

K R I S T I N   J A C K V O N Y

# THE COMPLETE SOFTWARE TESTER

CONCEPTS, SKILLS, AND STRATEGIES  
FOR HIGH-QUALITY TESTING



# The Complete Software Tester

*Concepts, Skills, and Strategies for High-Quality Testing*

Kristin Jackvony

Copyright © 2021 Kristin Jackvony

All rights reserved

No part of this book may be reproduced, or stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without express written permission of the publisher.

Cover design by: Vanessa Mendoza

# Contents

- [Title Page](#)
- [Copyright](#)
- [Introduction](#)
- [Note](#)
- [Part I: Why Do We Test?](#)
- [Chapter 1: Why We Test](#)
- [Chapter 2: Think Like a Tester](#)
- [Chapter 3: Why We'll Always Need Software Testers](#)
- [Part II: Manual Testing](#)
- [Chapter 4: Testing a Text Field](#)
- [Chapter 5: Break Your App with This One Weird Trick](#)
- [Chapter 6: CRUD Testing](#)
- [Chapter 7: Testing Postal Codes](#)
- [Chapter 8: Testing Date Fields](#)
- [Chapter 9: Testing Phone Number Fields](#)
- [Chapter 10: Testing Buttons](#)
- [Chapter 11: Testing Forms](#)
- [Chapter 12: Four Reasons You Should Test Input Validation \(Even Though It's Boring\)](#)
- [Chapter 13: Three Ways to Test Output Validation](#)
- [Chapter 14: Testing Back Buttons](#)
- [Chapter 15: Six Tips for File Upload Testing](#)
- [Chapter 16: Testing the Login Screen](#)
- [Chapter 17: Cross-Browser Testing](#)
- [Chapter 18: Before You Log That Bug...](#)
- [Chapter 19: How to Reproduce a Bug](#)
- [Chapter 20: How to Log a Bug](#)
- [Part III: How Applications Work](#)
- [Chapter 21: How HTTP Requests Work](#)
- [Chapter 22: Internet Routing](#)

- [Chapter 23: Encoding and Encryption](#)
- [Chapter 24: HTTPS, Tokens, and Cookies](#)
- [Chapter 25: The Joy of JWTs](#)
- [Chapter 26: Database Testing](#)
- [Chapter 27: Testing with Relational Databases](#)
- [Chapter 28: SQL Query Secrets](#)
- [Chapter 29: Testing with Nonrelational Databases](#)
- [Chapter 30: Serverless Architecture](#)
- [Part IV: API Testing](#)
  - [Chapter 31: Introduction to REST Requests](#)
  - [Chapter 32: Getting Set Up for API Testing](#)
  - [Chapter 33: Testing GET Requests](#)
  - [Chapter 34: Testing POST Requests](#)
  - [Chapter 35: Testing PUT Requests](#)
  - [Chapter 36: Testing PATCH Requests](#)
  - [Chapter 37: Testing DELETE Requests](#)
  - [Chapter 38: Response Codes](#)
  - [Chapter 39: Postman Assertions](#)
  - [Chapter 40: Using Variables in Postman](#)
  - [Chapter 41: Organizing Your API Tests](#)
  - [Chapter 42: Understanding JSON Data](#)
  - [Chapter 43: API Contract Testing Made Easy](#)
- [Part V: Mobile Testing](#)
  - [Chapter 44: The 12 Challenges of Mobile Testing](#)
  - [Chapter 45: Manual Mobile Testing](#)
  - [Chapter 46: Seven Tips for Mobile Automated Testing](#)
- [Part VI: Security Testing](#)
  - [Chapter 47: Introduction to Security Testing](#)
  - [Chapter 48: Using Dev Tools to Find Security Flaws](#)
  - [Chapter 49: Testing for IDOR Vulnerabilities](#)
  - [Chapter 50: Introduction to Cross-Site Scripting](#)
  - [Chapter 51: Three Ways to Test for Cross-Site Scripting](#)
  - [Chapter 52: Introduction to SQL Injection](#)
  - [Chapter 53: Introduction to Session Hijacking](#)
  - [Chapter 54: An Introduction to Mobile Security Testing](#)
- [Part VII: Performance Testing](#)
  - [Chapter 55: Introduction to Performance Testing](#)

- [Chapter 56: How to Design a Load Test](#)
- [Part VIII: Usability and Accessibility Testing](#)
- [Chapter 57: Localization and Internationalization Testing](#)
- [Chapter 58: User Experience Testing](#)
- [Chapter 59: Accessibility Testing](#)
- [Part IX: Software Development Basics](#)
- [Chapter 60: Code Like a Developer](#)
- [Chapter 61: Command-Line Basics](#)
- [Chapter 62: Coding Definitions](#)
- [Chapter 63: Object-Oriented Programming](#)
- [Chapter 64: Passing Parameters](#)
- [Chapter 65: Setting Up Node](#)
- [Chapter 66: Arrow Functions](#)
- [Chapter 67: Promises](#)
- [Chapter 68: Async/Await](#)
- [Chapter 69: Debugging for Testers](#)
- [Chapter 70: Seven Steps to Solve Any Coding Problem](#)
- [Chapter 71: Introduction to Git](#)
- [Chapter 72: Six Tips for Git Success](#)
- [Chapter 73: Merge Conflict Resolution](#)
- [Chapter 74: A Gentle Introduction to Regex](#)
- [Chapter 75: Logging, Monitoring, and Alerting](#)
- [Part X: Automated Testing](#)
- [Chapter 76: Why Automate?](#)
- [Chapter 77: When to Automate](#)
- [Chapter 78: Rethinking the Pyramid: The Automation Test Wheel](#)
- [Chapter 79: The Automation Test Wheel in Practice](#)
- [Chapter 80: Unit Tests](#)
- [Chapter 81: Component Tests](#)
- [Chapter 82: Services Tests](#)
- [Chapter 83: What API Tests to Run and When to Run Them](#)
- [Chapter 84: Setting Up UI Tests](#)
- [Chapter 85: Understanding the DOM](#)
- [Chapter 86: Locating Web Elements](#)
- [Chapter 87: Automating UI CRUD Testing](#)
- [Chapter 88: Automated Form Testing](#)
- [Chapter 89: Automated Visual Testing](#)

- [Chapter 90: Automated Security Testing](#)
- [Chapter 91: Automating Load Tests](#)
- [Chapter 92: Automated Accessibility Tests](#)
- [Chapter 93: Automation Wheel Strategy: Moving from What to How to When to Where](#)
- [Chapter 94: How Flaky Tests Destroy Trust](#)
- [Part XI: Testing Strategy](#)
- [Chapter 95: The Power of Not Knowing](#)
- [Chapter 96: The Power of Pretesting](#)
- [Chapter 97: Your Future Self Will Thank You](#)
- [Chapter 98: How to Design a Test Plan](#)
- [Chapter 99: Organizing a Test Plan](#)
- [Chapter 100: The Positive Outcomes of Negative Testing](#)
- [Chapter 101: What to Put in a Smoke Test](#)
- [Chapter 102: What to Test When There's Not Enough Time to Test](#)
- [Chapter 103: How to Keep Your Test Cases From Slowing You Down](#)
- [Chapter 104: Confused? Simplify!](#)
- [Chapter 105: Six Steps to Writing an Effective Bug Report](#)
- [Chapter 106: Should You Hunt for That Bug?](#)
- [Chapter 107: Why You Should Be Testing in Production](#)
- [Chapter 108: What to Do When There's a Bug in Production](#)
- [Chapter 109: Fix All the Things](#)
- [Chapter 110: The Hierarchy of Quality](#)
- [Chapter 111: Measuring Quality](#)
- [Chapter 112: Managing Test Data](#)
- [Chapter 113: A Question of Time](#)
- [Chapter 114: Why the Manual Versus Automation Debate Is Wrong](#)
- [Chapter 115: Tear Down Your Automation Silos](#)
- [Chapter 116: Stop Writing So Many UI Tests](#)
- [Chapter 117: Five Reasons You're Not Ready for Continuous Deployment](#)
- [Part XII: Soft Skills for Testers](#)
- [Chapter 118: Ask Your Way to Success](#)
- [Chapter 119: Seven Excuses Software Testers Need to Stop Making](#)
- [Chapter 120: Six Testing Personas to Avoid](#)
- [Chapter 121: How to Train Your Dev](#)
- [Chapter 122: Get Organized for Testing Success](#)

[Chapter 123: Time Management for Testers](#)

[Chapter 124: How to Be Seen](#)

[Acknowledgements](#)

[About The Author](#)

# Introduction

## Why I Wrote This Book

I discovered my love for software testing in 2009 when, after being laid off from my job as a support person for a small tech start-up, I took a job as a quality assurance intern at another company. On my first day there, I realized that software testing was something I wanted to do for the rest of my life.

I wanted to learn everything I could about software testing, but I found very few books that could teach me. I wound up learning through trial and error, reading blog posts, and talking with my co-workers.

Today there are some excellent books about specific areas of software testing, such as exploratory testing, Agile testing, and test automation, but I'm not aware of any book that aims to be a complete reference on testing. I started blogging about testing in 2012, and my occasional posts turned into weekly posts in 2017. It occurred to me that these posts could serve as the foundation for a comprehensive book on testing, and *The Complete Software Tester* was born.

## Who This Book Is For

This book is for anyone who cares about the quality of software. I wrote it with new software testers in mind; this is the book I wish I could have read when I was first starting out. You'll learn what to think about when manually testing an application, how web applications work, and how to create a test plan.

Veteran testers will also find helpful information in this book. Manual testers who want to start writing test automation will learn good coding practices and automation basics. Experienced testers will be able to fill gaps in their skill sets by learning about performance testing, security testing, mobile testing, and accessibility testing.

Finally, software developers who want to learn more about the practice of

testing can read about techniques that will help them think like a tester. Testing and development are two very different skills, and developers who take the time to learn what to test will discover that they are now writing better code.

## What's in This Book

This book is divided into 12 parts. Part I serves as an introduction to the importance of software testing and software testers. Part II discusses manual testing and all the potential bugs you can find through manually exploring an application.

Part III looks at how web applications work, from authentication to server-client interactions to HTML and CSS, because understanding the structure of an application can help you create better test scenarios. Part IV is devoted to API testing and includes step-by-step instructions for manual API testing with Postman.

In Part V we look at the special testing requirements for mobile applications. Parts VI, VII, and VIII focus respectively on security testing, performance testing, and accessibility testing, which are helpful skills for any tester who wants to completely test their application and differentiate themselves from other testers.

In Part IX we look at software development basics. Many testers come to the software testing field without a solid background in coding principles or knowledge of how to use tools like the command line or version control software. This section aims to fill in the knowledge gaps for those testers. Part X surveys many types of test automation and is filled with examples to demonstrate how they work.

Part XI is devoted to testing strategy. The most important quality of any tester is the ability to think of all the ways an application can possibly go wrong. Other important skills are the ability to prioritize testing needs and the ability to make good choices about the right testing methods to use. In this part of the book, you'll learn to organize your test plans, reproduce tricky bugs, and manage your test data.

Finally, Part XII covers the soft skills that all good testers need. Here you'll discover how to work effectively with software developers, get organized, manage your time, and be your own champion in your testing career.

It's my hope that you will find this book to be a helpful resource throughout your career, and that you'll refer to it again and again whenever you have a testing problem or want to improve your testing skills. Happy testing!

## Note

Occasionally in this book I refer to the Contact List app. This is a test application that I wrote specifically for use with this book. You can find it at  
<https://thinking-tester-contact-list.herokuapp.com>.

The API documentation for the app can be found at  
<https://documenter.getpostman.com/view/4012288/TzK2bEa8>.

# Part I: Why Do We Test?

# Chapter 1: Why We Test

Most software testers, when asked why they enjoy testing, will say things similar to the following:

- I like to think about all the ways I can test new features.
- It's fun to come up with ways to break the software.
- I like the challenge of learning how all the parts of an application work together.

I certainly agree with all of those statements! Testing software is creative, fun, and challenging.

But this is not why we are testing. We test to learn things about an application to ensure that our end users have a good experience with it. Software is built to be used for something; if it doesn't work well or correctly, it is not accomplishing its purpose.

For example:

- If a mobile app won't load quickly, users will stop using it or will delete the app from their phone.
- If a financial app suffers a security breach, the company that developed the app will lose customers and may even be sued for damages.
- If an online store has a bug that keeps shoppers from completing their purchases, the company will incur significant losses in sales.

There are even documented cases of people losing their lives because of problems with software. In the 1980's, three patients died when the Therac-25 medical accelerator mistakenly delivered lethal radiation doses due to a race condition in the software.

So, while it's fun to find bugs, it's also extremely important to find them. And it's even more important to remember that the true test of software is how it behaves in production with real users. Often, testers keep their focus

on their test environment because that's where they have the most control over the software under test.

But I have seen situations in which testers did this and then were surprised when users reported that the feature didn't work in production! This happened because they had hard-coded their environment variables to match the test environment, and they didn't bother to test the feature once it was released to production.

Having features that work correctly in production is only one aspect of quality. We also need to make sure pages load within a reasonable amount of time, data is saved correctly, and the system behaves well under times of high use.

Here are some questions to ask about any application you are testing:

- Is it usable?
- Is it reliable?
- Is your data secure?
- Do the pages load quickly?
- Are API response times fast?
- Do you monitor production use, and are you alerted automatically if there's a problem?
- Can you search your application's logs for errors?

It's fun to test and find bugs. It's fun to check items off in test plans. It's fun to see test automation run and pass. But none of that matters if your end user has a poor experience with your application. That is why we test.

# Chapter 2: Think Like a Tester

In 2017, I attended a large international computing conference where not a single workshop or presentation was focused on software testing. At this conference, I met computer science students who didn't even know software testing was a career!

Many large software companies no longer employ software testers, feeling sure that software developers are all they need to validate the quality of their software. And many developers focus solely on test automation, working from acceptance criteria in development stories and looking at the code rather than manually interacting with the software. How is that trend working out for end users?

Just last year, I experienced the following three bugs in three weeks:

- I received a (legitimate) email informing me that I had some money to accept from an online payment service. The email included an Accept the Money button to click. When I clicked it, I got this message: “The previous page is sending you to an invalid URL.”
- When I was using a mobile app, a screen that I needed never loaded, staying permanently blank.
- While I was writing a blog post, my blog hosting service experienced a page load error when I tried to add an image.

Three weeks, three major companies, three bugs. This is what comes from not employing people who think and act like testers.

It's true that the whole software development team owns quality and that quality is everyone's responsibility. There are many people on a software development team who care deeply about certain areas of an application:

- Developers write unit and integration tests to check the quality of their code.
- Product owners care whether the feature does what it's supposed to do.

- UX designers care whether the user journey is intuitive.
- Security testers check the software for vulnerabilities.
- Performance engineers care about the application's response time.

But only true software testers care so much about the quality of the application that they'll do things like the following:

- Type `~!@#$%^&*()=-_+{}|[]:;';<>?./` into every text field to test for invalid character handling
- Try to purchase -1, 999999999999, 1.3415, and foo of something
- Enter a birth year of 3019 to see what happens
- Click every button twice to check for multiple submissions
- Click the forward and back buttons on every page of a website
- Test 48 different permutations of feature sets to be as thorough as possible
- Create dozens of test users with myriad security settings to have a variety of scenarios ready for testing at a moment's notice
- Become experts on a particular feature and provide documentation and assistance to other testers
- Test the same feature in the QA, staging, and production environments to be absolutely sure it is working everywhere
- Test every feature on every supported browser and every supported mobile device

This is why we need software testers who test. We need people who will continually ask themselves: “How can we break this?” “What haven’t we tested yet?” and “What features will be used with this?” We need software testers who don’t rush into writing automation without first interacting with a feature. We need software testers who remember that the goal of all their efforts is to give the user a positive, bug-free experience.

# Chapter 3: Why We'll Always Need Software Testers

A current positive trend in the software industry is to have the whole team own quality. This means software developers think about quality in their code, test their own code, and develop and contribute to test automation frameworks. Some companies have taken this trend to mean dedicated testers are no longer needed. But this is not the case! Following are three reasons we will always need software testers.

## **Reason #1: Teams Change**

Life brings about changes, and even a perfect team doesn't last forever. A team member could retire, take another job, or move to another team. New members could join the team. I've heard that every time a team changes by even one person, it becomes a brand-new team. This means there will be new opportunities to coach the team to build in quality.

Even if the team members don't change, other changes can occur that can challenge a team, such as a new technical problem to solve, a big push toward a deadline, or a sudden increase in the number of end users. All of these changes might result in new testing strategies, which means the team will need a software tester's expertise.

## **Reason #2: Times Change**

When I started my first software testing job I had never heard of the Agile model of software development. But soon it became common knowledge, and now practically no one is using the old Waterfall model. Similarly, there was a time when I didn't know what CI/CD (Continuous Integration/Continuous Deployment) was, and now it's a goal for most teams. There are pushes to shift testing to the left (unit and integration tests running against a development build), and to shift testing to the right (testing in production).

Some of these practices may prove to be long-lasting and some will be replaced by new ideas. Whenever a new idea emerges, new ways of thinking and behaving are necessary. Software testing experts will be needed to determine the best ways to adapt to these new strategies.

### **Reason #3: Technology Changes**

The tools and languages that are in use today won't be in use forever. Someone will come along and create a newer, more efficient tool or language that will replace the previous one. For example, Cypress recently emerged as an alternative to Selenium WebDriver, which has been the automated browser testing tool of choice for years. And many companies are moving toward cloud providers such as Amazon Web Services or Azure Web Services to reduce the processing load on their servers.

When a team adopts a new technology there will always be some uncertainty. As a feature is developed, changes may be made to the configuration or coding strategy, so it may be unclear at first how to design test automation. It can take a team a while to adapt and learn what's best. A testing expert will be very valuable in helping to make these decisions.

