeding edge headless modes let's stick to the classic way using Selenium and a regular browser. A simple end-to-end test that fires up

Chrome, navigates to our service and checks the content of the website looks like this:

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class HelloE2ESeleniumTest {

    private WebDriver driver;

    @LocalServerPort
    private int port;

    @BeforeClass
    public static void setUpClass() throws Exception {
        ChromeDriverManager.getInstance().setup();
    }

    @Before
    public void setUp() throws Exception {
        driver = new ChromeDriver();
    }

    @After
    public void tearDown() {
        driver.close();
    }

    @Test
    public void helloPageHasTextHelloWorld() {
        driver.get(String.format("http://127.0.0.1:%s/hello", port));

        assertThat(driver.findElement(By.tagName("body")).getText(), containsString(
    }
}
```

Note that this test will only run on your system if you have Chrome installed on the system you run this test on (your local machine, your CI server).

The test is straightforward. It spins up the entire Spring application on a random port using `@SpringBootTest`. We then instantiate a new Chrome webdriver, tell it to go navigate to the `/hello` endpoint of our microservice and check that it prints "Hello World!" on the browser window. Cool stuff!

REST API End-to-End Test

Avoiding a graphical user interface when testing your application can be a good idea to come up with tests that are less flaky than full end-to-end tests while still covering a broad part of your application's stack. This can come in handy when testing through the web interface of your application is particularly hard. Maybe you don't even have a web UI but serve a REST API instead (because you have a single page application somewhere talking to that API, or simply because you despise everything that's nice and shiny). Either way, a Subcutaneous Test that tests just beneath the graphical user interface and can get you really far without compromising on confidence too much. Just the right thing if you're serving a REST API like we do in our example code:

```
@RestController
public class ExampleController {
    private final PersonRepository personRepository;

    // shortened for clarity

    @GetMapping("/hello/{lastName}")
    public String hello(@PathVariable final String lastName) {
        Optional<Person> foundPerson = personRepository.findByLastName(lastName);

        return foundPerson
             .map(person -> String.format("Hello %s %s!",
                    person.getFirstName(),
                    person.getLastName()))
             .orElse(String.format("Who is this '%s' you're talking about?",
                    lastName));
    }
}
```

Let me show you one more library that comes in handy when testing a service that provides a REST API. REST-assured is a library that gives you a nice DSL for firing real HTTP requests against an API and evaluating the responses you receive.

First things first: Add the dependency to your `build.gradle`.

```
testCompile('io.rest-assured:rest-assured:3.0.3')
```

With this library at our hands we can implement an end-to-end test for our REST API:

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class HelloE2ERestTest {

    @Autowired
    private PersonRepository personRepository;

    @LocalServerPort
    private int port;

    @After
    public void tearDown() throws Exception {
        personRepository.deleteAll();
    }

    @Test
    public void shouldReturnGreeting() throws Exception {
        Person peter = new Person("Peter", "Pan");
        personRepository.save(peter);

        when()
                .get(String.format("http://localhost:%s/hello/Pan", port))
        .then()
                .statusCode(is(200))
                .body(containsString("Hello Peter Pan!"));
```

```
    }
}
```

Again, we start the entire Spring application using `@SpringBootTest`. In this case we `@Autowire` the `PersonRepository` so that we can write test data into our database easily. When we now ask the REST API to say "hello" to our friend "Mr Pan" we're being presented with a nice greeting. Amazing! And more than enough of an end-to-end test if you don't even sport a web interface.

Acceptance Tests — Do Your Features Work Correctly?

The higher you move up in your test pyramid the more likely you enter the realms of testing whether the features you're building work correctly from a user's perspective. You can treat your application as a black box and shift the focus in your tests from

when I enter the values `x` and `y`, the return value should be `z`

towards

given there's a logged in user

and there's an article "bicycle"

when the user navigates to the "bicycle" article's detail page

and clicks the "add to basket" button

then the article "bicycle" should be in their shopping basket

Sometimes you'll hear the terms **functional test** or **acceptance test** for these kinds of tests. Sometimes people will tell you that functional and acceptance tests are different things. Sometimes the terms are conflated. Sometimes people will argue endlessly about wording and definitions. Often this discussion is a pretty big source of confusion.

Here's the thing: At one point you should make sure to test that your software works correctly from a *user's* perspective, not just from a technical perspective. What you call these tests is really not that important. Having these tests, however, is. Pick a term, stick to it, and write those tests.

This is also the moment where people talk about BDD and tools that allow you to implement tests in a BDD fashion. BDD or a BDD-style way of writing tests can be a nice trick to shift your mindset from implementation details towards the users' needs. Go ahead and give it a try.

