Testing

At some point during every project, there will inevitably be two questions that need answering for each change you make: does it work, and has it broken anything? Testing can be used to answer these two questions, although there are subtle differences between the types of testing used to resolve them.

Test-Driven Development

Test-driven development (TDD) is a well-established technique for developing software, although it is often misunderstood. The purpose of TDD is not to generate high test coverage, but to use tests to help drive the design of your code; TDD is really about development, not just tests.

TDD is based on a cycle of "red-green-refactor." At first, you write a test for the new functionality you want to add, which should fail (go red), and then you write the simplest bit of functionality to make the test pass (go green). Once you have done this, you can refactor your application and test code to tidy it up and remove duplication.

In test-driven development, it is common to write tests that only address a single module at a time. These types of tests are known as unit tests, as opposed to integration tests, which are discussed further later in the chapter, and which test multiple modules at once. Some people, especially those who practice behavior-driven development, will often start by writing a test that's wider than one particular module, and then write the necessary tests with a smaller scope that satisfies the requirements of the bigger test—this is the outside-in approach. Others prefer to start at the smallest unit and then write integration tests to wire the modules together once they have been built, referred to as the inside-out approach. It is often a matter of personal style as to which approach you go for, although proponents of the outside-in approach say it helps them decide which components need to be written, and proponents of the inside-out approach like the flexibility of not having to design the interfaces of the individual modules in advance.

C. Northwood, *The Full Stack Developer*, https://doi.org/10.1007/978-1-4842-4152-3_7

If you have issues writing tests that only target one module, that might suggest that there are several highly coupled modules that might actually be better joined as one, or, if there are multiple distinct types of tests for that one module, that your one module should in fact be split into multiple smaller ones.

"Arrange-act-assert" (sometimes called "given-when-then," especially when used with behavior-driven development, discussed later in this chapter) is a common pattern for arranging your test code. Take the example below:

```
it('should increase the size of the shopping cart when adding a new
item to it', () => {
        const cart = new  ShoppingCart();
        const product = fetchTestProduct();

        cart.add(product);

        expect(cart.size()).toEqual(1);
});
```

Here, the arrange-act-assert process is shown using white space, which is a common pattern that can help readability when scanning a test file. The "arrange" part of the test involves setting up the objects and bits of data you want to test into the right state, and then "acting" upon that by running the function that's being tested. Finally, "assert" checks that the act step was successful.

There may be times when this style does not work for the type of test you want to write, but be careful—a test that follows the arrange-act-assert-act-assert pattern should often be broken down into two separate tests, perhaps abstracting the "arrange" steps out into a helper function that is used in several tests.

Each test you write should test one logical bit of functionality: the part of the code you're testing. This might mean there are multiple assertion lines in a test, but there should only be one *concept* that is being asserted. The value in this is that if a test begins to fail, you should know from the name exactly why it is failing. If there are different concepts being checked in one test, some failures can mask others, so you might not get a full picture of why a change has introduced a regression until later on.

In the process of running the red-green-refactor cycle, you'll end up with a comprehensive automated test suite. With this kind of suite, you can go a long way toward answering the second question: has this change broken anything? A passing test suite should give you a high degree of confidence. One could argue that any good test

suite should give you this degree of confidence, but a test suite developed using TDD allows you to build in this confidence from the start; seeing a test go from red to green lets you know the tests are working.

Not having any automated tests at all is a very dangerous practice. Skipping testing completely is often too high a risk for any organization to accept, but without automated tests, manual QA is needed, and it is too slow and expensive for the fast-changing nature of a digital organization. Test automation, especially for rote checks, is therefore an essential part of modern product delivery.

ALTERNATIVES TO UNIT TESTING

The functional programming language Haskell uses an alternative approach to unit testing than the arrange-act-assert method described above. Instead of writing code with explicit examples for the input and expected output of a particular function, you instead write assertions that should be true for whole ranges of inputs. The testing library (in Haskell's case, this is called QuickCheck) then randomly generates input from the allowed ranges to try and find cases that fail.