Change happens, and teams must adapt to change; therefore, it is helpful to have a team member who understands the best way to write a test plan, conduct exploratory testing, or evaluate risk. Don't go looking for a job as a software developer! Your testing expertise is still needed.

## Part II: Manual Testing

# Chapter 4: Testing a Text Field

The most basic field in a web application is the text field. It seems so ordinary, and yet it is one of the most important things we can test because text fields provide an entryway into an application and its underlying database. Validation on a text field is what keeps bad data from getting into the database. Bad data can cause all sorts of problems for end users and engineers. Validation can also prevent cross-site scripting (XSS) attacks and SQL injection attacks.

There are myriad ways to test a text field. First, let's imagine we are testing a text field with absolutely no information about what it does:

A wireframe diagram of a web form. At the top is a rectangular input field. Below it is a large, empty rectangular area. At the bottom is a rectangular button with the word "Submit" centered inside it. The entire form is enclosed in a thick black border.

- Click Submit without filling in the text field.
- Press the space bar several times in the text field and then click Submit.
- See how many characters you can fit in the text field and then click Submit.

- Fill the field with as many numbers as you can and then click Submit.
- Add a negative sign, fill the field with as many numbers as you can, and then click Submit.
- Enter every nonalphanumeric field on the keyboard and click Submit. If you get an error, see whether you can narrow down which key (or keys) is causing the error.
- Enter non-ASCII characters and emojis and click Submit. If you get an error, see whether you can narrow down which symbol (or symbols) is causing the error.
- Try cross-site scripting by entering this script: `<script>alert("I hacked this!")</script>` Click Submit. If you receive a pop-up message, you know the field is vulnerable to cross-site scripting.
- Try a SQL injection attack, such as: `FOO');` **DROP TABLE USERS;** — Click Submit. See whether the database table was deleted. Do not try this on your production database!

Next, let's assume you have some knowledge about what is supposed to be entered into this text field and what the boundaries are on the data:

- Try entering a value that is a different data type from what is expected; for example, if this text field is expecting a value of currency, try entering a string or a date.
- If the field is expecting a string, try entering a string with one less character than expected, one more character than expected, the lower limit of characters expected, the upper limit of characters expected, and twice the maximum number of characters expected.
- If the field is expecting a numeric value, try entering the maximum value, the minimum value, a value above the maximum, a value below the minimum, and a value twice the maximum value.
- If the field is expecting an integer, try submitting a value with a decimal point.
- If the field is expecting a float, try submitting a value with two decimal points.
- If the field is expecting a value of currency, try submitting a value with more than two digits after the decimal point.
- If the field is expecting a date, try entering the maximum date, the minimum date, one day over the maximum date, one day before the

minimum date, and a date 100 years above or below the limit.

- For date fields, try entering a date that doesn't make sense, such as 6/31/17 or 13/13/17.
- If the field is expecting a time, try entering a time that doesn't make sense, such as 25:15.
- If the field is expecting a phone number, try entering a number that doesn't conform to the expected format.

For all the preceding tests, find out what sort of error message you are supposed to be receiving, and verify that you are getting the correct message.

Finally, let's think about automation. Assuming you have thoroughly tested your text field manually, it's probably not necessary to automate every single one of your tests. Moreover, most forms have more than one text field, and having many tests for each individual field could result in many time-consuming tests. Nonetheless, there are some tests that you might want to automate. Here are a few suggestions:

- Submitting a null value
- Submitting an empty string
- Submitting a value that meets the testing criteria (the happy path)
- Submitting the maximum number of characters or maximum value
- Submitting the minimum number of characters or minimum value
- Submitting just above the maximum number of characters or maximum value
- Submitting just below the minimum number of characters or minimum value

This list of tests is not meant to be exhaustive; it's just a way to get you to start thinking about the vast number of tests you can run on a single field. Never assume the developer who coded the field has put in the appropriate validation; check it for yourself! I once tested a date field that had a limit put on the year that could be entered: it couldn't be before 1900 or after the present year. While I did receive the appropriate message when I entered a value of 1880, I discovered I could enter a date from the year 1300!

# Chapter 5: Break Your App with This One Weird Trick

Sometimes the same bug can pop up in more than one place, and we can miss it every time. This happened to me once while I was testing file uploads and downloads. When the developers on my team initially coded the upload functionality, I dutifully tested all kinds of filenames: long names, short names, names without extensions, names with capital letters, and so on. Everything looked great, but I had missed one important test: names with spaces. I had tested only one-word filenames, like sunrise.jpg. I had forgotten to test names like Grand Canyon.jpg, and as it turned out, uploads with spaces in them weren't working correctly.

Spaces are easy to forget to test because they are invisible! That's why it's important to test with spaces in all text fields, not just filename fields. For example, when you are testing a first name field, make sure you test it with a first name that consists of two words, such as Mary Jo.

It's also important to remember to test with leading and trailing spaces for cases in which users mistakenly hit the space bar when entering text. Although your developer should be trimming this whitespace when processing input, if they forget to do this, it can cause trouble. For example, you may wind up in a situation where you want to sort a list of names alphabetically by last name, but the name Smith appears at the top of the list because someone entered " Smith", with a leading space.

Similarly, your users might have problems logging in to your application even though they are using the right username. This could occur because when they set their username they accidentally put a space at the end of the name and the space was not trimmed by the developer. As a result, they are trying to log in with "catLover" when they should be logging in with "catLover".

Back to my story: after the bug with file uploads was discovered and fixed, I carefully tested uploads again, this time being sure to include

filenames with spaces in the beginning, middle, and end of the name.

Our next development task involved the ability to resize a file upon request. When this functionality was ready I started running all kinds of tests: resizing by height only, by width only, and by height and width; resizing various file types; and so on. While I was testing, the developer who worked on the feature mentioned he had just discovered a bug: the files wouldn't resize if they had spaces in their names, because the spaces weren't being encoded properly. I'd like to think I would have discovered this eventually, but who knows? I was more focused on the new functionality than I was on regression testing.

The bug was fixed, and I carefully retested the resizing functionality. I added spaces in the beginning, middle, and end of the filenames and I used every kind of special character on my keyboard. Surely, I thought, this must be the end of this bug.

Our next development task concerned checking the size and type of files being uploaded. We didn't want to be able to upload a file if it was larger than the application was told to expect, or if it was the wrong file type. I tested with all kinds of file sizes and types and all kinds of mismatches. With each mismatch, I verified that the file was checked and rejected. I did a great deal of regression testing as well. It occurred to me to test this functionality using different filenames, but since I had already verified that filenames of all kinds were working with the resizing capability, doing it here seemed like overkill, so I chose not to do it.

I was wrong! As it turned out, the system that was handling file type checking was different from the system that was handling file resizing, and once again, spaces in filenames weren't being encoded properly. A developer noticed the system was not checking files that had spaces in their names. I was bitten by the filename space bug once again.

It's amazing how many things one simple space can break! When you are testing anything that can accept text, from a simple form field to a file upload, be sure to remember those invisible spaces.

# Chapter 6: CRUD Testing

In spite of its unappealing name, CRUD testing is extremely important! CRUD stands for Create, Read, Update, and Delete. Much of software testing involves these operations.

Let's take a look at how CRUD works. Imagine you have a simple form that is designed to add a user to the system:

The diagram shows a rectangular form with a thin black border. At the top center, the text "Add User" is displayed in a bold, sans-serif font. Below this, on the left, is the label "First Name". To the right of the label is a rectangular input field with a thin black border. Inside the input field, the name "Fred" is written in a standard sans-serif font. At the bottom center of the form is another rectangular input field with a thin black border. Inside this field, the word "Submit" is centered in a bold, sans-serif font.

- When you add a new user to the system, that is a Create operation.
- When you retrieve the user's information, that is a Read operation.
- When you edit the user's information, that is an Update operation.
- When you delete the user from the system, that is a Delete operation.

The most important thing to know about CRUD testing is that it's not enough to rely only on what you see in your UI (user interface, or what you see on the computer screen) to confirm that a field's value has been created or

changed. This is because the UI will sometimes cache a value for more efficient loading in the browser. To be absolutely sure the value has changed you must check the database where your value is stored. So when you do CRUD testing, you should be confirming that your value is set in two places: the UI and the database.

## **How to Test a Create Operation**

For the simple form in the preceding illustration, we'll enter the user's first name into the text field and click Submit. Then we'll look at the Users page of our imaginary application and verify that the new user is present:

| Users     |                  |
|-----------|------------------|
| <b>id</b> | <b>firstName</b> |
| 1         | Fred             |
|           |                  |

And there it is! Finally, we need to query our database to make sure the value was saved correctly there. In our imaginary database, we can do this by running:

```
SELECT * from Users
```

This will give us a result that should include a record with an id of 1 and a firstName of Fred.

To thoroughly test the Create function, we'll need to test with both valid and invalid input. Let's imagine our first name field has the following rules:

- It is a required field.
- It must have at least two characters.
- It must have 40 or fewer characters.
- It should only have alphanumeric characters or hyphens and apostrophes; no other symbols are allowed.

To test these validation rules, we can first submit valid entries of all kinds and verify that they have been saved to the database. Then we can try submitting invalid entries that break one or more of the aforementioned rules and verify that we get an appropriate error message and that the value is not saved to the database.

## How to Test a Read Operation

We actually started testing the Read operation when we checked the Users page to verify that our new user was added. But there is something else that is important to test: we need to find out what happens when bad data is in the database and we are trying to view it in the UI.

Here's what bad data might look like in the database:

| Users |   |
|-------|---|
| Id    | firstName                                     |
| 1     | Fred  |
| 2     | NULL  |
| 3     | ""  |
| 4     | Reallyreallyreallyreallyreallylongfirstname   |
| 5     | <script>alert("You've been hacked!")</script> |

As we discussed earlier, the first name field has some conditions: it is a required field, it must have at least two characters, it must have 40 or fewer characters, and it should only have alphanumeric characters or hyphens and apostrophes. As we can see in our table, we've got lots of bad data:

- User 2 has no data in the first name field.
- User 3 has an empty string for a first name.
- User 4 is violating the rule that the name must have 40 or fewer characters.
- User 5 is violating the rule that only hyphens and apostrophes are allowed for symbols.

What should happen when we view the Users list in our application? That will depend on what the product designers decide. They may choose to display bad data as long as it is not a security risk like the first name for user 5, which is actually a stored XSS attack. Whatever the rules are for display, it's important to test that those rules are respected.

You may be saying to yourself (or a developer may be saying to you), “Displaying bad data won’t be an issue, because we are putting good validation in place to make sure bad data won’t get in the database to begin with.” While this is absolutely standard practice today, there will always be cases where bad data will slip in. I once tested a PATCH operation that allowed phone numbers to be inserted into a record. Although validation was taking place when the PATCH body was formed correctly, I discovered an edge case in which the data was accepted without validation if the PATCH body was formed incorrectly. This bug was allowing bad phone numbers into the database.

## **How to Test an Update Operation**

In our discussion of the Read operation, I mentioned how important it is to test scenarios in which the data in the database is invalid. This is also true for Update operations. Just because a text field is supposed to be required and to have a certain number of characters doesn’t mean that’s how it is currently represented in the database!

Following is a matrix of testing scenarios for editing the text field. As with the Create operation, we need to be sure to test that the newly edited field is correct in the UI *and* in the database after the update:

| Original Value     | Edited Value        | Expected Result                |
|--------------------|---------------------|--------------------------------|
| NULL               | “ “ (empty string)  | Error message                  |
| NULL               | Bad value           | Error message                  |
| NULL               | Good value          | New value is saved to database |
| “ “ (empty string) | NULL                | Error message                  |
| “ “                | Bad value           | Error message                  |
| “ “                | Good value          | New value is saved to database |
| Bad value          | NULL                | Error message                  |
| Bad value          | “ “                 | Error message                  |
| Bad value          | Different bad value | Error message                  |
| Bad value          | Good value          | New value is saved to database |
| Good value         | NULL                | Error message                  |
| Good value         | “ “                 | Error message                  |
| Good value         | Bad value           | Error message                  |

Be sure to vary the bad values you are testing with so that you are covering a number of different validation scenarios. For the good values you are testing with, make sure you test hyphens and apostrophes, numbers and letters, and the upper and lower limits of the character count.

I once encountered a situation in which I was testing a person's contact information. There were a number of incorrectly formatted phone numbers in the database. When I tried to update any of the person's contact information, I received a message stating that the record couldn't be updated because invalid data was present. This occurred even when I was trying to update the invalid phone number in question!

## How to Test a Delete Operation

The main thing to test with a Delete operation is that the value has been deleted from both the UI and the database. But just as with Read and Update, you'll want to make sure you can delete bad values. For example, if your first name field has 41 characters in it (violating the rule that no more than 40 characters should be added), you'll want to make sure you can delete it in the UI.

You may be wondering how to find all these invalid values for testing. While it is possible to find them by searching through the existing records, the easiest way to test them is to put them in yourself. This is why it's important to be familiar with the query language for the database your application is using. We'll discuss databases in Part III.

# Chapter 7: Testing Postal Codes

A text field with a postal code looks so simple, and yet it can be one of the most complex things to test on a form. There are two important questions to ask before you start testing postal codes:

- What countries does the application support?
- What formats will the application accept for postal codes?

In the United States there are two formats for ZIP codes: the traditional five-digit format, as in 10012, and the ZIP+4 format, as in 10012-1234. The latter format is where the second question comes into play. Will the application be accepting ZIP+4 codes? Will it require the hyphen between the first five digits and the next four digits, or will it accept a space as well? What about just nine straight digits, with no hyphen or space in between? If nine straight digits will be accepted, it's important to also verify that six, seven, and eight digits will not be accepted.

There is another very important thing to test with U.S. ZIP codes: the leading zero. Often, applications will strip leading zeros off a number upon submission. If your application uses U.S. ZIP codes, it's important to test that ZIP codes submitted with a leading zero are saved correctly to the database.

In Canada, postal codes are six characters long and follow this pattern of letters and numbers: A1A 1A1. It's important to clarify with the developer whether the space between the two groups of three characters will be expected or whether you can submit the code with no space. The validation should expect the correct letter–number pattern and should reject postal codes that begin with a number instead of a letter.

Many countries have five-digit codes, and some, such as Russia and India, have six-digit codes; both are easy to validate. But consider Great Britain, whose postal codes consist of two sections: the first section can have between two and four characters and the second section always has three characters. There is a space between the two sections, and the postal code

always starts with a letter, not a number. When testing these postal codes, be sure to try several different, valid codes from various places in Great Britain, with two, three, and four characters in the first section. You can also test with codes that have the right number of characters but have the space in the wrong place, or with codes that have a number as the first character.

If your application expects postal codes from more than one country, there is a third question to ask:

- Is there a separate validation pattern for each type of postal code?

If the answer is yes, the script may first look to see what country the address contains and then use the appropriate validation. In this case, it's a good idea to test that you can't choose United States for your country and then add A1A 1A1 as the postal code. Alternatively, the validation pattern may be chosen based on the number of characters submitted. If six or seven characters (including the space) are submitted, a Canadian validation pattern could be used. If five, nine, or 10 characters (including the hyphen) are submitted, a U.S. validation pattern could be used. Understanding what validations the developer is using will allow you to craft appropriate test cases. For example, in a scenario where only U.S. and Canadian postal codes are used, eight characters should never be accepted.

It's also possible that your developer has used a universal postal code validation pattern. In the Contact List app I wrote for use with this book, I used the npm validator package. I set the validator to check for a valid postal code without knowing what country has been chosen. The validator checks the postal code against formats from all over the world.

Remember to verify that the valid postal codes you have submitted have been saved correctly to the database, that invalid postal codes return an appropriate error and are not saved to the database, and that retrieved postal codes are displaying correctly. If there are already invalid postal codes in the database due to poor validation practices in the past, be sure to verify that it's possible to edit both invalid and valid postal codes.

Asking the questions listed in this chapter can help you and your developer recognize potential issues with postal code validation before

testing has begun.

# Chapter 8: Testing Date Fields

Date fields are another data type that seems simple to test. After all, dates are standard throughout the world: there's a month, a day, and a year. But as you will learn in this chapter, there are many factors to consider and many scenarios to test.

There are three main areas to think about when testing a date field:

- What format will be accepted?
- How will the date be stored in the database?
- How will the saved date be displayed?

There are many ways to format a date when entering it into a form. One very important factor to consider is whether the system is expecting the format used in the United States—month, day, year—or the format that is more common in other parts of the world—day, month, year. For example, if I try to enter 12/13/2017, it won't be accepted if the form was expecting a day-month-year format, since there is no 13th month. Likewise, if I enter July 4, 2018, as 7/4/18, it may be saved to the database as April 7.

Beyond day-month versus month-day formatting, there are still myriad ways to format dates. For example, will all four digits of the year be expected, or is it possible to enter just the last two digits? Will single-digit months and days be allowed, or is it necessary to precede those digits with zeros, as in 07/04/2018? What about spelling out the month, as in December 13, 2017? The developer should have a clear idea of what format or formats will be allowed and should clearly communicate it to the user. For example, a tooltip such as “mm/dd/yyyy” can be used inside the field to help the user know what format to use. In the Contact List app, I am expecting the date field to be in yyyy-MM-dd format, and I've included a tooltip that shows this.

Regardless of whether a tooltip is in place, it is important to test with a number of different date formats to ensure that an appropriate error message is displayed when the expected format is not followed. It is also important

that the date itself is checked to be sure it is a valid date; for example, 2/30/2016 should never be allowed, nor should 18/18/18.

Now let's consider whether the date will be stored in the database as a datetime field or as a string. Storing the date as a string can be problematic, for a few reasons. If the accepted format changes over time or if good validation was not in place in the past, the database could contain dates saved in two different formats: for example, 7/4/2017 and December 13, 2017. This will make it difficult to display the data in a consistent style. In addition, dates stored as strings are difficult to sort correctly by ascending or descending date. For instance, 2017-12-13 might be displayed before 3/7/2017 in an ascending sort. It is therefore best to store dates as datetime fields whenever possible, and testers should check for this.

