Testing Microservices

Ham Vocke

Version 1.0, 2017-09-28

Table of Contents

Learning the Foundations

Microservices have been all the rage for a while. If you attended any tech conference or read software engineering blogs lately, you'll either be amazed or fed up with all the stories that companies love to share about their microservices journey.

Behind all the hype are some true advantages to adopting a microservice architecture. And of course — as with every architecture decision — there will be trade-offs. I won't give you a lecture about the benefits and drawbacks of microservices or whether you should use them. Others have done a way better job at breaking this down than I could. Chance is, if you're reading this you somehow ended up with the decision to take a look into microservices and what it means to test them.

This book is made up of two parts. The first part explains high-level concepts and what type of tests you should have for your microservices. It also introduces you to a number of resources that you can use to learn more about the field.

You want to get more hands on (and you're not afraid of Java)? Take a look at the second part where we look at a sample microservice codebase and see how the concepts we learned in this post can be implemented.

This is a long post, go grab a cup of coffee and take your time. Maybe add it to your bookmarks and come back later. This post tries to be exhaustive, still it's just a starting point. Go ahead and dive deeper into the world of test automation using the linked resources. And most importantly: Start, experiment and learn what it means to test microservices on your own. You won't get it all correct from the beginning. That's ok. Start with best intentions, be diligent and explore!

Ready? Let's rock!

Too long; Didn't read

Here's what you'll take away from this book:

NOTE

TL;DR

1. Automate your tests (whoa, surprise!)
2. Continuous delivery makes your life easier
3. Remember the test pyramid *(don't be too confused by the original names of the layers, though)*
4. Write tests with different granularity
5. Use unit test to test the insides of your application
6. Use integration tests to test data serialization/deserialization
7. Test collaboration between services with contract tests (CDC)
8. Use end-to-end tests sparingly, limit to high-value user journeys
9. Don't just test from a developer's perspective, make sure to test features from a user's perspective as well
10. Exploratory testing will spot issues your build pipeline didn't catch

Microservices Need (Test) Automation

Microservices go hand in hand with **continuous delivery**, a practice where you automatically ensure that your software can be released into production any time. You use a **build pipeline** to automatically test and deploy your application to your testing and production environments.

Once you advance on your microservices quest you'll be juggling dozens, maybe even hundreds of microservices. At this point building, testing and deploying these services manually 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.

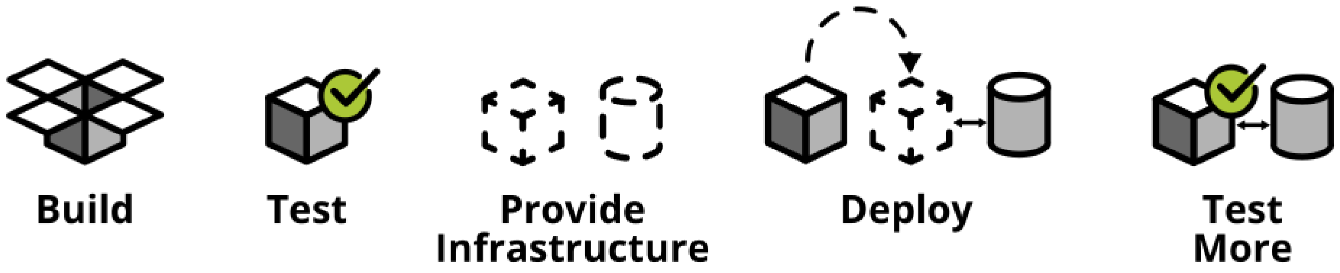

Figure 1. Use build pipelines to automatically and reliably get your software into production

Most microservices success stories are told by teams who use continuous delivery or **continuous deployment** (every software change that's proven to be releasable will be deployed to production). These teams make sure that changes get into the hands of their customers quickly.

How do you proof that your latest change still results in releasable software? You test your software including the latest change thoroughly. Traditionally you'd do this manually 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.

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 tests can be a big game changer in your life as a software developer. Automate your 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.

Automation in general and test automation specifically are essential to building a successful microservices architecture. Do yourself a favor and take a look at the concepts behind continuous delivery (the Continuous Delivery book is my go to resource). You will see that diligent automation allows you to deliver software faster and more reliable. Continuous delivery paves the way into a new world full of fast feedback and experimentation. At the very least it makes your life as a developer more peaceful.