For example, when testing a sorting algorithm, you may want to specify that, when the input is an array of integers, each item in the array is less than or equal to the next element.

This is known as "property-based testing" and is especially useful when developing more algorithmic aspects of code (such as sorting algorithms), where there are a large number of edge cases in the input data.

Test Pyramid

Test-driven development is an effective way of developing a test suite that can help protect you from accidental breakage, but there are many different ways a system can break which unit tests developed using TDD will not cover. Although an individual class may be behaving correctly according to the tests, there's no guarantee that any other class that relies on this one is making the correct assumptions about how that class behaves, or if any changes have caused a responsibility to be lost. Because of this, it's important to test at various levels within your application, from the smallest units (which could be an individual function or class) to the individual modules, and even the whole system.

Ultimately, all the organization cares about is whether or not the system as a whole works, and it can be very tempting to write the majority of your tests at this level. However, there is a downside to testing the system as a whole. It is often much slower than running an individual unit in isolation, and when external dependencies are involved, it can be brittle due to circumstances outside the direct control of the system.

The test pyramid shown in Figure 7-1 highlights the different layers to test. When your tests are considered at all levels of the pyramid, you should have a high degree of confidence in the performance of your application.

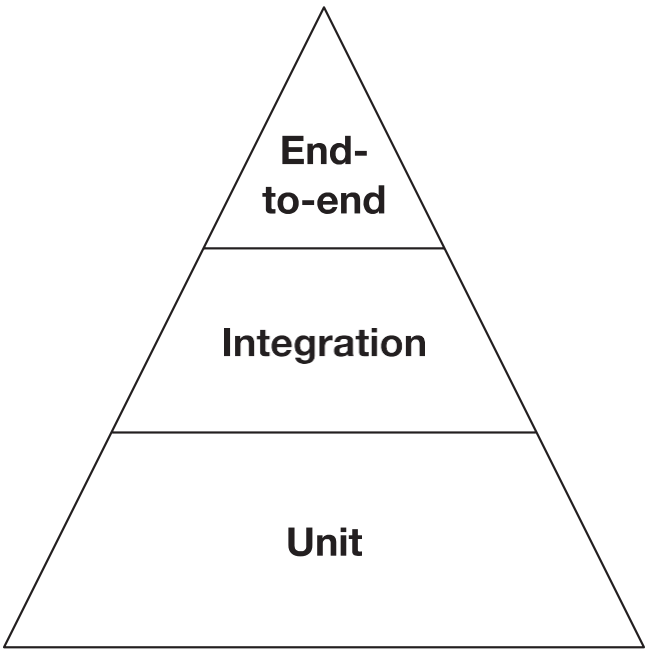

Figure 7-1. *The testing pyramid, with height showing level of abstraction and breadth showing coverage at each level*

At the bottom of the pyramid are unit tests. These test a single unit in isolation but test each system exhaustively. When it comes to unit tests, it's important to test only an individual system in isolation. However, many classes will have dependencies on others, and in this case, a technique known as mocking is used. Many testing libraries have built-in support for mocks, which are objects that have the same interface as the dependency. Mocks can be used to ensure they are interacted with in the correct way, or return canned responses to easily but exhaustively test that a class behaves correctly under all conditions.

Dependency injection is a technique where a class (or function) is given its dependencies, as opposed to instantiating them itself. When using dependency injection, when the system is being tested, instead of real instances of the dependencies being given to a class (for example, in its constructor), the mocks are given instead.

However, there are times when not all the dependencies can be mocked out—for example, a class that has to read a file from disk, or is responsible for actually making a database connection. It is important to make these classes as small as possible, so they are little more than wrappers around the external dependency. Then, for classes that need that functionality, mocks can be used instead. When it comes to testing these units, the integration testing level can be used.

Integration testing is used to ensure that a cluster of modules actually interact together in the expected way. When using mocks, it can be easy to introduce an assumption to a mock that does not reflect the way the real dependency operates. This is especially true in languages that do not have strong typing, such as JavaScript, Ruby, or Python. Integration testing allows a test to run against real instances of their dependencies, such as real files on disk, a real database, or other dependencies. However, integration testing does not need to be as exhaustive as unit testing. Rather than testing each possible configuration of inputs and outputs, the value from integration testing comes from ensuring the assumptions on the interactions that have been mocked are correct, so it's only tests for these interactions that must be tested.