The way dates are displayed is important as well. When a user calls up saved information, they should be able to read the dates easily; 2017-12-13T00:00:00 might be how a date is saved in the database, but a user won't be able to quickly interpret this date. The developer or designer should decide what date format would be best for display purposes and use it consistently throughout the application. Similarly, they should consider what should happen in the case of bad data. What if a date is saved in the database as simply December 13? Should it be displayed as 12/13/0000? Should it not be displayed at all? These are important scenarios to consider and test.

There is one final point to consider when testing date fields, and that is the upper and lower limits of the date. For example, are dates in the future ever allowed? What about dates from 100 years ago? Remember that the future and the past change every day! Let's say our application doesn't allow dates in the future. This means that as I'm writing this, 12/13/2025 is not an allowed date in our application. But in a few years, 12/13/2025 will be allowed. And of course, you may be reading this book years in the future, at which time 12/13/2025 will be a thing of the past!

# Chapter 9: Testing Phone Number Fields

Phone number fields are, without a doubt, the most time-consuming fields I have ever tested. Why do they cause so many headaches? Because, as we saw with postal codes, there are so many ways to get them wrong.

Let's take a look at some phone number patterns from around the world:

- In Mexico, phone numbers are 10 digits, plus an area code of either two or three digits. So the local number will be either 12 or 13 digits.
- In Italy, landline numbers are generally nine to 11 digits, but some can be as short as six. Mobile numbers are usually 10 digits, but there are also some nine-digit numbers.
- In Japan, phone numbers have an area code, an exchange number, and a subscriber number. Area codes can have from two to five digits. Generally, the entire number is limited to nine digits, so if the area code is longer, the exchange and subscriber numbers will be shorter.

International calling codes differ as well, ranging from one digit (such as +1 for the United States) to three digits (such as +351 for Portugal). So, between the international calling code and the phone number itself, the number of digits that might be expected for a phone number can vary widely.

Finally, let's think about number separation. While U.S. numbers are always separated in a 3-3-4 pattern, in other countries the numbers can be grouped in various ways. For instance, here are just some of the ways that phone numbers are grouped in England:

- (xxx) xxxx xxxx
- (xxxx) xxx xxxx
- (xxxxx) xxxxxx
- (xxxx xx) xxxx

With all of this variation to consider, how on earth can we validate international phone numbers?

Fortunately, developers don't have to try to come up with a validation regex on their own; that would make them (and you) go crazy! Rather, an international phone number formatting standard called E.164 has been developed, and many companies, including Microsoft and Google, have come up with regex patterns that can be used for E.164 validation. For testing, there are free websites that will let you know whether an international phone number is valid. You can create a list of international numbers to test, verify with the website that they are valid, and then try them in your application. For negative testing, simply use a sampling of invalid numbers and verify that you get an appropriate error message.

If you are fortunate enough to be testing a new application with no existing data, you can fend off most phone number headaches by clearly establishing validation rules for phone number fields. The easiest type of validation is one that accepts digits and no other characters. The developer can rely on UI formatting to put the number in an easily readable format, rather than allowing the user to do it themselves. This is the best course of action for U.S. phone numbers because it avoids ambiguity and ensures consistency. Otherwise, some users might enter numbers using only dashes, as in 800-867-5309, while others enter numbers using parentheses and a dash, as in (800) 867-5309. For international phone numbers, it may be difficult for the developer to determine what format to use. In this case, allowing the user to choose their formatting might be the best thing to do. In the Contact List app, I am using the npm validator package to validate that the phone number is valid, but I am also allowing the user to determine what formatting is best for display.

Many developers try to accommodate all styles by creating their own validation regex, but this makes things much more complicated. If the developer doesn't develop a good regex, a user could type in something like ((800)-867 5309 and the number will be accepted. When you are testing the phone number field, try entering values like 800-867-5309, 800(867)5309, and 8-0-0-867-5309. After you have demonstrated that it's possible to enter a badly formatted phone number, the developer's strategy may change!

As I mentioned earlier when discussing dates, it's very helpful to have hints in the UI so that the user knows what format to use. Providing a tooltip or an example number in the phone number field is a great way to communicate to users what they should be adding: for instance, "Include your country code at the beginning of the number" or "8005551234".

Now it's time to think about how bad phone numbers will be displayed to the user. If a number has been saved with formatting, will it be displayed with that formatting? Will some attempt be made to strip the formatting? Will it be displayed at all? I have seen all kinds of bad numbers in databases. One of my favorites was "dad's office number". No one can format that! A good strategy for formatting a number is to strip out all the non-number characters and then add whatever format is desired for display. For example, if the phone number is 11 digits and starts with a 1, the leading 1 could be stripped off and then the number could be formatted for display. With any other values, the "number" could be displayed as is, or it could not be displayed at all. Whatever is decided, it's important to test with any bad numbers that are currently in the database.

Another wrinkle not often considered is phone extensions. Many offices still use extensions to direct a call to a specific person. Sometimes phone fields allow extensions as part of the number, but this is a bad idea. How might the user indicate that an extension is part of the number? Will they enter 800-867-5309 ext. 1234? Or, perhaps, 800-867-5309×1234? Allowing these types of variations will mean allowing letters and other characters, which will make the phone number more difficult to validate. A far better solution is to include a separate database field for an extension. If your developer expresses interest in including the extension as part of the phone field, some testing with entries like 8008675309extextext...1234 should dissuade them.

Finally, remember to test the ability to edit phone numbers, especially numbers that are in an incorrect format. What will happen when the user enters an invalid phone number and tries to edit another field on the form? Will they be notified upon saving that the number is incorrect and needs to be fixed? This can be a good way to involve users in cleaning their own data. What should happen when the user tries to fix an invalid number? Ideally,

they should be able to save the value with a new, valid number.

# Chapter 10: Testing Buttons

Just like spaces, buttons tend to be easy to forget about. The Save button is so ubiquitous that it seems like it would just automatically work. But overlooking the testing of buttons on a page can also mean overlooking bugs. Once, a tester told me about new functionality she was testing on an existing web page. The new feature worked great, but her team had forgotten to test the Delete button. It turned out that the developers had neglected to account for the delete action in their new feature, and now Delete did nothing!

Here are a few things to do when testing buttons:

- **Test the happy path of the button.** Usually, buttons have some sort of message on them that tells you what they are supposed to do. So try it out and make sure the button delivers on its promise. Did the Save button really save your data? Did the Delete button delete it? Did the Clear button clear it? Did the Search button execute a search? (A back button is also common, but back buttons are tricky enough that I am saving them for Chapter 14.)
- **Misuse the button.** For example, quickly click the Save button twice when adding data. Were two records saved instead of one? Does the program get confused when you click a button twice? What about when you quickly click one button and then another? One of the most amusing bugs I found in my career was a Refresh button that redrew the screen, making the button larger every time I clicked it.
- **Make sure the button is there when it's supposed to be.** When we spend a lot of time testing an application, it's easy to get used to the page and not notice when things are missing. Think about your user stories when looking at the page. What would your user need when doing certain activities on that page? Follow the path the user would take and check for the buttons as you go.
- **Think about when the button is enabled and when it is disabled.**

Does it make sense? For example, is the Save button only enabled when all the required fields on your form have been filled out? Is the Clear button only enabled when a field has been dirtied? How do you know the button is enabled? Can you tell by looking at it? Does the button look enabled when it isn't? Does the button look disabled when it's really active? What happens when you click the button and it's not enabled? Do the enabled and disabled rules for the button make sense?

- **See whether you can hack your buttons.** For example, if you have a Save button that is disabled on your form because some of the required fields are missing, can you edit the HTML on the page so that it is enabled, and can you then use the button? If you view the HTML and you see there is a hidden button on the page, can you make it visible and active? At best, a bug like that is an annoyance that will allow inaccurate data into the database. At worst, the bug represents a way that a hacker can infiltrate your system. Imagine you have a button that should only be visible and enabled if the user is an administrator. If a malicious user can make the button appear on the screen and be active, they will have access to pages or features that only an admin should have. Your developer should include checks whenever a button is clicked to make sure the user has the rights to do what the button does.

If you have never edited the HTML on a web page while testing, here are a few simple instructions for the Chrome browser:

1. Click on the three-button menu at the upper-right of your screen and choose More Tools, then Developer Tools. A new panel will appear on the bottom or right side of your screen.
2. Right-click on the button you want to test, and click Inspect. In the Developer Tools panel, the HTML for that button will now be highlighted.

3. Right-click on that highlighted text and choose Edit as HTML. An editable text window will open.
4. If you see text such as “disabled’=disabled” delete the text. Click away from the editable field and see whether your button is now enabled. If it is, click on it and see what happens!
5. To find hidden buttons, look at the HTML in the Developer Tools and use the search bar to search for “button”.
6. If you find a button with the markup “hide”, try changing it to “show”. See whether you can get the button to appear on the page.

Buttons are one of the most important things to test. Just imagine a user’s frustration if the button they are trying to use is disabled or doesn’t do what they expect it to. By being diligent in our testing, we can ensure that our users will want to continue using our application.

# Chapter 11: Testing Forms

In the past few chapters, we looked at different types of text fields and buttons; now we'll discuss testing a form as a whole.

There are as many different ways to test forms as there are text field types! And unfortunately, testing forms is not particularly exciting. Because of this, it's helpful to have a systematic way to test forms that will get you through it quickly, while making sure you're testing all the critical functionality.

Let's take a look at a systematic approach to testing a form.

## **Step 1: Required Fields**

The first thing I do when I test a form is note which fields are required. Forms often denote required fields with an asterisk. Let's imagine in this scenario that the first name and last name are required. I want to verify that the form cannot be submitted when a required field is missing and that I'm notified of which field is missing when I attempt to submit the form.

1. I click the Save button when no fields have been filled out, and I make sure all the required fields have error messages.
  
2. I fill out some nonrequired fields, click the Save button, and verify that all the required fields have error messages.
  
3. I fill out the first name field, click the Save button, and verify that the last name field has an error message. Then I fill out just the last name field, click the Save button, and verify that the first name field has an error message.

4. I fill out all the fields—required and nonrequired—except for one required field, and verify that I still receive an error.
5. I fill out all the required fields, click the Save button, and verify that no error messages appear and that the entered fields have been saved correctly to the Contact List.
6. Finally, I fill out all the fields—required and nonrequired—and I verify that no error messages appear and that the entered fields have been saved correctly to the Contact List.

## **Step 2: Field Validation**

The next thing I do is verify that each individual text box has appropriate validation on it. First I discover what the upper and lower character limits are for each text field. I enter data for all the required fields except for the one text box I am testing. In that text box I do the following:

1. I try entering just one character.
2. I try entering the lower limit of characters, minus one character.
3. I try entering the upper limit of characters, plus one character.
4. I try entering a number of characters far beyond the upper limit.

In all of these instances, the form should not save and I should receive an

appropriate error message. Next:

1. I enter the lower limit of characters and verify that the form is saved correctly.
2. I enter the upper limit of characters and verify that the form is saved correctly.

Now that I have confirmed that the limits on characters are respected, it's time to try various letters, numbers, and symbols. For each text field, I find out what kinds of letters, numbers, and symbols are allowed. For example, the first name and last name fields should allow apostrophes (e.g., O'Connor) and hyphens (e.g., Smith-Clark), but should probably not allow numbers or other symbols. For each text field:

1. I try entering all the allowed letters, numbers, and symbols.
2. I try entering the letters, numbers, and symbols that are not allowed, one at a time, until I have verified that they are all not allowed.

For fields that have very specific accepted formats, I test those formats. For instance, in the postal code field, I should be able to enter 03773-2817 but not 03773-28.

Even though I've already tested all the forbidden numbers and symbols, I try a few cross-site scripting and SQL injection examples to make sure no malicious code gets through. (Cross-site scripting and SQL injection will be discussed further in Part VI.)

### **Step 3: Buttons**

I have already used the Save button dozens of times at this point, but

there are still a few things left to test here. Also, I have not yet tested the Cancel button. So:

1. I click the Save button several times in quick succession and verify that only one instance of the data is saved.
2. I click the Save button and then the Cancel button in quick succession and verify that the data is saved and there are no errors.
3. I click the Cancel button when no data has been entered and verify that there are no errors.
4. I click the Cancel button when data has been entered and verify that the data is cleared. I'll try this several times, with various combinations of required and nonrequired fields.
5. I click the Cancel button several times in quick succession and verify that there are no errors.

Once I've gone through a form in this systematic way, I've almost certainly found a few bugs to keep the developers busy. It's tedious, yes, but once the bugs have been fixed, I can move on to automation testing with confidence, knowing that I have thoroughly tested the form.

# Chapter 12: Four Reasons You Should Test Input Validation (Even Though It's Boring)

When I first started testing software, I found it fun to test text fields. It was entertaining to discover what would happen when I put too many characters in a field. But as I began my fourth testing job and discovered that once again I had a contact form to test, my interest started to wane. It's not all that interesting to input the maximum number of characters, the minimum number of characters, one too many characters, one too few characters, and so on, for every text field in an application!

However, it was around this time when I realized that input validation is extremely important. Whenever a user has the opportunity to add data in an application, there is the potential for malicious misuse or unexpected consequences. Testing input validation is a critical activity for the following four reasons.

## Security

Malicious users can exploit text fields to get information they shouldn't have. They can do this in three ways:

- **Cross-site scripting:** An attacker enters a script into a text field. If the text field does not have proper validation that strips out scripting characters, the value will be saved and the script will execute automatically when an unsuspecting user navigates to the page. The executed script can return information about the user's session ID, or even pop up a form and prompt the user to enter their password, which then gets written to a location the attacker has access to.
- **SQL injection:** If a text field allows certain characters such as semicolons, it's possible that an attacker can enter values into the field which will fool the database into executing a SQL command and return information such as the usernames and passwords of all the users on the site. It's even possible for an attacker to erase a data

table through SQL injection.

- **Buffer overflow attack:** If a variable is configured to have enough memory for a certain number of characters, but it's possible to enter a much larger number of characters into the associated text field, the memory can overflow into other locations. An attacker can then exploit this to gain access to sensitive information or even manipulate the program.

## Stability

When a user is able to input data that the application is not equipped to handle, the application can react in unexpected ways, such as crashing or refusing to save. Here are a couple of examples:

- My ZIP code begins with a 0. I have encountered forms where I can't save my address because the application strips the leading 0 off the ZIP code and then tells me my ZIP code has only four digits.
- I have a co-worker who has both a hyphen and an apostrophe in his last name. He told me that entering his name frequently breaks the forms he is filling out.

## Visual Consistency

A field with too many characters in it can affect the way a page is displayed. This can be easily seen when looking at any test environment. For example, if a list of first names and last names is displayed on a page of contacts, you will often see that some astute tester has entered "Reallyreallyreallyreallyreallylongfirstname  
Reallyreallyreallyreallyreallylonglastname" as one of the contacts. If a name like this causes the contact page to become excessively wide and require a horizontal scroll bar, a real user who has a long name could potentially cause the page to render poorly.

## Health of the Database

When fields are not validated correctly, all kinds of erroneous data can be

saved to the database. This can affect both how the application runs and how it behaves.

The phone number field is an excellent example of how unhealthy data can affect an application. I worked for a company where phone numbers were not validated properly for years. When we were updating the application, we wanted to automatically format phone numbers so that they would display attractively in this format: (800) 555-1000. But because phone numbers had not been validated properly, people had been allowed to enter into the database values such as “dad’s office number”. There was no way to automatically format such values, and this caused an error on the page.

Painstakingly validating input fields can be very tedious, but these examples demonstrate why it is so important. The good news is that there are ways to alleviate the boredom. Automating validation checks can keep us from having to manually run the same tests repeatedly. Monkey-testing tools can help flush out bugs. And adding a sense of whimsy to testing can help keep things interesting. I have all the lyrics to “Frosty the Snowman” saved in a text file. Whenever I need to test the allowed length of a text field, I paste all or some of the lyrics into the field. When a developer sees database entries with “Frosty the Snowman was a j”, they know I have been there!

# Chapter 13: Three Ways to Test Output Validation

Output validation means checking that an application is returning the correct data and displaying it correctly. There are three main things to think about when testing outputs.

## How Is the Output Displayed?

An example of an output that you would want to check for appearance is a phone number. When a user adds a phone number to your application's data store it may be saved without any parentheses, periods, or dashes. But when you display that number to the user, you probably won't want to display it this way, because it will be hard to read. You'll want the number to be formatted in a way that the user would expect: for example, something like (800) 867-5309 for a number in the United States and 20 7373 8299 for a number in Great Britain.

Another example is a currency value. If financial calculations are made and the result is displayed to the user, you wouldn't want the result displayed as \$45.655, because no one makes payments in half-pennies. The calculation should be rounded or truncated so that there are only two digits after the decimal.

## Will the Result of a Calculation Be Saved Correctly in the Database?

Imagine you have an application that takes a value for  $x$  and a value for  $y$  from the user, adds them together, and stores them as  $z$ . The data type for  $x$ ,  $y$ , and  $z$  is set to tinyint in the database. If you're doing a calculation with small numbers, such as when  $x$  is 10 and  $y$  is 20, this won't be a problem. But what happens if  $x$  is 255—the upper limit of tinyint—and  $y$  is 1? Now your calculated value for  $z$  is 256, which is more than can be stored in the tinyint field, and you will get a server error.

Similarly, you'll want to make sure your calculation results don't go

below zero in certain situations, such as with an e-commerce app. If your user has merchandise totaling \$20 and a discount coupon for \$25, you don't want to have your calculations show that you owe them \$5!