The Test Pyramid

If you want to get serious about automated tests for your software there is one key concept that 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.

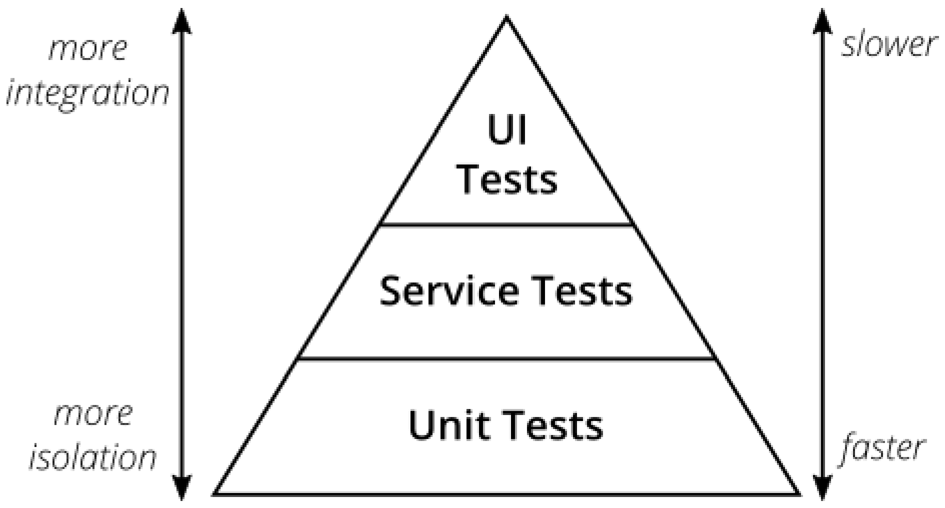

Figure 2. The test pyramid

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 optimal, and I have to agree. From a modern point of view the test pyramid seems overly simplistic and can therefore be a bit misleading.

Still, due to it's 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 modern 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.

Types of Tests

While the test pyramid suggests that you'll have three different types of tests (*unit tests*, *service tests* and *UI tests*) I need to disappoint you. Your reality will look a little more diverse. Lets keep Cohn's test pyramid in mind for its good things (use test layers with different granularity, make sure they're differently sized) and find out what types of tests we need for an effective test suite.

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. The number of unit tests in your test suite will largely outnumber any other type of test.

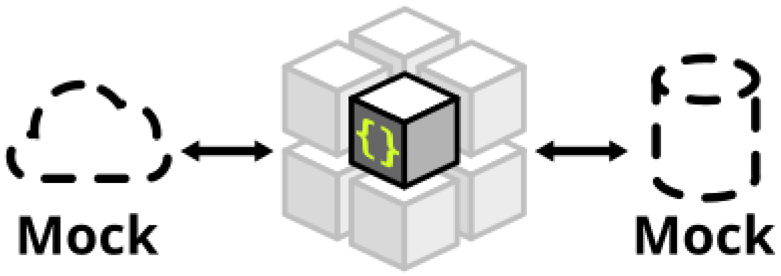

Figure 3. A unit test typically replaces external collaborators with mocks or stubs

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 extend 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 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

Mocking and **stubbing** (there's a difference if you want to be precise) should be heavily used instruments in your unit tests.

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.

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.

Unit Testing is Not Enough

A good unit test suite will be immensely helpful during development: You know that all the small units you tested are working correctly in isolation. Your small-scoped unit tests help you narrowing down and reproducing errors in your code. On top they give you fast feedback while working with the codebase and will tell you whether you broke something unintendedly. Consider them as a tool *for developers* as they are written from the developer's point of view and make their job easier.

Unfortunately writing unit alone won't get you very far. With unit tests you don't know whether your application as a whole works as intended. You don't know whether the features your customers love actually work. You don't know if you did a proper job plumbing and wiring all those components, classes and modules together.

Maybe there's something funky happening once all your small units join forces and work together as a bigger system. Maybe your code works perfectly fine when running against a mocked database but fails when it's supposed to write data to a real database. And maybe you wrote perfectly elegant and well-crafted code that totally fails to solve your users problem. Seems like we need more in order to spot these problems.

Integration Tests

All non-trivial applications will integrate with some other parts (databases, filesystems, network, and other services in your microservices landscape). When writing unit tests these are usually the parts you leave out in order to come up with better isolation and fast 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.