Integration tests are not as quick as unit tests, because they may have to do more computations rather than return a pre-canned answer, or have to communicate with a real database. Regardless, they are still fairly fast. Individual unit tests can be expected to take less than a few milliseconds to complete, whereas integration tests can take tens or hundreds of milliseconds. The slowest type of test is the end-to-end test, where each test can potentially take several seconds to complete.

End-to-end tests offer significant value, and because they are equivalent to the tests a human can run, it can be tempting to run a large number of them. This is a common pitfall for teams with someone in a traditional QA role who starts to embrace test automation.

End-to-end tests interact with systems in the same way a user would. For an API, this could be by simply making HTTP requests from a separate program, but for a web site, this can involve using either a headless browser (a type of browser that can be controlled programmatically, but never needs to show a visual representation of the web page to a human). However, headless browsers are not truly representative of how a user interacts

with a site, especially as there is no browser with a perfect implementation of HTML standards. A popular tool called Selenium allows for real web browsers to be controlled remotely, and to query the state of a web page to make assertions on the state. It is also common to run the same test suite against different browsers to ensure cross-browser compatibility.

As these tests interact like a real user, they give the highest degree of confidence. But they are also the slowest. You should write as few tests as possible at this level, but enough to cover the most important parts of the system, similar to the approach taken with integration versus unit tests. For example, if you have an HTML form, it is important to test the success and failure cases at the end-to-end level, but testing every single validation, and every way in which a validation can fail, is best performed at a lower level of the pyramid, such as the unit test level.

Integration and unit tests interact with classes directly and must be written in the same language, and interact directly with the code of the system under test. The same does not apply to end-to-end tests. It is not uncommon to see end-to-end tests written in a different language from the application (for example, using Ruby with Capybara to drive a browser), and end-to-end tests can grow in complexity considerably. A technique known as the Page Object Model allows you to organize your code, which can simplify the actual tests. In this process, a layer of abstraction is placed between the browser and the tests, in the form of classes that abstract interactions between the pages. This technique is very helpful in clarifying exactly what the purpose of the test is, as well as reducing any duplication of identifiers such as CSS selectors, which can make the end-to-end tests quite heavily coupled to implementation details of the page being tested.

Using the Page Object Model, you create an abstraction layer over your UI to hide any implementation details, and then create a test using that abstraction layer. This allows you to change UI implementation details without breaking a large number of tests at once, as you can refactor the abstraction layer at the same time, as well as adding clarity to your test code. The example below shows how you may use Page Object Model alongside Capybara in Ruby.

```
class HomePage
  include Capybara::DSL

  def visit!
    visit '/taster/' if category.nil?
  end
```

```
  def pilot_filter
    PilotFilter.new
  end
end

class PilotFilter
  include Capybara::DSL

  def visible?
    not container.nil?
  end

  def opened?
    container.find('.taster-filter_list', visible: :all).visible?
  end

  def open
    page.execute_script("$('.taster-filter').focusin()")
  end

  private

  def container
    first('.taster-filter')
  end
end

describe 'Pilot Filter' do
  before do
    @home_page = HomePage.new
    @home_page.visit!
  end

  it 'should appear on the home page', smoke: true do
    expect(@home_page.pilot_filter).to be_visible
  end

  it 'starts off closed' do
    expect(@home_page.pilot_filter).not_to be_opened
  end
```

```
  it 'opens when selected' do
    @home_page.pilot_filter.open

    wait_for(@home_page.pilot_filter).to be_opened
  end
end
```

Behavior-Driven Development

When it comes to answering the first question—whether or not a change works—automated testing introduced by developers often isn't quite enough. The main issue is that it's the developers who are defining both what the change is and whether or not it works. This is fine if the developers and the organization are in total agreement on what a successful change looks like, and what the desired change even is, but this is, unsurprisingly, rare. As discussed earlier, a lot of desired change at an organizational level is actually fairly abstract, but the product and UX functions of the team should have a fairly good idea of what concrete actions are needed in order to implement those changes, and it's those concrete actions that fall to the development team.