### **Are the Values Being Calculated Correctly?**

This is especially important for complicated financial applications. Let's imagine we are testing a tax application for the Republic of Jackvonia. The Jackvonia tax brackets are simple:

| <b>Income</b>      | <b>Tax Rate</b> |
|--------------------|-----------------|
| \$0-\$25,000       | 1%              |
| \$25,001-\$50,000  | 3%              |
| \$50,001-\$75,000  | 5%              |
| \$75,001-\$100,000 | 7%              |
| \$100,001 and over | 9%              |

There is only one type of tax deduction in Jackvonia, and that is the dependents deduction:

| <b>Number of Dependents</b> | <b>Deduction</b> |
|-----------------------------|------------------|
| 1                           | \$100            |
| 2                           | \$200            |
| 3 or more                   | \$300            |

The online tax calculator for Jackvonia residents has an income field, which can contain any dollar amount from \$0 to \$1million, and a dependents field, which can contain any whole number of dependents from 0 to 10. The user enters those values and clicks the Calculate button, and then the amount of taxes owed appears.

If you were charged with testing the tax calculator, how would you test it? Here's what I would do:

1. First I would verify that a person with \$0 income and 0 dependents would owe \$0 in taxes.
  
  
  
2. Next I would verify that it was not possible to owe a negative amount of taxes; if, for example, a person made \$25,000 and had three dependents, they should owe \$0 in taxes, not -\$50.
  
  
  
3. Then I would verify that the tax rate was being applied correctly at

the boundaries of each tax bracket. So a person who made \$1 and had 0 dependents should owe \$.01, and a person who made \$25,000 and had 0 dependents should owe \$250. Similarly, a person who made \$25,001 and had 0 dependents should owe \$750.03 in taxes. I would continue that pattern through the other tax brackets, and I would include a test with \$1 million, which is the upper limit of the income field.

4. Finally, I would test the dependents calculation. I would test with one, two, and three dependents in each tax bracket and verify that the \$100, \$200, or \$300 tax deduction was being applied correctly. I would also do a test with four, five, and 10 dependents to make sure the deduction was \$300 in each case.

This is a lot of repetitive testing, so it would be a good idea to automate it. Most automation frameworks allow a test to process a grid or table of data, so you could easily test all of the preceding scenarios, and even add more scenarios for more thorough testing.

Output validation is so important because if your users can't trust your calculations, they won't use your application! Start by testing common scenarios and then design tests that verify the correct functionality, even in boundary cases.

# Chapter 14: Testing Back Buttons

The back button is so ubiquitous that it is easily overlooked in web and application testing. There are many different types of back buttons. The two major types are those that come natively with the browser or operating system and those that are added to the application by the developer.

Native buttons can be embedded in the browser, embedded in a mobile application, or included in the hardware of a mobile device. In the Chrome browser, for example, there is a back button in the toolbar in the upper-left of the screen. Android devices generally have a back button at the bottom of the screen that can be used with any application. And most iPhone apps have a back button built in at the top of every page.

Developers add back buttons when they want to have more control over the user's navigation. For example, a developer might include breadcrumbs, which display where the user has been in the workflow of an application. The user can click on a previous link to go back to the page they were on before.

When you are testing websites and applications, it's important to test the behavior of all of your back buttons, even those your team didn't add. The first thing to do is to think about exactly where you would like those buttons to direct the user. This seems obvious, but sometimes you do not want your application to go back to the previous page. For example, when a user has finished making an online purchase, they shouldn't be able to use the back button to return to the purchase page, because the transaction is already finished.

Another thing to consider is how you would like back buttons to behave when the user has logged out of the application. In this case, you don't want to be able to backtrack into the application and log the user in again, because this would cause a security concern: if the user was on a public computer, the next user would be able to access the previous user's account.

For back buttons on mobile devices, think about what behavior you

would like the button to have when you are in your application. A user will be frustrated if they are expecting the back button to take them elsewhere in your app, but instead it takes them out of the app entirely.

If your application has a number of added back buttons, be sure to follow them all in the largest chain you can create. Look for errors in logic where the application takes you to someplace you weren't expecting or for memory errors caused by saving too many pages in the application's path.

You can also check whether the back button is enabled and disabled at appropriate times. You don't want your user trying to click on an enabled back button that doesn't go anywhere!

In summary, whenever you are testing a web page or an application, be sure to note all the back buttons that are available and all the behaviors those buttons should have. Then test those behaviors to make sure your users will have a positive and helpful experience.

# Chapter 15: Six Tips for File Upload Testing

Testing file uploads is important because uploading a malicious file is one of the ways a bad actor can exploit your application, either by taking it down or by extracting sensitive data from it. In this chapter, I offer six tips to help you be successful with testing file uploads.

## **Tip #1: Upload Files with Allowed and Forbidden Extensions**

The first step in testing file uploads is to find out what kinds of files can be uploaded. These file types should be compiled into an allowlist, which would specify the extensions that are allowed. They should not be compiled into a denylist, which would specify the extensions that are not allowed. As you can imagine, when a denylist is used, dozens and dozens of file types will be allowed, some of which you will not want in your application! Therefore, it's important to use an allowlist instead, which will be limited to the very few file types that you want interacting with your application. If your developers are not using an allowlist, please share this information with them.

Once you know what the allowlisted file types are, try uploading each type. Then try uploading a wide variety of files that are not allowlisted. Each of those files should be rejected with an appropriate error message for the user.

## **Tip #2: Upload Files with Inaccurate Extensions**

One of the tricks malicious users employ to upload forbidden files is to rename a malicious file with an allowed extension. For example, a bad actor could take a .js file and rename it as a .jpg file. If .jpg files are allowed in your application, the file might be uploaded and then executed when it's opened by an unsuspecting user. So it's important for your application to have checks in place that not only verify the extension, but also scan the file to verify its type.

It's easy to test this. Simply rename a forbidden file with an allowed

extension and try to upload the file. The file should be rejected with an appropriate error message. The attempt should also be logged by the application so that if there is ever an upload attempt of this kind in production, your security team can be alerted.

### **Tip #3: Test for Maximum File Size**

Your application should specify a maximum file size. Files that are too big can cause damage to your application, either by slowing it down or by causing it to crash. They can even cause data to be accidentally exposed, such as in a buffer overflow exploit.

Find out what your application's maximum file size is, and verify that files equal to and smaller than that size are uploaded appropriately. Then verify that files over that maximum size are rejected with an appropriate error message. Be sure to test with files just over the maximum size as well as with files well over the maximum size.

### **Tip #4: Test with Animated GIFs**

Often, when image uploads are allowed in an application, the .gif extension is one of the allowed types. GIFs can sometimes contain animation. Verify with your team whether your application will allow animated GIFs, and if not, verify what should happen if a user uploads one. Will the file display as a static image, or will the file be rejected? Make sure that uploading an animated GIF does not result in a broken image on the page. If animated GIFs are accepted, verify that they load and display the animations properly.

### **Tip #5: Verify That the File Was Uploaded Correctly**

It's not enough to verify that you don't get an error message when you upload an allowlisted file. You also need to verify that the file was saved to the database correctly. The easiest way to do this is to download the file and make sure it looks the same way it did when you uploaded it. If your file should be displayed in the UI, you should make sure the file looks correct in a browser and on a mobile device. If an image that you uploaded should be resized on the page, make sure it has resized correctly. You don't want to

have other data obscured because someone uploaded an image that's too large! If you are expecting a video or audio file to play, make sure it's playable.

### **Tip #6: Have a Folder with File Examples for Testing**

This is my favorite tip. I have a folder containing dozens of files with different extensions and sizes. Whenever I need to test file uploads, I'm ready to go with test files and I don't have to waste time combing the internet for good examples to use.

# Chapter 16: Testing the Login Screen

The login screen is the first line of defense between your application and a malicious unauthorized user. But it's so easy to forget about testing the login screen! This is because as a tester, you are in the application every single day. Getting past the login screen is a step you take dozens of times a day to get to whatever feature you are currently testing.

Let's take some time to look at the various ways we should be testing login screens.

First, make sure the password field masks text while you're typing in it. Also ensure that both the username and password fields are expecting case-sensitive text. In other words, if your username is "FOO" and your password is "bar", you should not be able to log in with "Foo" and "Bar". If the username is an email address, however, it usually does not expect case-sensitive text.

Next, find out what the validation requirements are for the username and password fields. How short can the values be? How long can they be? What characters will be accepted? Verify that the length requirements are respected in both fields, then test with every nonaccepted character that you can think of. Be sure to test with non-UTF-8 characters and emojis as well. Several years ago there was a story circulating the internet about a bank customer who broke the application by setting her password to have an emoji in it!

Also be sure to test for SQL injection. For example, if you enter `1 or 1 = 1` into the password field, you may be telling the database to return "true" and allow the login because, of course, 1 always equals 1.

Now let's test with every possible combination of username and password state. Each field can be either empty, incorrect, correct but with the wrong case, or correct. This generates 16 possible combinations:

| <b>Username</b> | <b>Password</b> | <b>Username</b> | <b>Password</b> |
|-----------------|-----------------|-----------------|-----------------|
| Empty           | Empty           | Wrong Case      | Empty           |
| Empty           | Incorrect       | Wrong Case      | Incorrect       |
| Empty           | Wrong Case      | Wrong Case      | Wrong Case      |
| Empty           | Correct         | Wrong Case      | Correct         |
| Incorrect       | Empty           | Correct         | Empty           |
| Incorrect       | Incorrect       | Correct         | Incorrect       |
| Incorrect       | Wrong Case      | Correct         | Wrong Case      |
| Incorrect       | Correct         | Correct         | Correct         |

It may seem like overkill to test all these combinations, but I have tested applications where I could log in with an empty username and password and with an incorrect username and password!

Another helpful thing to test is putting an empty character or two at the beginning or end of the fields. When the user submits their login request, empty characters should be stripped from these values. If this does not happen, the user may wind up in a situation where they have typed their correct password, “bar ”, but the password is not accepted because of the space at the end. This is very frustrating for an end user who doesn’t know the empty character is there.

Next, let’s think about the kind of error message you are getting when you try to log in with invalid credentials. You don’t want to give a malicious user any hints about whether they have guessed a username or password correctly. If the malicious user types in “FOO” for the username and “password” for the password, you don’t want to return a message that says “Invalid password”, because this will let the malicious user know the username is correct! Now all they have to do is keep the username the same and start guessing at the password. It is much better to return a message like “Invalid credentials”.

Now let’s take a look at what is passed into the server when you are making a login request. Open the developer tools for the browser you are

using and watch the request that goes through. Do you see the username or the password in clear text anywhere? If you do, this should be fixed! Usernames and passwords should be encrypted in the request. You can also check the request by using an interception tool like Fiddler or Burp Suite. (More on intercepting requests in Part VI.)

Once you have logged in with the correct credentials, take a look at the cookie or session ID that has been set for your user. Do you see the username and password anywhere? All cookies, web tokens, and session IDs should not include the user credentials, even if they are encrypted. (More on cookies and session IDs in Part III.)

Finally, be sure to test the procedure for logging out. When you log out, the username and password fields on the login screen should be cleared unless there is a feature to remember the credentials. If there is such a feature, the password should be masked and should not indicate how many characters are in it. Upon logging out, any session IDs or web tokens should be deleted. If there is no feature to remember credentials, the cookie should be deleted as well. There should be nothing that you can grab and use, from developer tools or interception tools, to pretend that you are logged in.

This is a lot to test; fortunately, we can take advantage of automation for regression testing once we have done an initial test pass. Here are a few ideas for what to automate (more on test automation in Part X):

- API login calls using all the combinations of username and password listed in this chapter (more on APIs in Part IV)
- UI logins with too many characters and invalid characters in the username and password fields; verify that the correct error message is returned
- Visual testing of the login screen to verify that the password is not displayed in clear text

Security is such an important part of every application. By running through all of these login scenarios, you can help bring peace of mind to your product team and to your end users.

# Chapter 17: Cross-Browser Testing

Browser parity is much better today than it was just a few years ago, but every now and then you will still encounter differences in how your application performs in different browsers. Here are just a few examples of discrepancies I've encountered over the years:

- A page that scrolls just fine in one browser doesn't scroll at all in another, or the scrolling component doesn't appear.
- A button that works correctly in one browser doesn't work in another.
- An image that displays in one browser doesn't display in another.
- A page that automatically refreshes in one browser doesn't do so in another, leaving the user feeling as though their data hasn't been updated.

Following are some helpful hints to make sure your application is tested in multiple browsers.

## **Know Which Browser Is the Most Popular with Your Users**

Several years ago I was testing a business-to-business CRM-style application. Our team's developers tended to use Chrome for checking their work, and because of this I primarily tested in Chrome as well. Then I found out that more than 90% of our end users were using our application in Internet Explorer 9. This definitely changed the focus of my testing! From then on, I made sure every new feature was tested in IE 9 and that a full regression pass was run in IE 9 whenever we had a release.

Find out which browsers are the most popular with your users and be sure to test every feature with them. This doesn't mean you have to do the bulk of your testing there; but with every new feature and every new release you should validate all the UI components in the most popular browsers.

## **Resize Your Browsers**

Sometimes a browser issue isn't readily apparent, because it only appears

when the browser is using a smaller window. As professional testers, we are often fortunate to be issued large monitors on which to test. This is great because it allows us to have multiple windows open and view one whole web page at a time, but it often means we miss bugs that end users will see.

End users are likely not using a big monitor when they are using our software. Issues can crop up, such as a vertical or horizontal scroll bar not appearing or not functioning properly; text not resizing, so it ends up going off the page and is no longer visible; or images not appearing or taking too much space on the page.

Be sure to build page resizing into every test plan for every new feature, and build it into a regression suite as well. Find out what the minimum supported window size should be and test all the way down to that level, with variations in both horizontal and vertical sizes.

### **Assign Each Browser to a Different Tester**

When doing manual regression testing, an easy way to make sure all browsers you want to test are covered is to assign each tester a different browser. For example, if you have two other testers on your team, you could run your regression suite in Chrome and Safari, another tester could run the suite in Firefox, and the third tester could run the suite in Edge. The next time the suite is run, you can swap browsers so that each browser will have a fresh set of eyes on it.

### **Watch for Changes After Browser Updates**

It's possible for something that worked great in a browser to suddenly stop working correctly when a new version of the browser is released. It's also possible that a feature that looks great in the latest version of the browser doesn't work in an older version. Many browsers, including Chrome and Firefox, are set to automatically update themselves with every release, but some end users may have turned this feature off, so you can't assume that everyone is using the latest version. It's often helpful if you have a spare testing machine to keep browsers installed with the next-to-last release. This way, you can identify any discrepancies that may appear between the old and new browser versions.

Browser differences can greatly impact the user experience. If you build in manual and automated systems to check for discrepancies, you can easily ensure a better user experience with minimal extra work.

# Chapter 18: Before You Log That Bug...

In “The Boy Who Cried Wolf,” one of Aesop’s Fables, a shepherd boy repeatedly tricks the people in his village by crying out that a wolf is about to eat his sheep. The villagers run to his aid, only to discover that he is playing a prank. One day, the boy really does see a wolf. He cries for help, but none of the villagers come, because they are convinced he is playing another trick.

We do not want to be “The Testers Who Cried Bug.” If we repeatedly report bugs that are actually caused by user error, our developers won’t believe us when we really find a bug. To keep this from happening, let’s take a look at the things we should check before we report a bug. These tasks fall into two categories: checking for user error and gathering information for developers.

## Check for User Error

Has the code you are testing been deployed to your test environment? When I was a new tester, I constantly made the mistake of testing before I determined that the deployment had completed successfully. It’s such a waste of time to keep investigating an issue with code when the problem is actually that the code isn’t there!

Are you testing in the right environment? When you have multiple environments to test in and they all look similar, it’s easy to think you are testing in one environment when you are actually in another. Take a quick peek at your URL if you are testing a web app, or at your build number if you are testing a mobile app.

Do you understand the feature? In a perfect world, we would all have great documentation and well-written acceptance criteria. In the real world, this often isn’t the case. Check with your developer and product owner to make sure you understand exactly how the feature is supposed to behave. Maybe you misunderstood something when you started to test.

Have you configured the test correctly? Maybe the feature only works

when certain settings are enabled. Think about what those settings are and go back and check them.

Are you testing with the right user? Maybe this feature is only available to admin users or paid users. Verify the criteria of the feature and check your user.

Does the backend data support the test? Let's say you are testing that a customer's information is displayed. You expect to see the customer's email address on the page, but the email is not there. Maybe the problem is actually that the email address is null, and that is why it is not displaying.

If you have checked all of these things and you still see an issue, it's time to think about reporting the bug. But before you do, consider the questions the developer might ask you when they begin to investigate the issue. It will save time for both of you if you have all of these questions answered ahead of time.

## **Gather Information for the Developer**

Are you able to reproduce the issue? You should be able to reproduce it at least once before logging the bug. This doesn't mean you shouldn't log intermittent issues, as they are important as well; it means you should have as much information as possible about when the issue does and doesn't occur.

Do you have clear, reproducible steps to demonstrate the issue? It is incredibly frustrating to a developer to hear that something is wrong with their software and to receive only vague instructions to use for investigation. For best results, give the developer a specific user, login credentials, and clear steps they can use to reproduce the problem.

Is this issue happening in production? Maybe this isn't a new bug; maybe the issue was happening already. This is especially possible when you are testing old code that no one has looked at or used in a while.

Does the issue happen on every browser? This information can be very helpful in narrowing down the possible cause of an issue.

Does the issue happen with more than one user? It's possible that the user you are testing with has some kind of weird edge case in their configuration or their data. This doesn't mean the issue you are seeing isn't a bug; if you can show there are some users for whom the issue is not happening, it will help narrow the scope of the problem.

Does the issue happen if the data is different? Try varying the data and see whether the issue goes away. Maybe the problem is caused by a data point that is larger than what the UI is expecting or by a field that is missing a value. The more narrowly you can pinpoint the problem, the faster the developer can fix it.

The ideal relationship between a tester and a developer is one of mutual trust. If you make sure to investigate each issue carefully before reporting it, and if you are able to report issues with lots of helpful details, your developer will trust that when you cry "Bug," it's something worth investigating!

# Chapter 19: How to Reproduce a Bug

Often in software testing we'll encounter a bug that we see only once or intermittently, and we can't figure out how to reproduce it. In this chapter I provide some helpful hints for reproducing bugs and getting to the root cause of issues.