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

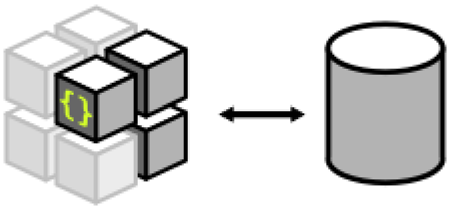

Figure 4. A database integration test integrates your code with a real database

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, an integration test for your REST API could look like this:

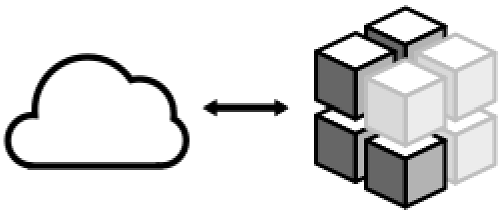

Figure 5. An HTTP integration test checks that real HTTP calls hit your code correctly

1. start your application
2. fire an HTTP request against one of your REST endpoints
3. check that the desired interaction has been triggered within your application

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. In a microservices architecture this happens more often than you might think. Think about:

- Calls to your services' REST API
- Reading from and writing to databases

- Calling other microservices
- 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.

If possible you should prefer to run your external dependencies locally: spin up a local MySQL database, test against a local ext4 filesystem. In some cases this won't be easy. If you're integrating with third-party systems from another vendor you might not have the option to run an instance of that service locally (though you should try; talk to your vendor and try to find a way).

If there's no way to run a third-party service locally you should opt for running a dedicated test instance somewhere 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).

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.

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.

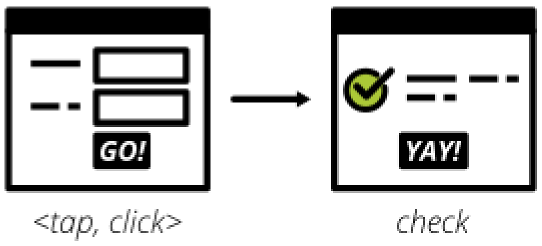

Figure 6. User Interface Tests

UI Tests and end-to-end tests are sometimes (as in Mark Cohn's case) said to be the same thing. For me this conflates two things that are not *necessarily* related.

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.

Fortunally, 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 thorougly 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 utilize 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.

Contract Tests

One of the big benefits of a microservice architecture is that it allows your organisation to scale their development efforts quite easily. You can spread the development of microservices across different teams and develop a big system consisting of multiple loosely coupled services without stepping on each others toes.

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 microservices can come in different shapes and technologies. Common ones are

- REST and JSON via HTTPS
- Remote Procedure Calls 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.

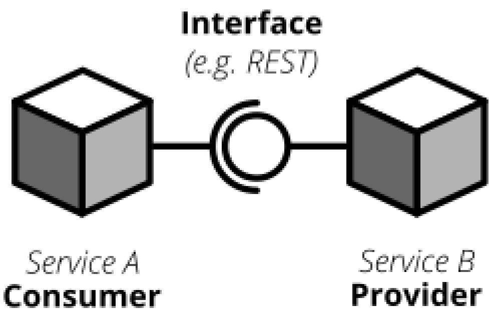

Figure 7. Each interface has a providing (or publishing) and a consuming (or subscribing) party. The specification of an interface can be considered a contract.

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:

1. Write a long and detailed interface specification (the *contract*)
2. Implement the providing service according to the defined contract
3. Throw the interface specification over the fence to the consuming team
4. Wait until they implement their part of consuming the interface
5. Run some large-scale manual system test to see if everything works
6. Hope that both teams stick to the interface definition forever and don't screw up

If you're not stuck in the dark ages of software development, you hopefully 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. All your microservices live 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.

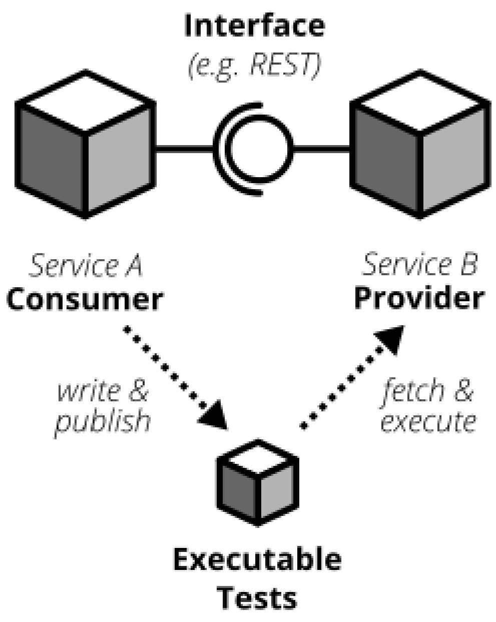

Figure 8. Contract tests ensure that the provider and all consumers of an interface stick to the defined interface contract. With CDC tests consumers of an interface publish their requirements in the form of automated tests; the providers fetch and execute these tests continuously