In the world of waterfall projects, requirements are often specified in large Word documents known as functional requirements documents, with requirement identifiers. These documents are handed over to the development team, which does what they think they're being asked to do, and then to a separate test team, which creates test cases and scenarios that satisfy each requirement. The main issue with this approach is the speed, checking that the developer and tester understanding of the requirements align comes at the end of the project. Most agile frameworks try to introduce the idea of a common understanding at the start of the build of a feature, and behavior-driven development (BDD) is a specific technique that can help with this.

Like TDD, BDD is often misunderstood, largely due to a growing number of BDD tools. Using a BDD tool like Cucumber, without fully understanding the rest of the BDD process, can become a more complex way of doing unit testing without giving you any additional benefit. Many BDD tools are based on the idea of executable specifications—that specification written by non-coders is responsible for executing your tests. These executable specifications are then parsed and matched against specific rules, which leads to real code being executed.

There are a lot of problems with this approach. Natural language is a bad proxy for code, so you either end up with a lot of ambiguous prose that is hard to parse, or, in order to make writing the parsers easier, you constrain the language of the specifications, which can frustrate business representatives. Executable specifications were designed to help business representatives and the development team communicate, and to ensure that the tests do actually cover all the requirements of the organization. There are other ways of achieving this same goal, though.

Waterfall projects often use requirement identifiers and the concept of traceability to ensure coverage—that is, you should be able to see which requirement caused some tests to come about, and that there are sufficient tests to cover a requirement. This can then be audited to ensure coverage. By stripping back this process to the essentials, we can re-use this in an agile project and get the best of both worlds. By giving each acceptance criteria an ID, and then putting a code comment by each automated test that implements the checks for the criteria specified in that ID, we can maintain traceability. These IDs can be trivial to generate. I often use a numbered list on a story card (either the back of a physical card or on a system like Jira), and then an ID can become simply the number in the bullet point plus the ticket number—e.g., #24/6 for the 6th bullet point on card 24. Instead of a full audit, a simple code review can then be used to double check the coverage.

Separating the specifications from the test code while maintaining traceability will make it easier to write the tests at the appropriate level of the test pyramid, which can help avoid many problems experienced by teams that are attempting to introduce executable specifications.

Three Amigos

So, how can we generate those acceptance criteria in the first place in order to gain a shared understanding? Getting this right is actually the very essence of BDD, and most people who focus purely on the tools may see their attempt to use BDD as a failure as they have failed to understand this point. The best way to achieve this is through shared authorship between all the parties involved. A technique known as the “Three Amigos” is a very effective way of doing this. The three amigos are a product representative, a developer, and a tester, the idea being that the product representative is the one who knows and understands the problem, the developer can ask questions in order to elicit a shared understanding, and the tester can guide the amigos to explore the whole problem space and interactions with other parts of the system to ensure all parts are covered.

In reality, the roles are a bit more blurred, and there can be more than three people involved; other business specialists, UX specialists, and project managers are commonly included. It's also okay for the specification to evolve as questions are asked, and it's completely acceptable for a developer to propose alternate solutions that may be simpler to implement as long as the essence of the requirements is still met.

The most important part of a three amigos meeting is that, at minimum, the people who will build the solution and the business people who know what is needed, have those conversations together, and are empowered to make and act upon decisions in these meetings. An amigos session with just a tester and a developer will end up simply guessing at the requirements. Similarly, one where the amigos are not empowered to make decisions will end up playing committee tennis, where the proposals go to another board to be reviewed and signed off, possibly with alterations added.

Often the outcome of an amigos session is the executable specification, but a set of acceptance criteria can be just as effective, especially if it's easier to use the expressiveness of natural language to say what the expectations are. These acceptance criteria are not the only point of reference, though. The developers and testers working on a story may not be the same as in the amigos session, but communication can help, especially because the avoidance of ambiguity in natural language is hard! It can be helpful to build a glossary on a wiki page over time, so stakeholders understand what a specific term means. This can be especially useful if there are fairly generic names in use (for example, an app may have an "about" page for the app in general, and an "info" page for each specific item).