## Gather Information

The first thing to do when you have a bug to reproduce is to gather as much information as you can about the circumstances of the issue. If it's a bug that you just noticed, think about the steps you took before the bug appeared. If it's a bug that someone else has reported, find out what they remember about the steps they took, and ask for details such as their operating system, browser type, and browser version.

## One Step at a Time

Next, see whether you can follow the steps that were taken by you or the person who reported the bug. If you can reproduce the bug right away, you're in luck! If not, try changing your steps one at a time and see whether the bug appears. Vary things like the text you entered into the field or the button you clicked to submit a form. Don't just thrash around trying to reproduce the issue quickly; if you're making lots of disorganized changes, you might reproduce the bug and not know how you did it. Keep track of the changes you made so that you know what you've tried and what you haven't tried yet.

## Logs and Developer Tools

Application logs and browser developer tools can provide helpful clues about what is going on behind the scenes in an application. A browser's developer tools can generally be accessed in the menu found in the upper-right corner of the browser; in Chrome, for example, you would click on the three-dot menu, then choose More Tools, then Developer Tools. This will open a window at the bottom or side of the page where you can find information such as errors logged in the console or what network requests

were made.

## **Factors to Consider When Trying to Reproduce a Bug**

When you're trying to reproduce a bug, it can be helpful to consider the following factors:

**User:** What user was utilized when the bug was seen? Did the user have a specific permission level? You may be dealing with a bug that is only seen by administrators or by a certain type of customer.

**Authentication:** Was the user logged in? The bug may appear only when the user is not authenticated or only when the user is authenticated.

**State of the data:** What kind of data does the user have? Can you try reproducing the bug with exactly the same data? The bug might only appear when a user has a very long last name or a particular type of image file.

**Configuration issues:** Something in the application may be set up incorrectly. For example, a user who isn't getting an email notification might not be getting it because email functionality is turned off for their account. Check all the configurable settings in the application and try to reproduce the issue with exactly the same configuration.

**Actions taken before the issue occurred:** Sometimes bugs are caused not by the current state where they are seen, but by some event that happened before the current state. For example, if a user started an action that used a lot of memory, such as downloading a very large file, and then continued on to other activities while the file was downloading, an error caused by running out of memory might affect their current activity.

**Back button:** The back button can be the culprit in all kinds of mysterious bugs. If a user navigates to a page through the back button, the state of the data on the page might be different from what it would be through standard forward navigation.

**Caching:** Caching can result in unexpected behavior as well. For example, it might appear as though data is unchanged when in fact it has been

changed. If a cache never expires or takes too long to expire, the state of the data can be very different from what is displayed.

**Race conditions:** These issues are very difficult to pin down. Stepping through the code probably won't help, because when the program is run one step at a time the problem doesn't occur. The best way to determine whether there is a race condition is to run your tests several times and document the inconsistent behavior. You can also try clicking on buttons or links before a page has loaded to speed up input, or throttling your internet connection to slow down input.

**Browser/browser version:** Browsers are definitely more consistent in their behavior than they used to be, and most browsers are now updated to the latest version automatically, but it's still possible to find bugs that only appear in certain browsers or versions. If your end user is using IE 8 on an old Windows desktop, for example, it's highly likely that your application will behave differently for them.

**Browser size:** If a customer is saying they don't see a Save button or a scroll bar in their browser window, ask them to run a browser size tool in another tab on their browser. Set your browser to be the same size as theirs and see whether you now have the same problem.

**Machine or device:** Mobile devices are highly variable, so it's possible that a user is seeing an issue on an Android device that you are not seeing on an iOS device, or that a user is seeing a problem on a Motorola device that you are not seeing on a Samsung device. Laptops and desktop computers are less variable, but it is still possible that a Mac owner is experiencing an issue that you don't see in Windows. Mobile testing sites that allow you to test on their library of devices can be helpful for diagnosing an issue on a machine or device that you don't own.

## Once You've Reproduced the Bug

Once you've finally reproduced a tricky bug, you might want to just show it to your developer and be done with it. But your work is not done! To make sure the bug is fixed correctly, you'll want to narrow down the steps to reproduce it as much as you can so that you can be as precise as possible. The

more precise you are, the easier it will be for the developer to locate the true cause in the code and fix it.

For example, if you were able to reproduce a bug by using an admin user who navigated to the page with the back button on a Firefox browser, are you sure you need all three of those conditions to see the bug? Do you see the bug when you use an admin user and the back button in Chrome? If you do, you can eliminate the browser from your bug report. Similarly, if you see the bug when you are using a non-admin user, you can take that out of the bug report as well. Now you have narrowed down the issue to just the back button, giving the developer a clear indication of where to start with a fix.

As a software tester, I am often annoyed by bugs and poor user experiences I encounter in my daily life. I frequently ask myself: “Who is testing this website? What were they thinking? Does this company even have testers?” We are the last line of defense to keep our customers from being frustrated, which is why it is important to chase down those elusive bugs.

# Chapter 20: How to Log a Bug

In the previous two chapters, we looked at how to make sure a bug is really a bug and the strategies to employ to reproduce it. Once you are sure you have a bug and you know how to reproduce it, it's time to log it. But just throwing a few sentences in your team's bug-tracking system is not a good idea! The way you log a bug and the details you include can determine whether the bug will be prioritized or left on the backlog, and whether a developer is able to find the problem or closes the bug out with a "cannot reproduce" message.

Let's imagine we have an application called the Super Ball Sorter that sorts out Super Balls among four children according to a series of color and size rules set up for each child. When you test the feature, you discover that if three of the children have a rule stating they accept only large balls of some color, the small purple ball is never sorted, even though the fourth child without a rule should accept the ball. We'll use this example bug to show what to do and what not to do when logging a bug.

Here are the components of a well-logged bug:

**Title:** The title of the bug should begin with the area of the application it is referring to. For example, if a bug was found in the Contacts section of your application, you could begin the title with "Contacts". In this case, the area of the application is the Super Ball Sorter. After the application area, you should continue the title with a concise statement of the problem.

- **RIGHT**—Super Ball Sorter: Small purple ball is not sorted when three children have large-ball rules
- **WRONG**—Small purple ball not sorted

While the wrong example gives at least a clue as to what the bug is about, it will be hard to find among dozens of other bugs later, when you might try to search using the term "Super Ball". Moreover, it doesn't state what the conditions are when the ball is not sorted, so if another bug is found later in

which this same ball isn't sorted, there could be confusion as to which bug is which.

**Description:** The first sentence of the bug should describe the issue. This sentence can provide a bit more detail than the title. I often start this sentence with “When”, as in “When I am testing x, y happens.”

- **RIGHT**—When three children have sorting rules involving large balls, the small purple ball is not sorted.
- **WRONG**—Doug doesn’t get the small purple ball.

A number of things are wrong with the second example sentence. First, the issue happens regardless of which of the three children has a rule stating they get only large balls, so referring to Doug here could be misleading. Second, the statement doesn’t describe what rules have been set up. A developer could read this sentence and assume the small purple ball is never sorted, regardless of what rules are set up.

**Environment and browser details:** These can be optional if it’s assumed that the issue was found in the QA environment and occurs on all browsers. But if there’s any chance that the developer won’t know what environment you are referring to, be sure to include it. And if the issue is found on one browser but not others, be sure to mention that detail.

- **RIGHT**—This is happening in the staging environment, on the Edge browser only.

**Steps to reproduce:** The steps should include any configuration and login information, and clearly defined procedures to display the problem.

- **RIGHT:**

1. Log in with the following user:
2. username: foobar
3. password: mebs47
4. Assign Amy a rule where she only gets large red balls. Assign Bob a rule where he only gets large orange balls. Assign Carol a rule

- where she only gets large yellow balls.
5. Create a set of Super Balls to be sorted, and ensure that there is at least one small purple ball.
  6. Run the Super Ball Sorter.
  7. Check each child's collection for the small purple ball or balls.

- **WRONG:**

Everyone has a rule except Doug, and no one is getting the small purple ball.

The preceding example doesn't provide nearly enough information. The developer won't know the user's login credentials or what the three rules should be for large balls.

- **ALSO WRONG:**

1. Open the application.
2. Type "foobar" in the username field.
3. Type "mebs47" in the password field.
4. Click the login button.
5. Go to the Super Ball Sorter rules page.
6. Click on Amy's name.
7. Click on the large ball button.
8. Click on the red ball button.
9. Click the Save button.
10. Click on Bob's name.
- etc. etc. etc.

These steps provide *way* too much information. It's safe to assume the developer knows how to log in; simply providing the credentials should be enough. It's also safe to assume the developer knows how to set a rule in the Super Ball Sorter, since they wrote the code.

**Expected and actual result:** State what behavior you were expecting and what behavior you got instead. This can help prevent misunderstandings, and it's also helpful when a bug has been sitting on the backlog for months and you've forgotten how the feature is supposed to work.

- RIGHT

Expected result: Doug gets the small purple ball because he is the only child configured to accept it.

Actual result: Doug does not get the small purple ball, and the ball remains unsorted.

**Screenshot or stack trace:** Include this information only if it will be helpful.

- RIGHT

Exception: Ball was not recognized

Caused by: Ball.sort(Ball.java:11)

- WRONG

Exception: An unknown error occurred

- ALSO WRONG:

# Doug



Large Green  
Small Yellow  
Small Red

The exception “An unknown error occurred” doesn’t tell the developer anything, so there’s no reason to include the message. And the screenshot is not particularly helpful. While it shows that Doug doesn’t have a small purple ball, this is easily conveyed by the Actual Result that was already described; no picture is needed.

A clearly written bug is helpful to everyone: the product owner, who prioritizes bug fixes; the developer, who has to figure out what’s wrong and fix it; and the tester, who will need to retest the issue once it’s fixed. When you take care to log bugs well, it prevents frustration and saves everyone time.

## Part III: How Applications Work

# Chapter 21: How HTTP Requests Work

Recently I read a great [post](#) about the importance of having technical skills as a software tester. The author makes an excellent analogy: a software tester who doesn't understand technical concepts is like a surgeon who doesn't understand anatomy. If we are going to test our applications thoroughly, we should understand the underlying systems that make them work. We'll begin by learning about HTTP requests.

When you type a website's address into a web browser, you are typing a URL. URL (uniform resource locator) is simply a fancy name for a web address. The URL contains a domain name. The domain name identifies a specific grouping of servers on the internet; examples of domain names are google.com and amazon.com.

Once the browser has the domain name, it uses it to look up the associated IP address in the DNS (Domain Name System), which is a database that contains all the mappings of domain names and IP addresses. An IP address (Internet Protocol address) is a unique series of numbers that is assigned to every device connected to the internet.

Once the IP address is known, a connection is opened to that address using HTTP. HTTP (HyperText Transfer Protocol) is an application protocol that allows information to be transmitted electronically.

When the connection is opened to the server at the IP address, a request can be made to get information from that server. Information sent over the internet is called a message. The request uses TCP (Transmission Control Protocol), which is a system of delivering messages over the internet.

TCP divides a message into a series of packets, which are fragments of between 1,000 and 3,000 characters. Each packet has a series of headers that include the address of the packet's destination, information about the ordering of the packets, and other important information.

If, for any reason, a packet doesn't make it to its destination, the client

(the address making the request) can request that the packet be re-sent. Once all the packets have arrived, the client reassembles them according to the instructions in the header.

How does data know how to get from one IP address to another? I'll address this in the next chapter.

# Chapter 22: Internet Routing

Nearly every home has an internet connection, but very few people know how the router that allows the connection works. Every device that can be connected to the internet has a network interface card (NIC), which is a piece of hardware installed on the device that allows it to make a network connection. Each NIC has a media access control (MAC) address that is specific to only that device.

A modem is a device that connects to the internet. Other devices can connect to the modem to receive internet transmissions. A wireless router is capable of receiving Wi-Fi transmissions. The router connects to the modem, and other devices connect wirelessly to the router to receive data from the internet. Many internet service providers (ISPs) provide customers with a combination modem/router, which connects to the internet and sends and receives wireless signals.

In the preceding chapter, you learned that every device connected to the internet has an IP address. An IP address is different from a MAC address in that a MAC address is assigned by the manufacturer of the device and an IP address is assigned by the network. An IP address has two sections, called the subnet and the host. The subnet refers to one subsection of the entire internet. The host is the unique identifier for the device on the network. The IP address combines these two numeric values using bitwise operations. You can't look at an IP address and say that the numbers on the left make up the subnet and the numbers on the right make up the host; the IP address is more like a sum of two long numbers.

When a packet of data is sent from a server to a client, it is sent with the destination's IP address. To get to that destination, the packet will hop from one network to another. The routing protocol of the internet is called Border Gateway Protocol (BGP). This is a system that helps determine a route which will traverse the smallest number of networks to reach the destination. Every router in a network has a series of routing tables, which are sets of directions for how to get from one network to another.

When a packet of information is first sent to the network's router, it looks at the destination's IP address and determines whether the directions to the destination are available in the routing tables. If they are not, BGP is used to determine the next logical network where the packet should be sent. A gateway is an entrance to a network, and a default gateway is the address the request is sent to if there's no knowledge of a specific address in that network. When a packet arrives at the new gateway, BGP calculates the next appropriate destination.

After traversing through networks in this way, eventually the packet arrives at the router for the network that contains the destination's IP address. The router determines the destination's MAC address and sends the packet to that address.

One more important feature of networking is the use of a proxy server, which is a server that is positioned between the client and destination servers. It is configured so that any requests your client makes will go through it before the requests reach their destination. There are many uses for a proxy server; the main use is to keep the actual address of a site or router private. Proxy servers can also be used by hackers to intercept requests, especially on a public network.

Using proxy servers is a great way to do security testing! With a tool like Fiddler or Burp Suite, you can intercept the requests you make to your application and the responses you get in return. You'll learn more about security testing in Part VI.

# Chapter 23: Encoding and Encryption

Encoding and encryption are two very different methods. As a software tester, it's important to understand how each method works.

Encoding simply means transforming data into a form that's easier to transfer. URL encoding is a simple type of encoding. For example, in URLs the spaces are encoded using %20. Other symbols, such as !, are replaced in URL encoding as well. If you'd like to learn more about URL encoding, you can play around with an encoding/decoding tool on the Web.

Another common type of encoding is Base64 encoding. Base64 encoding is often used to send data; the encoding keeps the bytes from getting corrupted. This type of encoding is also used in Basic authentication. You may have seen a username and password encoded in this way when you logged in to a website. It's important to know that Basic authentication is not secure! Let's say a malicious actor has intercepted my login with Basic auth, and they've grabbed the authentication string: a2phY2t2b255OnBhc3N3b3JkMTIz. That looks pretty secure, right? Nope! All the hacker needs to do is go to a website that decodes Base64 encoding and decode my username and password. Try it for yourself!

Encryption transforms data to keep it secret. A common method of password encryption is hashing, which is a mathematical way of encrypting that is impossible to decrypt. This seems puzzling: if a string is impossible to decrypt, how will an application ever know that a user's password is correct? What happens is that the hashed password is saved in the application's authentication database. When a user logs in, their submitted password is encrypted with the same hashing algorithm that was used to store the password. If the hashed passwords match, the password is correct.

What if two users have the same password? If a user somehow was able to access the authentication database to view the hashed passwords and saw that another user had the same hashed password as they did, that user would now know someone else's password. We solve this problem through salting.

A salt is a short string that is added to the end of a user's password before it is encrypted. Each password has a different salt added to it, and that salt is saved in the database along with the hashed password. This way, if a hacker gets the list of stored passwords, they won't be able to find any two that are the same.

A common hashing algorithm is SHA256. SHA stands for Secure Hash Algorithm. The 256 value refers to the number of bits used in the encoding.

Other types of encryption can be decoded as well. Two examples are AES encryption and RSA encryption. AES stands for Advanced Encryption Standard. This type of encryption is called symmetric key encryption. In symmetric key encryption, the data is encoded with a key, and the receiver of the data needs to have the same key to decrypt the data. AES encryption is commonly used to transfer data over a VPN.

RSA stands for Rivest-Shamir-Adleman, who are the three inventors of this encryption method. RSA uses asymmetric encryption, also called public key encryption, in which a public key is used to encode the data and a private key is used to decode it. This can work in a couple of ways. If the sender of the message knows the receiver's public key, they can encrypt the message and send it; then the receiver can decrypt the message with the private key. Or the sender of the message can sign the message with their private key, and then the receiver of the message can decode it with the sender's public key.

In the second example, the private key is used to show that the message is authentic. How does the receiver know the message is authentic if they don't know what the private key is? They know because if the private key is tampered with, it will be flagged to show that it has been manipulated. A very common use of RSA encryption is TLS, which is what is used to send data to and from websites.

Encryption involves very complicated mathematical algorithms. Fortunately, we don't have to learn them to understand how encryption works!

# Chapter 24: HTTPS, Tokens, and Cookies

When information is passed back and forth between systems, we need to make sure it can't be intercepted by others for whom it was not intended. That's why HTTPS was created. In this chapter, we'll talk about how encryption is used in HTTPS, the difference between cookies and tokens, the different types of cookies, and how cookies can be protected.

## How HTTPS Works

When two systems communicate with each other, we refer to them as the client and the server. The client is the system making the request, such as a browser, an application, or a mobile device, and the server is the system that supplies the information, such as a datastore. HTTPS is a method of securely transmitting information between the client and the server. HTTPS uses SSL (Secure Sockets Layer) and TLS (Transport Layer Security) to encrypt the data being transmitted and decrypt it only when it arrives at its destination.

TLS is a newer version of SSL. Here's how it works. Before any data is transmitted, the client and server perform a handshake. The handshake begins with the client contacting the server with a suggested encryption method and the server responding that it agrees to use that encryption method. It then continues with the client and the server swapping certificates. A certificate is like an ID card; it verifies the identity of the client or server. The certificates are checked against the CA (Certificate Authority), which is a third-party entity that creates certificates specifically for the purpose of HTTPS communication.