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:

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

If your organisation adopts microservices, 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 third-party 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. You can also find a hands-on example in the second part.

Consumer-Driven Contract tests can be a real game changer as you venture further on your microservices journey. 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.

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.

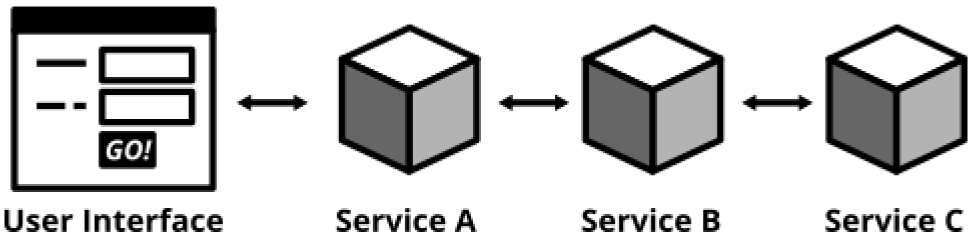

Figure 9. End-to-end tests test your entire, completely integrated system

End-to-end 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 unforseeable 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. Once you have 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 microservices 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 don’t and make sure you’ll lose trust in your tests rather sooner than later.

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 Behaviour-Driven Development (BDD) and tools that allow you to implement tests in a BDD fashion. BDD or a BDD-style way of wrtiting 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

```
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.

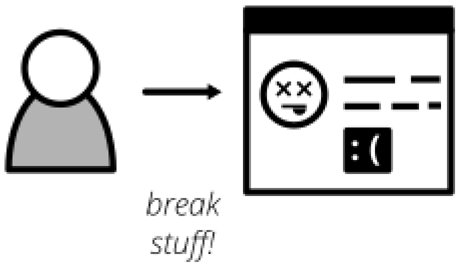

Figure 10. Use exploratory testing to spot all quality issues that your build pipeline didn't spot

Include Exploratory Testing in your testing portfolio. It is a manual testing approach that emphasizes 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.

Avoid Test Duplication

Now that you know that you should write different types of tests there's one more pitfall to avoid: test duplication. 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. If you managed to test all of your code's edge cases on a unit level there's no need to test these edge cases again on a higher-level. Keep this as a rule of thumb.

If your high-level test adds additional value (e.g. testing the integration with a real database) than it's a good idea to have this higher level test even though you might have tested the same database access function in a unit test. Just make sure to focus on the integration part in that test and avoid going through all possible edge-cases again.

Duplicating tests can be quite tempting, especially when you're new to test automation. Be aware of the additional cost and don't be afraid to delete tests if you were able to replace them with lower level tests or if they no longer provide any value.

Getting Hands on with Java & Spring Boot

The first part was a round-trip of what it means to test microservices. We looked at the test pyramid and found out that you should write different types of automated tests to come up with a reliable and effective test suite.

While the first part was more abstract this part will be more hands on and include code, lots of code. We will explore how we can implement the concepts discussed before. The technology of choice for this part will be **Java** with **Spring Boot** as the application framework. Most of the tools and libraries outlined here work for Java in general and don't require you to use Spring Boot at all. A few of them are test helpers specific to Spring Boot. Even if you don't use Spring Boot for your application there will be a lot to learn for you.