You don't even need to adopt full-blown BDD tools like Cucumber (though you can). Some assertion libraries (like chai.js allow you to write assertions with `should`-style keywords that can make your tests read more BDD-like. And even if you don't use a library that provides this notation, clever and well-factored code will allow you to write user behaviour focused tests. Some helper methods/functions can get you a very long way:

```
# a sample acceptance test in Python

def test_add_to_basket():
    # given
    user = a_user_with_empty_basket()
    user.login()
    bicycle = article(name="bicycle", price=100)

    # when
    article_page.add_to_.basket(bicycle)
```

```
    # then
    assert user.basket.contains(bicycle)
```

Acceptance tests can come in different levels of granularity. Most of the time they will be rather high-level and test your service through the user interface. However, it's good to understand that there's technically no need to write acceptance tests at the highest level of your test pyramid. If your application design and your scenario at hand permits that you write an acceptance test at a lower level, go for it. Having a low-level test is better than having a high-level test. The concept of acceptance tests - proving that your features work correctly for the user - is completely orthogonal to your test pyramid.

Exploratory Testing

Even the most diligent test automation efforts are not perfect. Sometimes you miss certain edge cases in your automated tests. Sometimes it's nearly impossible to detect a particular bug by writing a unit test. Certain quality issues don't even become apparent within your automated tests (think about design or usability). Despite your best intentions with regards to test automation, manual testing of some sorts is still a good idea.

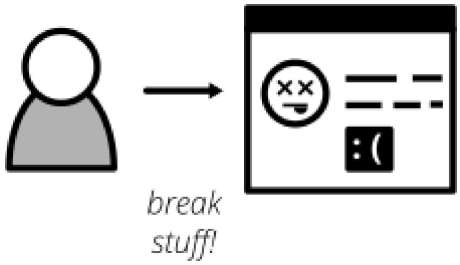


Figure 12: Use exploratory testing to spot all quality issues that your build pipeline didn't spot

Include Exploratory Testing in your testing portfolio. It is a manual testing approach that emphasises the tester's freedom and creativity to spot quality issues in a running system. Simply take some time on a regular schedule, roll up your sleeves and try to break your application. Use a destructive mindset and come up with ways to provoke issues and errors in your application. Document everything you find for later. Watch out for bugs, design issues, slow response times, missing or misleading error messages and everything else that would annoy you as a user of your software.

The good news is that you can happily automate most of your findings with automated tests. Writing automated tests for the bugs you spot makes sure there won't be any regressions of that bug in the future. Plus it helps you narrowing down the root cause of that issue during bugfixing.

During exploratory testing you will spot problems that slipped through your build pipeline unnoticed. Don't be frustrated. This is great feedback on the maturity of your build pipeline. As with any feedback, make sure to act on it: Think about what you can do to avoid these kinds of problems in the future. Maybe you're missing out on a certain set of automated tests. Maybe you have just been sloppy with your automated tests in this iteration and need to test more thoroughly in the future. Maybe there's a shiny new tool or approach that you could use in your pipeline to avoid these issues in the future. Make sure to act on it so your pipeline and your entire software delivery will grow more mature the longer you go.

The Confusion About Testing Terminology

Talking about different test classifications is always difficult. What I mean when I talk about *unit tests* can be slightly different from your understanding. With integration tests it's even worse. For some people integration testing is a very broad activity that tests through a lot of different parts of your entire system. For me it's a rather narrow thing, only testing the integration with one external part at a time. Some call them *integration tests*, some refer to them as *component tests*, some prefer the term *service test*. Even others will argue, that all of these three terms are totally different things. There's no right or wrong. The software development community simply hasn't managed to settle on well-defined terms around testing.

Don't get too hung up on sticking to ambiguous terms. It doesn't matter if you call it end-to-end or broad stack test or functional test. It doesn't matter if your integration tests mean something different to you than to the folks at another company. Yes, it would be really nice if our profession could settle on some well-defined terms and all stick to it. Unfortunately this hasn't happened yet. And since there are many nuances when it comes to writing tests it's really more of a spectrum than a bunch of discrete buckets anyways, which makes consistent naming even harder.

The important takeaway is that you should find terms that work for you and your team. Be clear about the different types of tests that you want to write. Agree on the naming in your team and find consensus on the scope of each type of test. If you get this consistent within your team (or maybe even within your organisation) that's really all you should care about. Simon Stewart summed this up very nicely when he described the approach they use at Google. And I think it shows perfectly how getting too hung up on names and naming conventions just isn't worth the hassle.