Once the certificates are swapped and verified, the client and the server swap decryption keys and the handshake is completed. Now the data is encrypted, transmitted from the server to the client, and decrypted once it arrives at the client safely.

## Session Cookies and Tokens

Another important way data is secured is through the use of session

cookies and tokens. Session cookies and tokens are strings that are passed in with a client's request to verify that the person making the request has the right to see the data requested. The main difference between session cookies and tokens is that a session cookie is stored on the client and the server, and a token is only stored on the client.

In systems that use tokens, the token is created when a user logs in. The token is made up of encrypted information that identifies the user. It is stored in local storage in the client's browser and is sent with every HTTPS request to the server. When the server receives the token, it decrypts it, validates the user's information, and then returns the response.

The most popular system of tokens in use today is JWT (JSON Web Token). You'll learn more about JWTs in the next chapter.

A session cookie is a unique string that the server creates when the user logs in. It is saved in the server's datastore as a session ID. The server returns the cookie to the client, and the client saves it in the browser while the session is active. Whenever the client makes a request to the server, it sends the cookie with the request. The server then compares the cookie with the one it has saved to make sure they match before returning the response.

Tokens and session cookies are usually set to expire after a period of time or after an event. For example, an issued token might be good for one hour. Just before the hour is up, a request can be made for a new token (called a refresh token) to keep the user signed in. Session cookies usually expire when the user logs out or when the browser is closed.

## Persistent Cookies

Another type of cookie used is the persistent cookie, which is a bit of data saved on the server about the user's preferences. For example, if a user goes to a website and chooses German as the language they would like on the site, a persistent cookie will remember that information. The next time the user goes to the site, the cookie will be examined and the site will load in German.

## Securing Cookies

Because they are stored on the server, cookies are more vulnerable than tokens to being intercepted and used by someone other than the user. To help protect cookies, the following flags (attributes) can be added to them at the time of creation:

- **Secure flag:** Ensures that the cookie can only be transmitted over HTTPS requests and not over HTTP requests
- **HttpOnly flag:** Keeps a cookie from being accessed via JavaScript, which helps protect it from cross-site scripting (XSS) attacks.
- **SameSite flag:** Ensures that the cookie can only be sent from the original client that requested the cookie, which helps protect it from Cross-Site Request Forgery (CSRF) attacks.

# Chapter 25: The Joy of JWTs

If you have ever tested anything with authentication or authorization, chances are you used a JWT. The term “JWT” is pronounced “jot” and it stands for JSON Web Token. JWTs are created by a company called Auth0, and their purpose is to provide a method for an application to determine whether a user has the credentials necessary to request an asset.

JWTs are useful because they allow an application to check for authorization without passing in a username and password or a cookie. Requests of all kinds can be intercepted by a malicious user, but a JWT contains nonsensitive data and is encrypted, so intercepting it doesn’t provide much useful information.

A JWT has three sections, which are made up of a series of letters and numbers and are separated by periods. One of the best ways to learn about JWTs is to practice using the official JWT Debugger, so go to <https://jwt.io> and scroll down until you see the Debugger section.

## Section One: The Header

The header lists the algorithm that is used for encrypting the JWT, and also lists the token type (which is JWT, of course):

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

## Section Two: The Payload

The payload lists the claims that the user has. There are three types of claims:

- **Registered claims:** These are standard claims that are predefined by the JWT code. They include the following:

iss (issuer): Who is issuing the claim  
iat (issued at): What time, in Epoch time, the claim was issued  
exp (expiration time): What time, in Epoch time, the claim will expire  
aud (audience): The recipient of the token  
sub (subject): What kinds of things the recipient can ask for

- **Public claims:** These are other frequently used claims, and they are added to the JWT registry. Some examples are name, email, and timezone.
- **Private claims:** These are claims that are defined by the creators of an application, and they are specific to that company. For example, a company might assign a specific user ID to each of its users, and that could be included as a claim.

Here's an example of claims used in the JWT Debugger:

```
{  
  "sub": "1234567890",  
  "name": "John Doe",  
  "iat": 1516239022  
}
```

Here the subject is 1234567890, the name of the user who has access to the subject is John Doe, and the token was issued at 1516239022 Epoch time.

### Section Three: The Signature

The signature takes the first two sections and encodes them in Base64. Then it takes those encoded sections and adds a secret key, which is a long string of letters and numbers. Finally, it encrypts the entire thing with the HMAC SHA256 algorithm.

### Putting It All Together

The JWT is made up of the encoded header, then a period, the encoded payload, then another period, and finally the encrypted signature. The JWT

Debugger helpfully color-codes these three sections so that you can distinguish them.

If you use JWTs regularly in the software you test, try taking one and putting it in the JWT Debugger. The decoded payload will give you insight into how your application works.

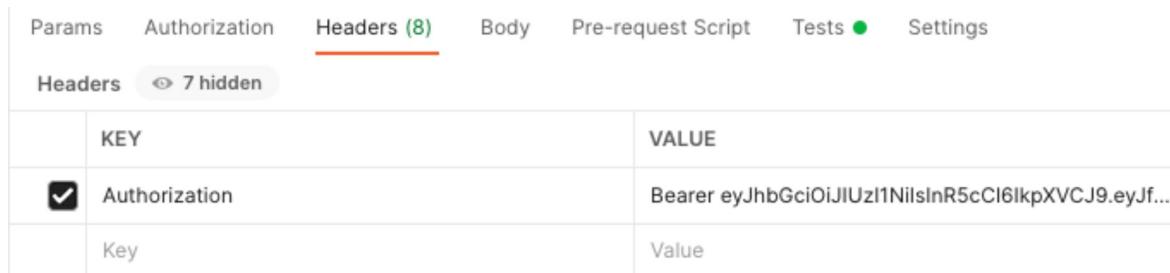
If you don't have a JWT to decode, try making your own! You can paste values like this into the Payload section of the Debugger and see how the encrypted JWT changes:

```
{  
  "sub": "userData",  
  "userName": "kjackvony",  
  "iss": 1516239022,  
  "exp": 1586606340  
}
```

When you decode a real JWT, the signature doesn't decrypt. That's because the secret used is a secret! But because the first and second sections of the JWT are encoded rather than encrypted, they can be decoded.

## Using JWTs

How JWTs are used will vary, but a common usage is to pass them with an API request using a Bearer token. In Postman, it will look something like this:



The screenshot shows the Postman interface with the 'Headers' tab selected. There are 8 headers listed. One header, 'Authorization', is explicitly defined with a value of 'Bearer eyJhbGciOiJIUzI1NilsInR5cCI6IkpXVCJ9.eyJ...'. Another header, 'Key', is listed without a value.

| KEY           | VALUE  |
|---------------|--|
| Authorization | Bearer eyJhbGciOiJIUzI1NilsInR5cCI6IkpXVCJ9.eyJ... |
| Key           | Value  |

## Testing JWTs

Now that you know how JWTs work, you can test them in the following

ways:

- Try whatever request you are making without a JWT, to validate that data is not returned.
- Change or remove one letter in the JWT and make sure that data is not returned when the JWT is used in a request.
- Decode a valid JWT in the Debugger, change it to have different values, and then see whether the edited JWT will work in your request.
- Use a JWT without a valid signature and make sure you don't get data in the response.
- Note when the JWT expires, and try a request after it expires to make sure you don't get data back.
- Create a JWT that has an issue time somewhere in the future and make sure you don't get data back when you use it in your request.
- Decode a JWT and make sure there is no sensitive information, such as a bank account number, in the payload.

# Chapter 26: Database Testing

As software testers, we often take for granted the fact that our application has a database behind it. When we are testing, we tend to focus on the visible, such as the user interface, or on the application logic of the API. But it's important to test the database as well. Following are six ways to test your application's database.

## **Verify That the Data Fields Are the Correct Type**

Each data field in the database will be a specific type, such as int, float, string, or datetime. Verify that each field's data type is appropriate. For example, a date field in the application should be saved as a datetime type rather than as a string. This might not seem like a big deal, but if sorting functionality is added to the application, you'll find that April 10 is sorted before January 11 because the dates are being sorted alphabetically.

## **Verify That the Fields That Are Required in the Database Are Also Required in the API and the UI**

Generally, each record in a database will have some fields that are required. If a field is required in the database, it should also be required in the API and the UI. If the UI doesn't set the last name field as required but the database has set it as required, a user could submit a new record without a last name and the database will return an error.

## **Verify That the Parameters Set on the Fields in the Database Match the Parameters in the API and the UI**

Data fields in a database will have specific limits set, such as a character limit for a string or a maximum value for an int. The fields in the API and the UI should have the same limits. If the limits do not match, confusion can ensue. Let's say, for example, that your application has a field for a street address. In the UI the developer has set the character limit to 50 characters, but in the database the limit is set to 40. If a user types in a street address with 45 characters, the UI will accept that value but the database will not. This will

result in the record not being saved, and the problem won't be obvious to the user.

## **Verify That Sensitive Data, Such as User Passwords, Are Encrypted in the Database**

I once worked for a company that did not encrypt their user passwords. I had access to the production database, which meant I could see the username and password for every single one of the company's customers. It should be pretty obvious that this was a huge security risk! Passwords should always be encrypted in the database so that there is no way for anyone but the user to know what their password is.

## **Verify That Your Database Supports All of Your API's Operations**

Just because your POST request returns a 200 message doesn't mean the data was saved to the database correctly. When you make an API request, make sure the database saves every field correctly. (You'll learn about API testing in Part IV.) I have seen situations in which a PUT request did not result in every new value being saved to the database. I've also seen a situation in which a PATCH request updated the correct fields but set every other field to null!

Be sure to test every available CRUD operation and check every field for accuracy. Also make sure you change field values from null to an entry and from an entry to null. If a field you are testing is a string, determine with your team whether empty strings will be allowed, and test going from null to an empty string, from an empty string to a value, from a value to an empty string, and so on.

In addition, if your API is going to support the DELETE operation, find out from your team whether those deletes will be hard-deletes, meaning the record will be completely removed from the database, or soft-deletes, meaning the record will stay in the database but move to another table or stay in the existing table with a "deleted" flag set. Then test the DELETE operation to verify that it is behaving as expected.

## **Verify That Leading and Trailing Spaces Are Removed When Saving**

## **to the Database**

Have you ever had trouble logging in to an application but were sure you had the right username? The problem could have been that when the username was originally created, a trailing space was accidentally added and it was saved to the database with the extra space. When you tried to log in as “doglvr49”, the system was expecting “doglvr49 ”. Similarly, if you entered a contact’s last name as “ Jones” instead of “Jones” and the leading space wasn’t trimmed, and then you tried to sort the contacts alphabetically, you’d find “ Jones” listed before “Allan”. When you are testing text fields in your application, try testing them with leading and trailing whitespaces, and then verify in the database that those spaces have been trimmed.

A mismatch between character limits in the database and the UI, having a string instead of an int, or having trailing spaces in a record can seem like a small issue, but it can result in a big problem. By following these tactics in database testing, you will help ensure that your end users will have a good experience entering and retrieving data.

# Chapter 27: Testing with Relational Databases

In the preceding chapter, I discussed various ways to test your application's database. To verify that your data has been saved correctly, you'll need to query the database, and the way to do that will depend on what type of database you have. In the past most databases were relational, but in recent years there has been a trend toward using nonrelational databases. In this chapter and the next I'll address relational databases, and in Chapter 29 I'll talk about nonrelational databases.

Relational databases, such as MySQL and Microsoft SQL Server, are based on tables. Each table relies on a schema, which defines what columns will be in the table, what data types they will have, and which columns will accept null values. Here's an example of a typical SQL table:

Contacts:

| contactId | firstName | lastName  | email                | phone      | city    | state |
|-----------|-----------|-----------|----------------------|------------|---------|-------|
| 10000     | Prunella  | Prunewhip | pprunewhip@fake.com  | 8005551000 | Phoenix | AZ    |
| 10001     | Joe       | Schmoe    | jschmoe@alsofake.com | NULL       | NULL    | RI    |

Note that there are seven different columns in the table. The first column, contactId, is the primary key for the table. This will be a unique value; there will never be two contactIds with the same value.

With a relational database, the schema remains unchangeable, so when Joe Schmoe is added to the database without a phone or city, those places in the table need to be filled with NULL.

Tables in a relational database can connect to one another. Here is a table in the same database that shows the contacts' favorite foods:

Foods:

| foodId | contactId | food      |
|--------|-----------|-----------|
| 1      | 10000     | Pizza     |
| 2      | 10000     | Ice cream |
| 3      | 10001     | Sushi     |

In this table, the primary key is foodId. But notice that the contactId is present in this table, and the values are the same as those in the first table. So we can see in this table that Prunella has two different favorite foods, pizza and ice cream, and Joe's favorite food is sushi.

When testing a relational database, you can use SQL query language to verify that the values you are looking for are present in the database. For example, if you had just added a new contact with the name of Amy Smith to the Contacts table, you could query the database to see whether it had been added, like this:

```
select * from Contacts where lastName = 'Smith' and firstName = 'Amy'
```

and the query would return a table row in response:

| contactId | firstName | lastName | email                | phone      | city  | state |
|-----------|-----------|----------|----------------------|------------|-------|-------|
| 10003     | Amy       | Smith    | amysmith@faketoo.com | 8885551001 | Boise | ID    |

In the preceding query, the asterisk, \*, tells SQL that we want all the columns for the record returned.

Because this is a relational database, you could also do a query with a join. A SQL join combines the data from two tables, joining on a column that they have in common.

In the preceding examples, both tables have a contactId column. Let's say you have given your new contact, Amy, a favorite food (chocolate), and you want to verify that it has been saved to the database correctly but you don't know what Amy's contactId is. You can't just query the Food table for "Amy Smith", because her first and last names aren't in there. And you can't query the Contacts table for the food, because it's not in that table. But you could query the Contacts table with that information, get the contactId from that, and then use the contactId to query the Food table for the favorite food, creating a join.

This is what such a query would look like:

```
select food from Foods
inner join on Contacts
where Foods.contactId = Contacts.contactId
and Contacts.firstName = 'Amy'
and Contacts.lastName = 'Smith'
```

The query will return this response:

food

Chocolate

Let's walk through what happens in the query:

- **select food from Foods:** This tells SQL to return just the food column from the Foods table.
- **inner join on Contacts:** This tells SQL that the query will be joining information from the Foods table with information from the Contacts table.
- **where Foods.contactId = Contacts.contactId:** This is instructing SQL to find the contactIds in the Foods table and match them up with the contactIds from the Contacts table.
- **and Contacts.firstName = 'Amy' and Contacts.lastName = 'Smith':** These last two lines are telling SQL that we are only interested in the record with the first name Amy and the last name Smith.

There are many more complicated ways to query a relational database, but with these two query types you will be able to do most of your data validation.

# Chapter 28: SQL Query Secrets

Have you ever been querying a SQL table and one of your queries seems to take forever, but then the next query you run takes milliseconds? This would frequently happen to me, and I thought it meant the server that hosted the database was unreliable in some way. But it turns out that the way we structure our queries has a huge impact on how long they will take to execute. In this chapter, I describe what indexes are and talk about the ways we can use them to optimize our queries.

## What Is an Index?

An index is a database structure that is designed to speed up queries in a table. An easy way to understand this is to think about the index at the back of a book. Let's say you have a book on car repair and you want to find information about your car's brakes. You could look up "brakes" in the index, or you could search through every single page of the book for the word "brakes". It's pretty obvious which would take less time!

Unlike books, databases can have more than one index. There are two different kinds of indexes: clustered and unclustered. A clustered index is used to store a table in sorted order. There can only be one clustered index because the table is stored in only one order. Unclustered indexes are stored in the original table order, but they save the location of certain fields in the table.

Let's take a look at an example. If we had a table like this, called the `Users` table:

| <b>UserId</b> | <b>State</b> | <b>LastName</b> | <b>FirstName</b> | <b>Email</b>        | <b>Mobile Phone</b> |
|---------------|--------------|-----------------|------------------|---------------------|---------------------|
| 1             | MA           | Prunewhip       | Prunella         | pprunewhip@fake.com | 800-867-5309        |
| 2             | RI           | Schmoe          | Joe              | jschmoe@notreal.com | 401-555-8765        |
| 3             | NH           | Smith           | Amy              | amysmith@foo.com    | 603-555-3635        |
| 4             | RI           | Jones           | Bob              | bob@bar.com         | 401-555-2344        |
| 5             | MA           | Jones           | Amy              | aj@me.com           | 617-555-2310        |

and we had a clustered index defined to have UserId as the key, a search on UserId would be very fast, and the data returned would be in order by UserId.

The table could also use unclustered indexes, such as the following:

- **State:** The records in the table are indexed by state.
- **LastNameFirstName:** The records in the table are indexed by LastName and FirstName.

When you query a database, the query will first look to see whether an index can be used to speed up the search. For example, if I made this request:

`select LastName, FirstName from Users where UserId = 5`

the query would use the UserId index and the LastNameFirstName index to find the record.

Similarly, if I made this request:

`select LastName, FirstName from Users where State = 'MA'`

the query would use the LastNameFirstName index and the State index to find the record.

Of course, with a table of only five records, optimizing in this way won't make much of a difference. But imagine that this table had 5 million records and you can see how using an index would be very helpful.

Querying a table on a nonindexed field is called a table scan. The query needs to search through the entire table for the values, just as a person who wasn't using a book index would have to search through every single page of the book for a term they were looking for.

How can you know what indexes a table has? You can find out with one simple query:

```
EXEC sp_helpindex "Users"
```

where you would replace "Users" with the name of the table. This will return a result of all the clustered and unclustered indexes applied to the table, and the result will include the name of the index, a description of the index, and all the keys used in the index.

If you want to optimize your SQL queries, only ask for the data you really need, rather than asking for `select *`. Because not every field in the table is indexed, looking for every field will take longer.