Tools and Libraries We’ll Look at

This part will demonstrate several tools and libraries that help us implement automated tests. The most important ones are:

JUnit
: as our test runner

Mockito
: for mocking dependencies

Wiremock
: for stubbing out third-party services

MockMVC
: for writing HTTP integration tests (this one’s Spring specific)

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. There are more tests than necessary for an application of this size. The tests on different levels overlap. This actively contradicts the advice that you should avoid test duplication throughout your test pyramid. Here I decided to go for duplication for demonstration purposes. Please keep in mind that this is not what you want for your real-world application. Duplicated tests are smelly and will be more annoying than helpful in the long term.

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 GithHub. 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:

1. `GET /hello`: Returns *"Hello World"*. Always.
2. `GET /hello/Vocke`: Looks up the person with the provided last name. If the person is known, returns *"Hello Ham Vocke"*.
3. `GET /weather`: Returns the current weather conditions for *Hamburg, Germany*.

High-level Structure

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

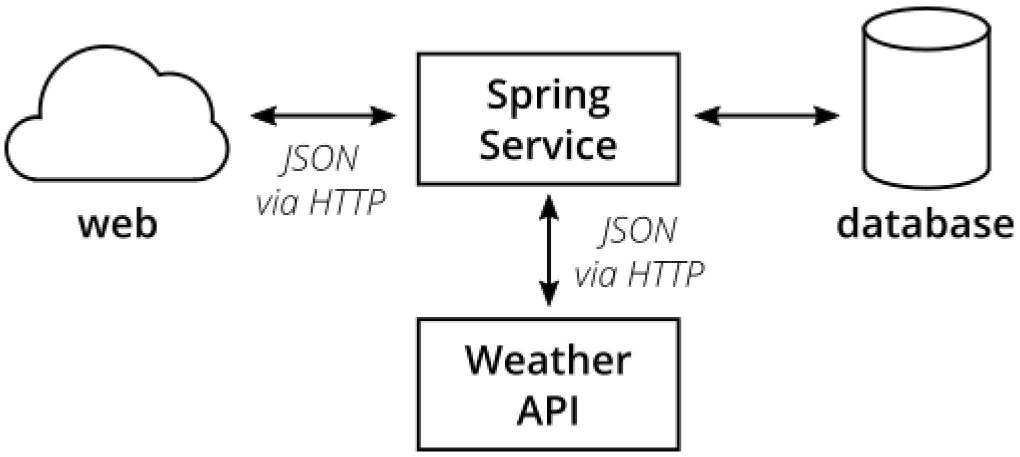

Figure 11. the high level structure of our microservice system

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:

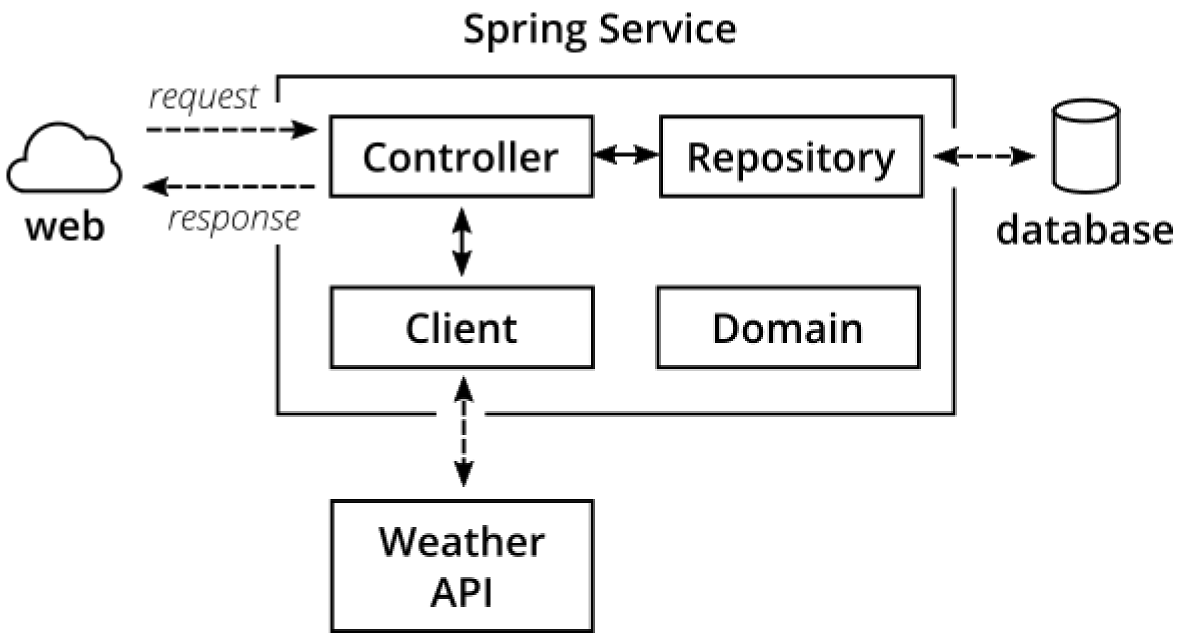

Figure 12. the internal structure of our microservice

- `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 https://en.wikipedia.org/wiki/Domain_model) including the domain logic (which, to be fair, is quite trivial in our case[domain model].

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 (Martin Fowler calls this an [Aenemic 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 utilize the power of object orientation.

Our repositories are straightforward and provide simple Create, Read, Update, Delete (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

Unit tests have the narrowest scope of all the tests in your test suite. Depending on the language you're using (and depending on who you ask) unit tests usually test single functions, methods or classes. Since we're working in Java, an object-oriented language, our unit tests will test methods in our Java classes. A good rule of thumb is to have one test class per class of production code.

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 behavior) 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 behavior 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 Test-Driven Development (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**. 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! Don't worry, Kent Beck said it's ok.

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.

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 <abbr title="Behavior-Driven Development">BDD</abbr>. 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.

An Example

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:

ExampleController.java