The conversation is just as important as the outcome. Therefore, it's often unhelpful to develop the acceptance criteria for a story much in advance of when the build starts, partly because the business needs may have changed, and partly because people forget the discussions. Nailing those discussions down in the form of code is the ultimate goal.

One popular technique for specifying acceptance criteria is to use the Given-When-Then form of specification. This form starts by setting up the preconditions or assumptions for a particular part of the story, then specifies what actions the user takes, and then the post-conditions, or what should happen when a user performs those actions.

- Given I have an empty shopping cart
- When I add an in-stock product to my shopping cart
- Then the shopping cart should show that it is not empty

If you are taking this approach, it can be easy to fall into a trap of over-specification (this is especially true when these Given-When-Thens are being used as the basis for an executable specification). The above scenario can be considered a fairly high-level specification that still captures the essence of what is needed. When combined with a set of wireframes from a designer, it will often show exactly what is needed. The following example shows the same scenario but is over-specified.

- Given I have an empty shopping cart
- And there is an in-stock product in the database
- When I visit the page of the in-stock product
- Then I should see an "Add to Cart" button
- When I click the "Add to Cart" button
- Then the shopping cart icon should change to show a full cart
- And the number '1' should appear on the shopping cart icon

The issue with over-specifying like this is that it makes it hard to see the essence of the scenario, which can disengage business stakeholders.

An alternate style for Given-When-Thens is to use concrete examples. I personally prefer the more abstract ones, but it is just a matter of preference. The above scenario could be expressed like this:

- Given I have an empty shopping cart
- And carrots are in stock
- When I add carrots to my shopping cart
- Then the shopping cart should show that it is not empty

For teams that are considering adopting BDD, it's important to start with the essence of it and then choose tools based on their needs. You can do BDD without using Cucumber, and you can do BDD without using the Given-When-Then format for your acceptance criteria. Experiment and find what works well for your team!

Manual Testing

Regardless of how much you automate, there will always be a need for some form of manual QA. Automation removes the rote of simple testing (does clicking on this button do the right thing?), allowing testers to focus on more complex and emergent behavior. This can be a quick sanity check on a new feature, or an attempt to reproduce some unexpected behavior a user reported, or an aid to developing automated tests (if an automated test is failing, does a manual test pass, or when developing new automated tests, knowing the automated steps can come from first testing manually).

Most complexity in a system comes from interaction between components. When a whole system is tested together, those components can interact in unexpected ways, giving rise to unexpected behaviors. These behaviors often need to be kept in check, and identifying them is what manual testing excels at.

The two major parts of manual testing are functional testing and regression testing. Functional testing checks that any changes that have been made behave as expected, and make sense within the context of the whole; regression testing checks that any pre-existing functionality has not been broken by the new feature. Regression testing is the area where automation can have the most impact, as this is the most repetitive type of testing. Automation by itself is often not sufficient, and this is where close collaboration between a developer and a tester can help. A developer will be aware of which code paths they may have touched in implementing a feature (which may sometimes be non-obvious), which allows for testing to focus on the areas that are most likely to have been changed.

It is common for testers to write a test plan that lays out what they are going to test and their approach to doing so, and historically this document can be quite large. A particular test case might break down the steps to follow and the expected response, but at that level it becomes somewhat similar to writing code, so this level of detail is now often expressed in code. Especially when using behavior-driven development, the test plan can simply reference those acceptance criteria and focus on high-level approaches, if a test plan even exists at all.

It used to be common for dedicated test teams to sit apart from development teams, who accepted builds at the end of the sprint and then spent a test sprint testing the previous release—this led to incredibly slow feedback cycles. A more modern approach is to sit testers down with a development team and work alongside the developers, testing features incrementally as each component is developed. Bugs are raised in the

same sprint and can be fixed while the work is fresh in the developers' minds. A more dysfunctional model could involve completely outsourcing a function like regression testing to an offshore facility where the testers simply follow test scripts expressed in a test plan. This kind of testing is ripe for automation.