Let's say you want to query the Users table to find the email addresses of all the users who live in Massachusetts (MA). But you also would like to have some more information about those users. You could ask for this:

```
select FirstName, LastName, Email from Users where State = 'MA'
```

To find the records, the query will use the FirstNameLastName index and the State index. Only the Email will be a nonindexed field.

But if you asked for this:

```
select * from Users where State = 'MA'
```

now the query needs to look for two different nonindexed fields: Email and Mobile Phone.

Another helpful tip is to specify all the keys in an index when you want to use that index to make a query. For example, if you wanted to find the Email for Prunella Prunewhip, you should ask for this:

`select Email from Users where LastName = 'Prunewhip' and FirstName = 'Prunella'`

rather than this:

`select Email from Users where LastName = 'Prunewhip'`

In the second example, the LastNameFirstName index won't be used.

When you want to use an index, the query will run faster if you specify the keys in the order they appear, so it's better to say  
`where LastName = 'Prunewhip' and FirstName = 'Prunella'`  
than it is to say  
`where FirstName = 'Prunella' and LastName = 'Prunewhip'.`

Here's one more tip: when you want to use an index, be sure not to manipulate one of the index keys in your query, because this will mean the index won't be used. For example, if you had a table like this, called Grades:

| <b>StudentId</b> | <b>LastName</b> | <b>FirstName</b> | <b>Grade</b> |
|------------------|-----------------|------------------|--------------|
| 1                | Miller          | Kara             | 89           |
| 2                | Smith           | Carol            | 56           |
| 3                | Jones           | Bob              | 99           |
| 4                | Davis           | Frank            | 78           |
| 5                | Green           | Doug             | 65           |

and you had an unclustered index called LastNameGrade, and you executed the following query:

`select LastName from Grades where (Grade + 100) = 178`

the LastNameGrade index wouldn't be used, because the Grade value was being manipulated. It's necessary for the query to go through the entire table and add 100 to each Grade field to search for the correct value.

Armed with this knowledge, you should be able to create queries that will run as fast as possible, getting you the data you need.

# Chapter 29: Testing with Nonrelational Databases

In the preceding two chapters, I discussed ways to query relational databases for testing. In this chapter I will explain nonrelational databases, describe how they are different from relational databases, and discuss how to query them in your testing. Nonrelational databases, such as MongoDB and DynamoDB, are sometimes called NoSQL databases and are becoming increasingly popular in software applications.

The main difference between relational and nonrelational databases is that relational databases use tables to store their data, whereas nonrelational databases use documents. The documents are often in JSON format (more on JSON format in Chapter 42). Let's take a look at what the records in the Contacts table from Chapter 27 would look like if they were in a nonrelational database:

```
{
  contactId: "10000",
  firstName: "Prunella",
  lastName: "Prunewhip",
  email: "pprunewhip@fake.com",
  phone: "8005551000",
  city: "Phoenix",
  state: "AZ"
}
{
  contactId: "10001",
  firstName: "Joe",
  lastName: "Schmoe",
  email: "jschmoe@alsofake.com",
```

```
    state: "RI",
}
```

Note that Joe does not have a value for phone or city entered, so they are not included in his document. This is different from relational databases, which are required to include a value for every field. Instead of having a NULL value for phone and city as Joe's record did in the SQL table, those fields are simply not listed.

Another key difference between relational and nonrelational databases is that it's possible to add a new field to a table without adding it in for every document. Let's imagine we are adding a new record to the table and we want that record to include a spouse's name. When that record is added it will look like this:

```
{
  contactId: "10002",
  firstName: "Amy",
  lastName: "Smith",
  email: "amysmith@faketoo.com",
  phone: "8885551001",
  city: "Boise",
  state: "ID",
  spouse: "John"
}
```

The original documents, 10000 and 10001, don't need to have this spouse value. In a relational database, if a new field is added the entire schema of the table needs to be altered, and Prunella and Joe will need to either have spouse values or NULL entered in those fields.

With a nonrelational database, it's not possible to do joins on table data, as you saw in Chapter 27. Each record should be treated as its own separate document, and you can do queries to retrieve the documents you want. What that query language looks like depends on the type of database used. The following examples use MongoDB's query language, which is JavaScript based, and query on the documents listed earlier:

- `db.contacts.find()`: This will return all the contacts in the table.
- `db.contacts.find( { contactId: "10001" } )`: This will return the document for Joe Schmoe.

To make the responses easier to read, you can append the command `.pretty()`, which will organize the data returned in JSON format rather than a single line of values.

You can also run a query to return a single field for each document:

- `db.contacts.find( {}, {firstName:1, _id:0} )`: This will return just the first name for each contact. The `_id:0` setting is asking the query not to return IDs for the records, which a query does by default.

Because the documents in a nonrelational database have a JSON-like structure, it's possible to have documents with arrays. For example, our Contacts table could have a document that lists a contact's favorite foods:

```
{
  contactId: "10000",
  firstName: "Prunella",
  lastName: "Prunewhip",
  email: "pprunewhip@fake.com",
  phone: "8005551000",
  city: "Phoenix",
  state: "AZ",
  foods: [ "pizza", "ice cream" ]
}
```

It's even possible to have objects within arrays, as follows:

```
{
  contactId: "10001",
  firstName: "Joe",
  lastName: "Schmoe",
  email: "jschmoe@alsofake.com",
  state: "RI",
  pets: [ { type: "dog", name: "fido" }, { type: "cat", name: "fluffy" } ]
```

```
}
```

You can see how this type of data storage might be advantageous for your application's data. Nesting data in this fashion makes it easier to read at a glance than it would be in a relational database, where the pets might be in their own separate table.

To run a query that will return all the contacts that have cats, you would simply request:

```
db.contacts.find( {"pets.type":"cat"} )
```

To run a query that will return all the contacts that have cats named Fluffy, you would request:

```
db.contacts.find( {$and: [{"pets.type":"cat"}, {"pets.name":"fluffy"}]} )
```

These are just a few simple examples of how to query data with a nonrelational database, and they should be enough to get you started in your testing. To learn more, be sure to read the documentation for the type of database you are using. As nonrelational databases become increasingly popular, this knowledge will be extremely useful.

# Chapter 30: Serverless Architecture

Have you heard of serverless architecture and wondered what it could possibly be? How could an application be deployed without a server? Here's the secret: it can't.

Remember a few years ago when cloud computing first came to the public, and it was common to say, "There is no cloud, it's someone else's computer"? Now we can say, "There is no serverless architecture; you're just using someone else's server."

Serverless architecture means using a cloud provider for the server. Often the same cloud provider will also supply the database, an authentication service, and an API gateway. Examples of serverless architecture providers include AWS (Amazon Web Services), Microsoft Azure, Google Cloud, and IBM Cloud Functions.

Why would a software team want to use serverless architecture? Here are several reasons:

- You don't have to reinvent the wheel. When you sign up to use serverless architecture you get many features, such as an authentication service, a backend database, and monitoring and logging, directly in the service.
- You don't have to purchase and maintain your own equipment. When your company owns its own servers, it's responsible for making sure they are safely installed in a cool place. The IT team needs to make sure all the servers are running efficiently and that they're not running out of disk space. But when you are using a cloud provider's servers, that responsibility falls to the provider. There's less initial expense for you to get started, and less for you to worry about.
- The application can scale up and down as needed. Most serverless providers automatically scale the number of servers your app is

running on depending on how much demand there is for your app at that moment. So if you have an e-commerce app and you are having a big sale, the provider will add more servers to your application for as long as they're needed, then scale back down when the demand wanes.

- With many serverless providers, you only pay for what you use. So if you are a startup and have only a few users, you'll only be paying pennies per month.
- Applications are really easy to deploy with serverless providers. They take care of most of the work for you. And because the companies that are offering cloud services are competing with one another, it's in their best interest to make their development and deployment processes as simple as possible. So, deployments will certainly get even easier in the future.
- Monitoring is usually provided automatically. It's easy to take a look at the calls to the application and gather data about its performance, and it's easy to set up alarms that will notify you when something's wrong.

Of course, nothing in life is perfect, and serverless architecture is no exception. Here are some drawbacks to using a serverless provider:

- There may be some things you want to do with your application that your provider won't let you do. If you set up everything in-house, you'll have more freedom.
- If your cloud provider goes down, taking your app with it, you are completely helpless to fix it. Recently AWS was the victim of a DDoS attack. In an effort to fight off the attack, AWS blocked traffic from many IP addresses. Unfortunately, some of those addresses belonged to legitimate customers, so the IP blocking rendered its applications unusable.
- Your application might be affected by other customers. For example, a company that encodes video files for streaming received a massive upload of videos from one new customer. It swamped the encoding

company, which meant that other customers had to wait hours for their videos to be processed.

How do you test serverless architecture? The simplest answer is that you can test it the same way you would test an in-house application. You'll be able to access your web app through your URL in the usual way. If your application has an API, you can make calls to the API using Postman, curl, or your favorite API testing tool.

If you are given login access to the serverless provider, you can also do things like query the datastore, see how the API gateway is set up, and look at the logs. You'll probably have more insight into how your application works than you will with a traditionally hosted application.

The best way to learn how serverless architecture works is to play around with it yourself. You can sign up for a free AWS account and follow along with a tutorial. After you get some experience with serverless architecture, you will have no trouble figuring out all kinds of great ways to test it.

## Part IV: API Testing

# Chapter 31: Introduction to REST Requests

More and more companies are moving toward a microservices model for their applications. This means different sections of their application can have a separate datastore and separate commands for interacting with that datastore. The advantage to this is that it's easier to deploy changes to a small component of the application rather than the entire application; it also means that if one microservice goes down, the rest of the application can continue to function.

For example, let's imagine you have a website for a bike rental service. The site has one microservice for the reservation system and a second microservice for the inventory. If the microservice for the inventory goes down, users will still be able to make reservations for bike rentals using cached data from the inventory microservice.

Most microservices are using APIs, or application programming interfaces, which are a set of commands for how a service can be used. And most APIs are using REST requests, or Representational State Transfers, through HTTP to request and send data.

Yet, despite the common usage of REST APIs in today's applications, many testers do not know just how easy it is to test them. Why would you want to test REST requests rather than just test through the UI? Here are a few good reasons:

- Testing REST requests means you can find bugs earlier in the development process, sometimes even before the UI has been created!
- Malicious users know how to make REST requests and can use them to exploit security flaws in your application by making requests the UI doesn't allow; you'll want to find and fix these flaws before they are exploited.
- It's easy to automate REST requests, and they run much faster than UI automation.

To get started with REST testing, first think about what you see in a URL when you navigate to a website. For example, you might see <https://www.foobank.com/customers/login>. It's easy to see how this URL is defining what page you are navigating to:

- “https” specifies that this is a secure request.
- “www.foobank.com” is the domain, which says you want to go to the Foobank website.
- “customers” is the first part of the path, which says you are a customer and therefore want to go to the Customers section of the website.
- “login” is the last part of the path, which says you want to go to the login screen.

One thing that's not seen in the URL is the type of REST request being made. This is known as an HTTP verb, and they are easy to understand:

- A POST request adds a new record to the database.
- A GET request retrieves a record from the database.
- A PUT request takes a record from the database and replaces it with a new record.
- A PATCH request modifies an existing record in the database.
- A DELETE request removes a record from the database.

In this case, a record can be any section of data that is grouped together. For example, it could be a mailing address for a customer, all the contact information for that customer, or every single datapoint associated with that customer. It's up to the API's creators to decide what should make up the record.

In the next several chapters, I'll discuss each HTTP verb in detail and describe how to test them. I am sure you will be excited to discover just how much you can test without navigating to your application's web pages!

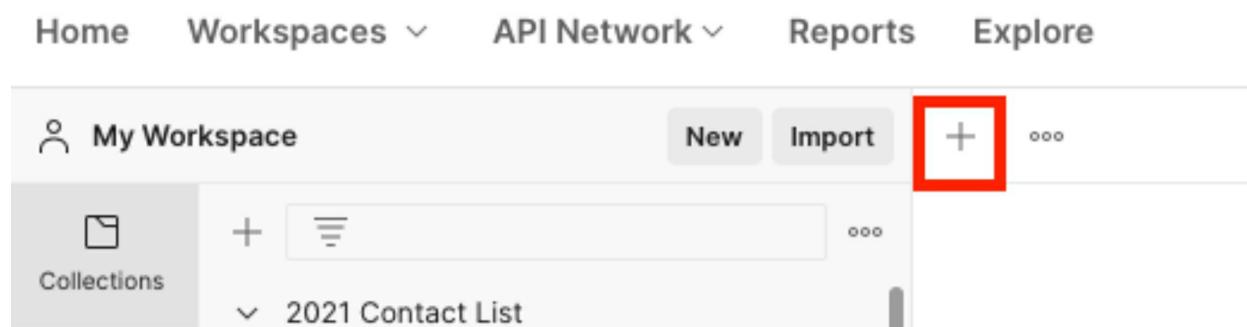
# Chapter 32: Getting Set Up for API Testing

We'll be using the Contact List API to learn about REST requests. Since the Contact List API requires authentication, you will need to have an authentication token to make all your requests. And to have an authentication token, you'll need a username and password. To create these, simply go to <https://thinking-tester-contact-list.herokuapp.com>, click on the "Sign up" button, fill out the fields on the Add User page, and click Submit. You can use any email address for your account; it does not have to be real, though it does have to be in the usual "abc@def.com" format. Make a note of what email and password you chose because you will need them to generate an authentication token.

Next, you will need to use an API testing application to create and send REST requests. My favorite testing application is Postman, which is available for free at <https://www.postman.com/downloads>. You'll need to create a username and password to log in to Postman. Once you are logged in, it's time to create your first request!

Note that Postman periodically changes their user interface and the screens that are displayed when you first create an account. Once you've moved through the initial screens, make sure that you've selected the "Collections" tab on the left of the screen.

Click on the plus (+) button on the upper-left of the screen to create a new request:



You will see that the HTTP verb dropdown is currently set to GET. Click this dropdown and select POST instead:

The screenshot shows the Postman application interface. At the top, there's a red header bar with the text "GET Untitled Request" and icons for closing, adding, and more. Below this is a white header section with the text "Untitled Request". Underneath is a grey configuration bar. On the left of the bar is the word "GET" in bold. To its right is a dropdown arrow button, which is highlighted with a red square. Further to the right is the placeholder text "Enter request URL".

In the section that says “Enter request URL”, type the following:  
<https://thinking-tester-contact-list.herokuapp.com/users/login>

The screenshot shows the Postman interface again. The configuration bar now has "POST" selected instead of "GET". To the right of the dropdown is the URL "https://thinking-tester-contact-list.herokuapp.com/users/login".

Click on the Body tab underneath the URL:

The screenshot shows the Postman interface with the configuration bar still showing "POST" and the URL "https://thinking-tester-contact-list.herokuapp.com/users/login". Below the configuration bar, there are several tabs: "Params", "Authorization", "Headers (9)", "Body", "Pre-request Script", "Tests", and "Settings". The "Body" tab is highlighted with a red square.

Click on the “raw” button:

Params    Authorization    Headers (9)    **Body**    Pre-request Script    Tests    Settings  
● none    ● form-data    ● x-www-form-urlencoded    **raw**    ● binary    ● GraphQL

Click on the Text dropdown, and select JSON instead:

Params    Authorization    Headers (9)    **Body**    Pre-request Script    Tests    Settings  
● none    ● form-data    ● x-www-form-urlencoded    **raw**    ● binary    ● GraphQL    **Text** ▾

JSON stands for JavaScript Object Notation. I'll talk more about JSON in Chapter 42.

Add this JSON text in the text field...

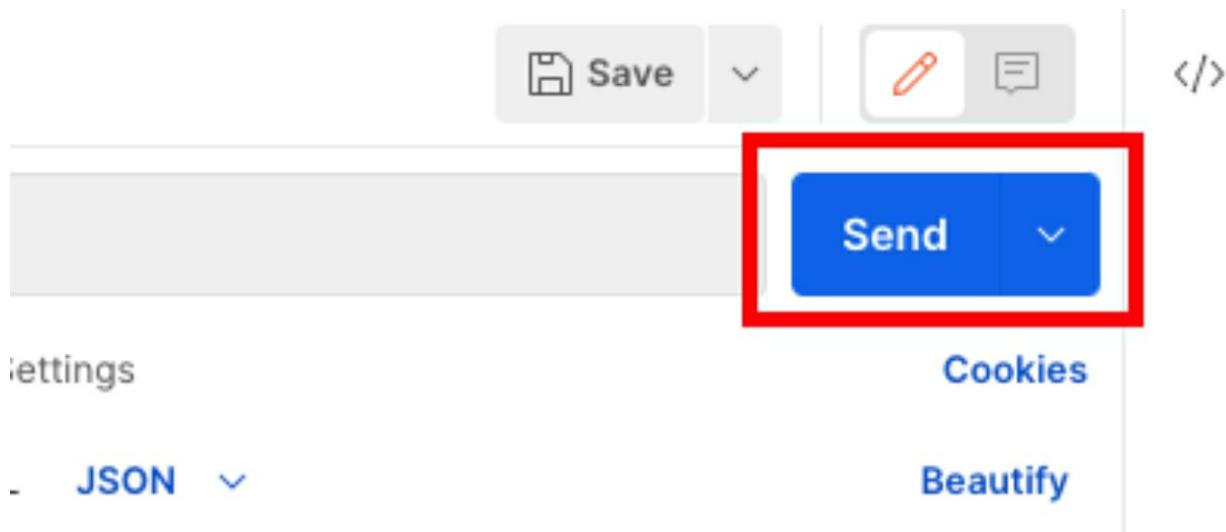
POST    https://thinking-tester-contact-list.herokuapp.com/users/login

Params    Authorization    Headers (9)    **Body** ●    Pre-request Script    Tests    Settings  
● none    ● form-data    ● x-www-form-urlencoded    **raw**    ● binary    ● GraphQL    **JSON** ▾

```
1 {  
2   "email": "your email here",  
3   "password": "your password here"  
4 }
```

...but replace “your email here” with the email you set for your user account and “your password here” with the password you set for your user account. Both the email and the password should be in quotation marks so that they appear blue.