```
@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:

ExampleControllerTest.java

```
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

Integration tests are the next higher level in your test pyramid. They test that your application can successfully integrate with its sorroundings (databases, network, filesystems, etc.). 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 use it as load it in your integration test.

What to Test?

A good way to think about where you should have integration tests is to think about all places where data gets serialized or deserialized. Common ones are:

1. reading HTTP requests and sending HTTP responses through your REST API
2. reading and writing from/to a database
3. reading and writing from/to a filesystem
4. sending HTTP(S) requests to other services and parsing their responses

In the sample codebase you can find integration tests for `Repository`, `Controller` and `Client` classes. All these classes interface with the sorroundings of the application (databases or the network) and serialize and deserialize data. We can't test these integrations with unit tests.

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.

PersonRepository.java

```java
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 \`Person\`s 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 magic 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 magic 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:

PersonRepositoryIntegrationTest.java

```java
@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!

REST API Integration

Testing our microservice's REST API is quite simple. Of course we can write simple unit tests for all `Controller` classes and call the controller methods directly as a first measure. `Controller` classes should generally be quite straightforward and focus on request and response handling. Avoid putting business logic into controllers, that's none of their business (*best pun ever…*). This makes our unit tests straightforward (or even unnecessary, if it's too trivial).

As Controllers make heavy use of Spring MVC's annotations for defining endpoints, query parameters and so on we won't get very far with unit tests. We want to see if our API works as expected: Does it have the correct endpoints, interpret input parameters and answer with correct HTTP status codes and response bodies? To do so, we have to go beyond unit tests.

One way to test our API were to start up the entire Spring Boot service and fire real HTTP requests against our API. With this approach we were on the very top of our test pyramid. Luckily there's another, a little less end-to-end way.

Spring MVC comes with a nice testing utility we can use: With MockMVCwe can spin up a small slice of our spring application, use a <abbr title="Domain-Specific Language">DSL</abbr> to fire test requests at our API and check that the returned data is as expected.

Let's see how this works for the `/hello/<lastname>` endpoint `ExampleController`:

ExampleController.java

```
@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));
    }
}
```

Our controller calls the `PersonRepository` in the `/hello/<lastname>` endpoint. For our tests we need to replace this repository class with a mock to avoid hitting a real database. Even though this is an integration test, we're testing the REST API integration, not the database integration. That's why we stub the database in this case. The controller integration test looks as follows:

ExampleControllerIntegrationTest.java

```
@RunWith(SpringRunner.class)
@WebMvcTest(controllers = ExampleController.class)
public class ExampleControllerIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private PersonRepository personRepository;

    // shortened for clarity

    @Test
    public void shouldReturnFullName() throws Exception {
        Person peter = new Person("Peter", "Pan");
        given(personRepository.findByLastName("Pan")).willReturn(Optional.of(peter));