Test automation might seem to eliminate the need for a tester, but the responsibilities and mindset of a tester still play a valuable role on modern teams. To that end, it's quite common for testers to refer to themselves as QAs, or quality analysts, who take a role more akin to a business analyst than a traditional tester. Where the difference of this role is best exemplified is in a three amigos-type situation, where the critical eye and perspective of a tester can help flesh out additional acceptance criteria. Another area where a QA can help is "exploratory" testing. In exploratory testing, the QA takes on the mindset of a user, and does complete end-to-end tests of various parts of functionality, ensuring that everything happens as expected (not just as specified). The QA has to understand the product and its customers to be able to do this effectively, and although testing the "happy path" (the most common use case that a user is expected to follow) is useful, a QA should also diverge from these paths where appropriate and do things that may seem strange, but are perhaps something a confused user would do. This kind of exploratory testing, especially when combined with user testing in the UX process, is where testing adds significant value. It is not just bugs that can be found, but also deficient specifications.

Visual Testing

Another area where manual testing excels is "eyeballing" a page to ensure it is visually effective. There have been many attempts to automate this process, with various levels of success, and there is no general solution that satisfies the question, "does this web page look like it's supposed to?".

Selenium is useful for testing whether or not content appears on a page and is visible, and it is possible to write tests that make assertions regarding the position (in terms of x and y coordinates) of a particular element on a page, but these tests will be very brittle. Small differences between browser versions, or available fonts, can introduce small errors, and changes such as introducing a new component on a page may cause a large number of tests to be updated. Unreliable tests will often be ignored, and combined with the amount of work required to maintain those tests, it usually means it is not worth writing them.

The human eye is therefore the best tool to verify the look and feel of a web page, but as a web application grows, running regression tests can get tiresome when all are done manually. Fortunately, there are tools that can help here, although they often have a high maintenance cost and are not used often. The general idea is to take either entire pages or individual components on a page and take a "known-good" screenshot of that page or component. As changes are made, the tests are then re-run and new screenshots taken. If any screenshots have changed, then the user of the testing tool is informed, and either approves the change or flags a regression. Adoption of these tools should always be taken with caution, as they will often flag false positives.

Cross-Functional Testing

So far, we have discussed ways of checking how your application behaves and whether it satisfies the requirements placed upon it. This kind of testing is often called "functional testing," but there are other attributes of your application that you will want to test. These don't relate to any particular feature (or function), but instead deal with attributes of your application as a whole. The types of requirements in this category are often called "non-functional" (as they don't relate to any one function of your application), or, perhaps less confusingly, "cross-functional" (as they cut across each part of your application).

Testing these cross-functional requirements is just as important as testing the explicit functions of your application. The kinds of things you may classify as cross-functional include security, usability, accessibility, performance, ability to handle load, and device compatibility. For the most part, these kinds of cross-functional requirements are common to many web applications, although some of the details may vary. Because of this, there is a large suite of common tools available to help you test your application, and many of them just need appropriate configuring.

However, these tools are not always necessary. For example, if you establish a requirement that an API call returns in less than 200ms, then you could write an HTTP request wrapped in a timer and make an assertion on that timer, like any other test. But if your requirement is that a page loads and renders to a usable state in less than 2s, then writing a test to do this becomes more complicated due to the complexity of rendering. A performance testing tool can be configured to make this type of assertion.

For other cross-functional requirements, such as security and accessibility, tools can help, but like functional testing, it often requires a bespoke approach, and there is no automation silver bullet. The general testing approach to these kinds of factors is often

very much like normal testing, but is across the whole product (or focusing on where the changes have occurred) and may require specialized skills. More details on how to quickly check accessibility can be found in the Accessibility chapter, but simple security checking can be executed in a similar way to checking other features—by trying negative test cases and checking that they work as expected. The Security chapter will discuss some security aspects to watch out for, which can help you develop test cases, but security testing often requires you to try very non-obvious ways to break the application, which can benefit from specialized skills for apps with particularly stringent security requirements.