Putting Tests Into Your Deployment Pipeline

If you're using Continuous Integration or Continuous Delivery, you'll have a Deployment Pipeline in place that will run automated tests every time you make a change to your software. Usually this pipeline is split into several stages that gradually give you more confidence that your software is ready to be deployed to production. Hearing about all these different kinds of tests you're probably wondering how you should place them within your deployment pipeline. To answer this you should just think about one of the very foundational values of Continuous Delivery (indeed one of the core values of Extreme Programming and agile software development): **Fast Feedback**.

A good build pipeline tells you that you messed up as quick as possible. You don't want to wait an hour just to find out that your latest change broke some simple unit tests. Chances are that you've probably gone home already if your pipeline takes that long to give you that feedback. You could get this information within a matter of seconds, maybe a few minutes by putting the fast running tests in the earlier stages of your pipeline. Conversely you put the longer running tests - usually the ones with a broader scope - in the later stages to not defer the feedback from the fast-running tests. You see that defining the stages of your deployment pipeline is not driven by the types of tests but rather by their speed and scope. With that in mind it can be a very reasonable decision to put some of the really narrowly-scoped and fast-running integration tests in the same stage as your unit tests - simply because they give you faster feedback and not because you want to draw the line along the formal type of your tests.

Avoid Test Duplication

Now that you know that you should write different types of tests there's one more pitfall to avoid: duplicating tests throughout the different layers of the pyramid. While your gut feeling might say that there's no such thing as too many tests let me assure you, there is. Every single test in your test suite is additional baggage and doesn't come for free. Writing and maintaining tests takes time. Reading and understanding other people's test takes time. And of course, running tests takes time.

As with production code you should strive for simplicity and avoid duplication. In the context of implementing your test pyramid you should keep two rules of thumb in mind:

1. If a higher-level test spots an error and there's no lower-level test failing, you need to write a lower-level test
2. Push your tests as far down the test pyramid as you can

The first rule is important because lower-level tests allow you to better narrow down errors and replicate them in an isolated way. They'll run faster and will be less bloated when you're debugging the issue at hand. And they will serve as a good regression test for the future. The second rule is important to keep your test suite fast. If you have tested all conditions confidently on a lower-level test, there's no need to keep a higher-level test in your test suite. It just doesn't add more confidence that everything's working. Having redundant tests will become annoying in your daily work. Your test suite will be slower and you need to change more tests when you change the behaviour of your code.

Let's phrase this differently: If a higher-level test gives you more confidence that your application works correctly, you should have it. Writing a unit test for a `Controller` class helps to test the logic within the Controller itself. Still, this won't tell you whether the REST endpoint this Controller provides actually responds to HTTP requests. So you move up the test pyramid and add a test that checks for

exactly that - but nothing more. You don't test all the conditional logic and edge cases that your lower-level tests already cover in the higher-level test again. Make sure that the higher-level test focuses on the part that the lower-level tests couldn't cover.

I'm rigorous when it comes to eliminating tests that don't provide any value. I delete high-level tests that are already covered on a lower level (given they don't provide extra value). I replace higher-level tests with lower-level tests if possible. Sometimes that's hard, especially if you know that coming up with a test was hard work. Beware of the sunk cost fallacy and hit the delete key. There's no reason to waste more precious time on a test that ceased to provide value.

Writing Clean Test Code

As with writing code in general, coming up with good and clean test code takes great care. Here are some more hints for coming up with maintainable test code before you go ahead and hack away on your automated test suite:

1. Test code is as important as production code. Give it the same level of care and attention. "*this is only test code*" is not a valid excuse to justify sloppy code
2. Test one condition per test. This helps you to keep your tests short and easy to reason about
3. "*arrange*, *act*, *assert*" or "*given*, *when*, *then*" are good mnemonics to keep your tests well-structured
4. Readability matters. Don't try to be overly DRY . Duplication is okay, if it improves readability. Try to find a balance between DRY and DAMP code

5. When in doubt use the Rule of Three to decide when to refactor. *Use before reuse*

Conclusion

That's it! I know this was a long and tough read to explain why and how you should test your software. The great news is that this information is pretty timeless and independent of what kind of software you're building. It doesn't matter if you're working on a microservices landscape, IoT devices, mobile apps or web applications, the lessons from this article can be applied to all of these.

I hope that there's something useful in this article. Now go ahead and check out the sample code and get some of the concepts explained here into your testing portfolio. Having a solid test portfolio takes some effort. It will pay off in the longer term and it will make your live as a developer more peaceful, trust me.

Acknowledgements

Thanks to Clare Sudbery, Chris Ford, Martha Rohte, Andrew Jones-Weiss David Swallow, Aiko Klostermann, Bastian Stein, Sebastian Roidl and Birgitta Böckeler for providing feedback and suggestions to early drafts of this article. Thanks to Martin Fowler for his advice, insights and support.

▶ Significant Revisions