        mockMvc.perform(get("/hello/Pan"))
                .andExpect(content().string("Hello Peter Pan!"))
                .andExpect(status().is2xxSuccessful());
    }
}
```

I annotated the test class with `@WebMvcTest` to tell Spring which controller we're testing. This mechanism instructs Spring to only start the Rest API slice of our application. We won't hit any repositories so spinning them up and requiring a database to connect to would simply be wasteful.

Instead of relying on the real `PersonRepository` we replace it with a mock in our Spring context using the `@MockBean` annotation. This annotation replaces the annotated class with a Mockito mock globally, all classes that are `@Autowired` will only find the `@MockBean` in the Spring context and wire that one instead of a real one. In our test methods we can set the behaviour of these mocks exactly as we would in a unit test, it's a Mockito mock after all.

To use `MockMvc` we can simply `@Autowire` a MockMvc instance. In combination with the `@WebMvcTest` annotation this is all Spring needs to fire test requests against our controller and expect return values and HTTP status codes. The `MockMVC` DSL is quite powerful and gets you a long way. Fiddle around with it to see what else you can do.

Integration With Third-Party 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 is 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 (e.g. when you're coding on the airplane again instead of enjoying being crammed into a tiny airplane seat) or the darksky servers are down for some reason.

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:

WeatherClientIntegrationTest.java

```java
@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("Rain"));
        assertThat(weatherResponse, is(expectedResponse));
    }
}
```

To use Wiremock we instanciate 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 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 instanciating 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:

WeatherClient.java

```java
@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.

Parsing and Writing JSON

Writing a REST API these days you often pick JSON when it comes to sending your data over the wire. Using Spring there's no need to writing JSON by hand nor to write logic that transforms your objects into JSON (although you can do both if you feel like reinventing the wheel). Defining POJOs that represent the JSON structure you want to parse from a request or send with a response is enough.

Spring and Jackson take care of everything else. With the help of Jackson, Spring automagically parses JSON into Java objects and vice versa. If you have good reasons you can use any other JSON mapper out there in your codebase. The advantage of Jackson is that it comes bundled with Spring Boot.

Spring often hides the parsing and converting to JSON part from you as a developer. If you define a method in a `RestController` that returns a POJO, Spring MVC will automatically convert that POJO to a JSON string and put it in the response body. With Spring's `RestTemplate` you get the same magic. Sending a request using `RestTemplate` you can provide a POJO class that should be used to parse the response. Again it's Jackson being used under the hood.

When we talk to the weather API we receive a JSON response. The `WeatherResponse` class is a POJO representation of that JSON structure including all the fields we care about (which is only `response.currently.summary`). Using the `@JsonIgnoreProperties` annotation with the `ignoreUnknown` parameter set to `true` on our POJO objects gives us a tolerant reader, an interface that is liberal in what data it accepts (following [Postel's Law]. This way there can be all kinds of silly stuff in the JSON response we receive from the weather API. As long as `response.currently.summary` is there, we're happy.

If you want to test-drive your Jackson Mapping take a look at the `WeatherResponseTest`. This one tests the conversion of JSON into a `WeatherResponse` object. Since this deserialization is the only conversion we do in the application there's no need to test if a `WeatherResponse` can be converted to JSON correctly. Using the approach outlined below it's very simple to test serialization as well, though.

WeatherResponseTest.java

```
@Test
public void shouldDeserializeJson() throws Exception {
   String jsonResponse = FileLoader.read("classpath:weatherApiResponse.json");
   WeatherResponse expectedResponse = new WeatherResponse("Rain");

   WeatherResponse parsedResponse = new ObjectMapper().readValue(jsonResponse,
WeatherResponse.class);

   assertThat(parsedResponse, is(expectedResponse));
}
```

In this test case I read a sample JSON response from a file and let Jackson parse this JSON response using `ObjectMapper.readValue()`. Then I compare the result of the conversion with an expected `WeatherResponse` to see if the conversion works as expected.

You can argue that this kind of test is rather a unit than an integration test. Nevertheless, this kind of test can be pretty valuable to make sure that your JSON serialization and deserialization works as expected. Having these tests in place allows you to keep the integration tests around your REST API and your client classes smaller as you don't need to check the entire JSON conversion again.

CDC Tests

Consumer-Driven Contract (CDC) tests ensure that both parties involved in an interface between two services (the provider and the consumer) stick to the defined interface contract. This way contract tests ensure that the integration between two services remains intact.

Writing CDC tests can be as easy as sending HTTP requests to a deployed version of the service we're integrating against and verifying that the service answers with the expected data and status codes. Rolling your own CDC tests from scratch is straightforward but will soon send you down a rabbit hole. All of a sudden you need come up with a way to bundle our CDC tests, distribute them between teams and find a way to do versioning. While this is certainly possible, I want to demonstrate a different way.