Like any other type of testing, testing non-functionals like security and accessibility does not prove the absence of bugs, but performance and load testing operate in a completely different way. Whenever you run these tests, you always get a result that can be evaluated against a benchmark, rather than it simply being a "pass" or "fail" scenario. These types of tests are also very environment specific, which can make them hard to run regularly as part of your normal development cycle.

Load testing (sometimes called volume testing) is a type of testing where the application is given a lot of simulated traffic to check that it behaves correctly under load. However, a load test running against a local development environment might not give meaningful results, as the hardware the application is running on may be significantly different. Some organizations configure their testing environments to have the same hardware as their live environment to allow these environments to be used for load testing, but this can be prohibitively expensive. Cloud computing can be helpful in this area, allowing you to spin up an environment that mirrors production for load testing. Some organizations feel confident enough to run their load testing on their live environment without negatively affecting their customers.

Load testing is often done at scale, and therefore uses quantifiable metrics to determine its success or failure. To set these metrics, you must first determine the baseline performance for your application, which is how it performs when it is not under any load—often with a single user. You should then determine the application's typical level of use—often based on real-world metrics if modifying an existing system, or a best guess based on what the organization knows—and the acceptable deviation from this level. This could be expressed in terms of percentage increase, or specific targets if hard metrics are known. Whether or not a system passes or fails a load test is then evaluated against these metrics. Load tests can also be modified to test "until destruction," where the number of requests or volume of traffic increases until the system completely fails, to give a good indication as to how much headroom your system has.

With this kind of testing, it is often useful to track the test results over time to identify trends, as well as checking for hard passes/fails against benchmarks.

Due to the unpredictability of environments when running load and performance tests, it is often necessary to repeat the tests to get meaningful results. Many tools will do this for you—for example, a performance testing tool will often render the same page 100 times and take an average that will give you more confidence than running it only once.

It can also be useful to check for these requirements by analyzing the behavior of real users, especially for performance and load tests. By monitoring real user performance, you can get a much better idea of the actual performance of your site than by simulating a load test (although load tests are still important for gaining early confidence). Monitoring things like server response time and CPU is discussed in further detail in the In Production chapter. For testing the performance of your front-end code, a suite of testing tools known as RUM (real user monitoring) allow you to capture analytics on statistics from real devices, which can be much nosier but give you a better idea of how your site performs on a wide range of real-world devices.

Summary

Testing is key to any software project, and should be used to not only ensure that a system works correctly, but that the correct system was built. Some level of testing can be done automatically by writing test code that runs individual components of the system in isolation (unit tests), or wired together (integration tests) in various states, and asserting that their responses or actions are correct.

Test-driven development is a mechanism for using tests to help structure your development. Tests are written first, and then the code needed to make it past second, finally once the functionality is proven correct the code is tidied up (refactoring). This allows a problem to be broken down into small, individually testable pieces, and can add up to be an effective suite for checking that nothing has accidentally broken or regressed.

Manual testing occurs when a user drives the application and checks that its behavior both meets the spec and makes sense. This can have some level of automation applied to it, where test code can drive the application as a user does—these are known as end-to-end tests. Manual testing is also effective at spotting visual errors, which end-to-end tests can miss. They can also be relatively brittle toward changes. Manual testing is best applied when the elements to check are ill-defined or abstract, and

human creativity takes precedence. Exploratory testing is also something humans can undertake, where a tester takes a free-flowing journey through an application to test scenarios on the fly, rather than those which are pre-defined as requirements.

Behavior-driven development takes test-driven development a step further and adds specification at a higher level of abstraction. TDD might focus on unit testing, but BDD usually looks at integration or end-to-end testing. BDD is mostly used to ensure that there is a shared understanding between the developers, testers, and product representatives (the “three amigos”), who document the different scenarios the product is used in, often expressed in the Given-When-Then form.

The final aspect of testing to consider addresses components that go across all the functions of your application. This can include performance testing, security holes, or accessibility flaws. Like functional testing, automation can assist, but applying human skills to testing these will help catch areas that automation can miss.