In this example I'm using Pact to implement the consumer and provider side of our CDC tests.

Pact is available for multiple languages and can therefore also be used in a polyglot context. Using Pact we only need to exchange JSON files between consumers and providers. One of the more advanced features even gives us a so called "pact broker" that we can use to exchange pacts between teams and show which services integrate with each other.

Contract tests always include both sides of an interface — the consumer and the provider. Both parties need to write and run automated tests to ensure that their changes don't break the interface contract. Let's see what either side has to do when using Pact.

Consumer Test (our end)

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:

WeatherClientConsumerTest.java

```java
@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
IOException {
        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. The difference is that the consumer 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.

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 fulfill all expectations and they're done. No gold-plating, no YAGNI and stuff.

We can take the pact file and hand it to the team providing the interface. They in turn can take this pact file and write a provider test using the expectations defined in there. This way they test if their API fulfills all our expectations.

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:

WeatherProviderTest.java

```
@RunWith(RestPactRunner.class)
@Provider("weather_provider") // same as in the "provider_name" part in our
clientConsumerTest
@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()" part in our
clientConsumerTest
    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.

I know that this whole CDC thing can be confusing as hell when you get started. Believe me when I say it's worth taking your time to understand it. If you need a more thorough example, go and check out the fantastic example my friend Lukasz has written. This repo demonstrates how to write consumer and provider tests using pact. It even features both Java and JavaScript services so that you can see how easy it is to use this approach with different programming languages.

End-to-End Tests

At last we arrived at top of our test pyramid (phew, almost there!). Time to write end-to-end tests that calls our service via the user interface and does a round-trip through the complete system.

Using Selenium (testing via the UI)

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`:

```
testCompile('org.seleniumhq.selenium:selenium-firefox-driver:3.5.3')
```

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

HelloE2ESeleniumTest.java

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

    private WebDriver driver = new FirefoxDriver();

    @LocalServerPort
    private int port;

    @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("Hello World!"));
    }
}
```

Note that this test will only run on your system if you have Firefox 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 instanciate a new Firefox 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!

Using RestAssured (Testing via the REST API)

I know, we already have tests in place that fire some sort of request against our REST API and check that the results are correct. Still, none of them is truly end to end. The MockMVC tests are "only" integration tests and don't send real HTTP requests against a fully running service.

Let me show you one last tool that can come in handy when you write 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 checks the responses. It looks similar to MockMVC but is truly end-to-end (fun fact: there's even a REST-Assured MockMVC dialect). If you think Selenium is overkill for your application as you don't really have a user interface that needs testing, REST-Assured is the way to go.

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 a end-to-end test for our REST API:

HelloE2ERestTest.java

```
@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.

Some Advice Before You Leave

There we go, you made it through the entire testing pyramid. Congratulations! Before you go, there are some more general pieces of advice that I think will be helpful on your journey. Keep these in mind and you'll soon write automated tests that truly kick ass:

1. Test code is as important as production code. Give it the same level of care and attention. Never allow sloppy code to be justified with the *"this is only test code"* claim
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 (*Don't Repeat Yourself*). 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*.

Now it's your turn. Go ahead and make sure your microservices are properly tested. Your life will be more relaxed and your features will be written in almost no time. Promise!

Further reading

Building Microservices by Sam Newman

This book contains so much more there is to know about building microservices. A lot of the ideas in this article can be found in this book as well. The chapter about testing is available as a free sample over at O'Reilly.

Continuous Delivery by Jez Humble and Dave Farley

The canonical book on continuous delivery. Contains a lot of useful information about build pipelines, test and deployment automation and the cultural mindset around CD. This book has been a real eye opener in my career.

Working Effectively with Unit Tests by Jay Fields

If you level up your unit testing skills or read more about mocking, stubbing, sociable and solitary unit tests, this is your resource.

Testing Microservices by Toby Clemson

A fantastic slide deck with a lot of useful information about the different considerations when testing a microservice. Has lots of nice diagrams to show what boundaries you should be looking at.

Growing Object-Oriented Software Guided by Tests by Steve Freeman and Nat Pryce

If you're still trying to get your head around this whole testing thing (and ideally are working with Java) this is the single book you should be reading right now.

Test-Driven Development: By example by Kent Beck

The classic TDD book by Kent Beck. Demonstrates on a hands-on walkthrough how you TDD your way to working software.