Click the blue Send button:



If everything is set up correctly, you should see text in the Response window:

The screenshot shows the Postman interface with a successful API response. The top navigation bar includes tabs for Body, Cookies (1), Headers (9), and Test Results (5/5). On the right, there's a status indicator showing 200 OK with a globe icon. Below the tabs, there are buttons for Pretty, Raw, Preview, Visualize, and JSON (with a dropdown arrow). The JSON response body is displayed in a code editor-like format with line numbers (1-10) on the left. The response content is as follows:

```
1 {
2   "user": {
3     "_id": "60aace7dec4b3a0015155a07",
4     "firstName": "Greyson",
5     "lastName": "Yundt",
6     "email": "destany.ebert@gmail.com",
7     "__v": 3
8   },
9   "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJfaWQiOiI2MGFhY2U3ZGVjNGIzYTAwMTUxNTVhMDciLCJpYXQiOjE2MjE4MDY3MzJ9.
9AtHEVg9HUMhTbuYKHCyLaq_XWZiTYvwEddQlDKwVbU"
10 }
```

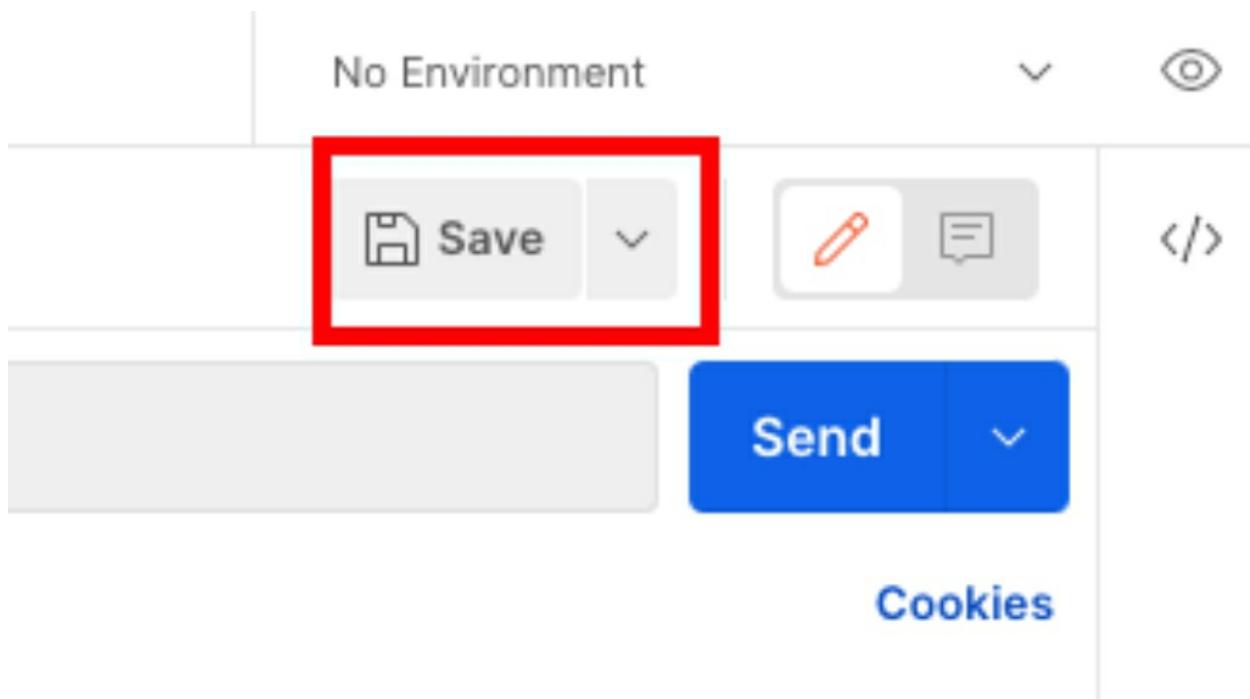
Your response will, of course, look different because your user is different.

You have now been assigned an authentication token! Copy the token, which is everything between the blue quotation marks after “token”. In the preceding example, the token would be:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9eyJfaWQiOiI2MGFhY2YwM

Save this token somewhere, such as in a text file, because we'll be using it in future Postman requests. This token has a very long life, but it will expire eventually. If that happens, all you need to do is create a new token by doing another login POST request. Let's save the POST request so that you won't have to re-create it later.

Click the Save button in the upper-right of the screen:



A pop-up will appear:

## SAVE REQUEST

Request name

`https://thinking-tester-contact-list.herokuapp.com/users/login`

Add description

**Save to** Select a collection/folder



**You don't have any collections.**

Collections let you group related requests, making them easier to access and run.

**Create a collection**

New Collection

Cancel

Save

Change the “Request name” to something that’s easy to read and makes sense, such as “Login”. Then click the “Create a collection” button. The collection section will open:

## SAVE REQUEST

Request name

Login

Add description

**Save to** Select a collection/folder

≡ Search for collection or folder



Name your collection

Create

Cancel

New Collection

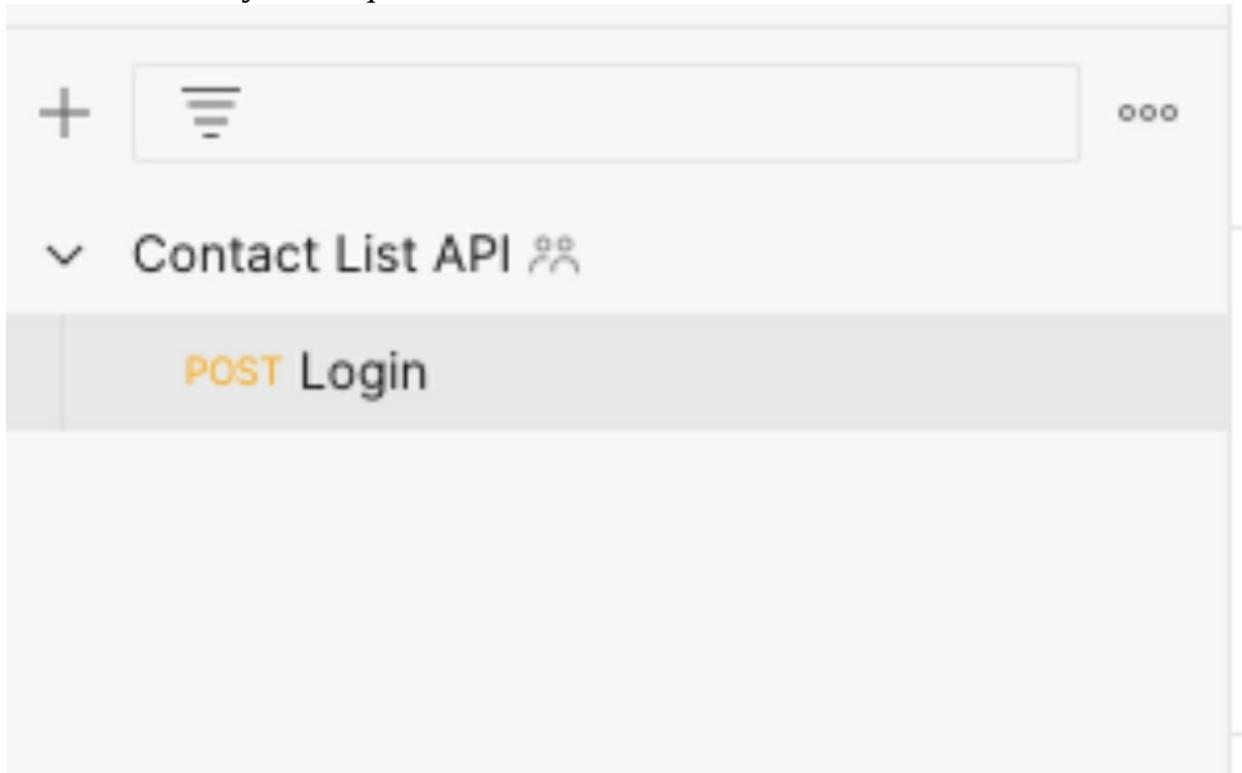
Cancel

Save

Give your collection a memorable name, such as “Contact List API”. Then click Create. Finally, click the Save button.

You should see in the left panel of the Postman window that your

collection and your request have been saved:



Now that we have an authentication token to use and a request saved that will help generate it, we can start learning more about the different types of REST requests.

# Chapter 33: Testing GET Requests

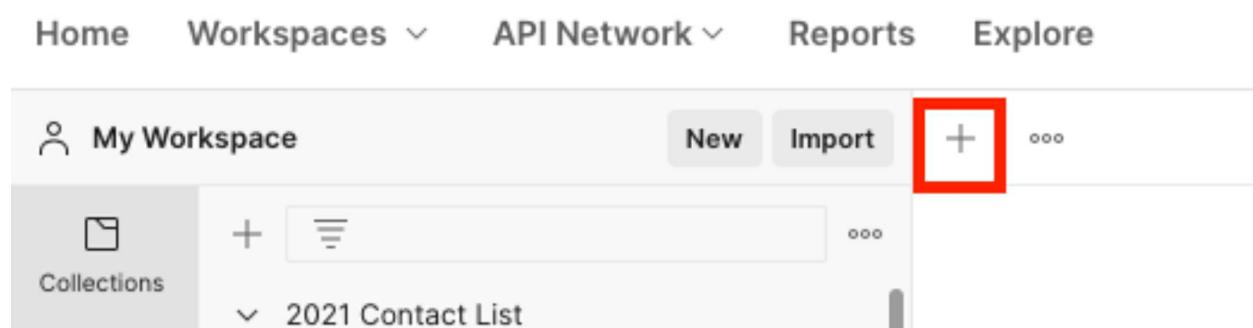
We will begin our discussion of REST request types with the GET request. This is usually the easiest request to test because all we are doing is retrieving data from the database. We don't need to worry about whether we are manipulating data correctly; we just need to retrieve it and check that we get a correct response.

To get set up for testing GET requests in our API, let's add some contacts to the Contact List app. Navigate to <https://thinking-tester-contact-list.herokuapp.com> in your web browser and log in with the email and password you used in Chapter 32.

Click Add a New Contact, enter some contact information, and click Submit. You should see that your new contact has been added to the Contact List page. Repeat this process two more times so that you have three contacts in your contact list.

Now that you have some contacts to work with, let's return to Postman and set up a GET request.

Click on the plus (+) button at the top of the screen to create a new request:



Notice that the request verb is already set to GET, so we don't need to make any change there. Next, enter the request URL:

<https://thinking-tester-contact-list.herokuapp.com/contacts>

GET https://thinking-tester-contact-list.herokuapp.com/contacts

Params Authorization Headers (8) Body Pre-request Script Tests Settings

This request will go to the server and ask for the list of all the contacts you created. But there's one more thing we need to do to run this request, and that is to add our Authorization token. Click on the Authorization tab:

GET https://thinking-tester-contact-list.herokuapp.com/contacts

Params **Authorization** Headers (8) Body Pre-request Script Tests Settings

Query Params

From the Type dropdown, select Bearer Token:

Params **Authorization** ● Headers (9) Body

Type **Bearer Token**

The authorization header will be automatically generated when you send the request.

[Learn more about authorization ↗](#)

In the text field next to Token, paste in the authentication token you created in the previous chapter:

Token

eyJhbGciOiJIUzI1NilsInR5cCI6IkpXVCJ9.ey...

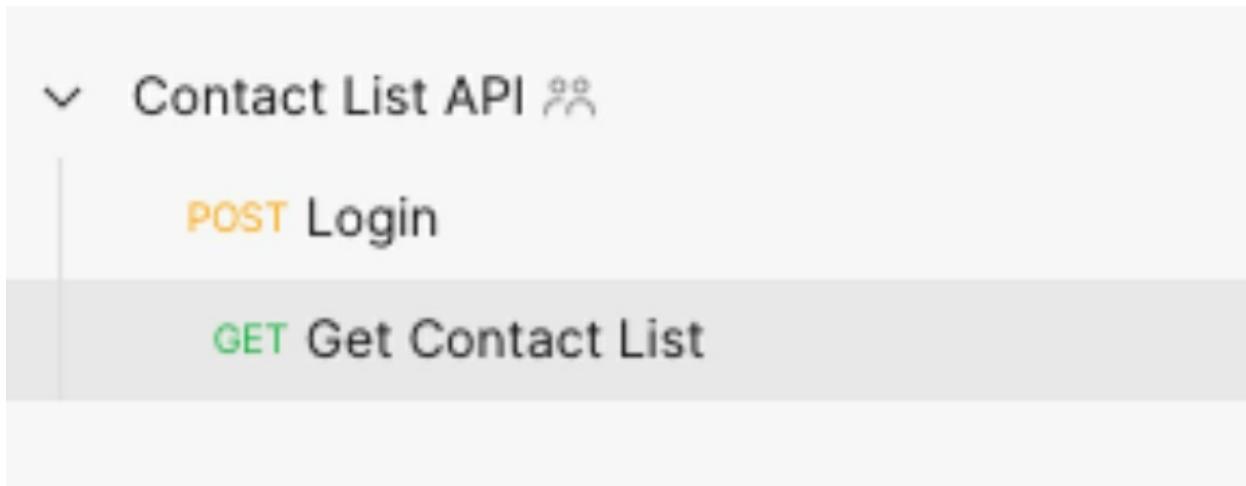
Now you are ready to send the request. Click the Send button, and you should see the three contacts you created in the body of the response. The code will look something like this:

```
{  
    "_id": "60aad929ec4b3a0015155a14",  
    "firstName": "Carol",  
    "lastName": "Brady",  
    "birthdate": "1931-07-05",  
    "email": "cbrady@fake.com",  
    "phone": "8005551001",  
    "street1": "123 Elm St.",  
    "city": "Los Angeles",  
    "stateProvince": "CA",  
    "postalCode": "90210",  
    "country": "USA",  
    "owner": "608a0fa49c2dd10015d2a051",  
    "__v": 0  
},  
{  
    "_id": "60aad872ec4b3a0015155a11",  
    "firstName": "Mike",  
    "lastName": "Brady",  
    "birthdate": "1930-04-05",  
    "email": "mbrady@fake.com",  
    "phone": "8005551001",  
    ...  
}
```

(partial screenshot)

You should see three complete records in your response body; you may need to scroll down to see them all.

Let's save this request. Click on the Save button, give the request a good name, like "Get Contact List", choose your Contact List API collection if it is not already selected, and click Save. You should see that your request is saved to your collection on the left side of the screen:



Get Contact List is a very simple request asking for a complete list of all the contacts associated with your account. But we can also use a GET request to ask for one particular contact.

To do this, look in the response section of your Get Contact List request and choose an ID to copy:

```
{  
    "_id": "60aad929ec4b3a0015155a14",  
    "firstName": "Carol",  
    "lastName": "Brady",  
    "birthdate": "1931-07-05",  
    "email": "cbrady@fake.com",  
    "phone": "8005551001",  
    "street1": "123 Elm St.",  
    "city": "Los Angeles",  
    "stateProvince": "CA",  
    "postalCode": "90210",  
    "country": "USA",  
    "owner": "608a0fa49c2dd10015d2a051",  
    "__v": 0  
},  
{  
    "_id": "60aad872ec4b3a0015155a11",  
    "firstName": "Mike",  
    "lastName": "Brady",  
    "birthdate": "1930-04-05",  
    "email": "mbrady@fake.com",  
    "phone": "8005551001",  
    ...  
}
```

Your ID will, of course, be different. Copy this ID and paste it into a text

file for now.

Let's create a new request. Click on the plus (+) button, and in the URL, type <https://thinking-tester-contact-list.herokuapp.com/contacts/<your contact's id>>, replacing <your contact's id> with the contact ID you pasted into your text file:



We'll need to add our authentication token again, so click on the Authorization tab, choose Bearer Token, and paste your token into the token text field.

Now click Send to run the request. You should see just one contact returned in the response body, and it should be the contact that has the ID you included in the request URL:

```
{  
  "_id": "60aad929ec4b3a0015155a14",  
  "firstName": "Carol",  
  "lastName": "Brady",  
  "birthdate": "1931-07-05",  
  "email": "cbrady@fake.com",  
  "phone": "8005551001",  
  "street1": "123 Elm St.",  
  "city": "Los Angeles",  
  "stateProvince": "CA",  
  "postalCode": "90210",  
  "country": "USA",  
  "owner": "608a0fa49c2dd10015d2a051",  
  "__v": 0  
}
```

Save this request by clicking the Save button, giving the request a name like “Get Contact”, and saving it to your Contact List API collection.

Now that you understand how GET requests work, let’s think about how to test them. What happens if we put a contact ID in the URL that doesn’t exist? What happens if we use a contact ID that exists but doesn’t belong to our user account? What happens if we pass in “0” or “foo” for the contact ID?

The status codes you get after you make a REST request tell you a bit

about the behavior of your application. We'll discuss status codes further in Chapter 38, but for now, note that a status code of 200 is good and a status code that begins with 4 generally indicates that something has gone wrong. You may get a 404 status when you search for a contact ID that doesn't exist. This means the record was not found:

The screenshot shows a REST API testing interface. At the top, there are tabs for 'Body' (which is selected), 'Cookies (1)', 'Headers (6)', and 'Test Results'. On the right, a status bar displays 'Status: 404 Not Found' with a red border around it. Below the tabs, there are buttons for 'Pretty', 'Raw', 'Preview', 'Visualize', 'Text' (with a dropdown arrow), and a copy icon. The main area contains the number '1' followed by a JSON response: { "error": "Please authenticate." }.

If we don't put a contact ID in the URL, it will match the first GET request we created—the Get Contact List request—which will return all the contacts that belong to your user account.

We can test other things with this endpoint, such as its security. For instance, what happens if you don't put in the authentication token? What happens if you put in a bad token? You can do these security tests with the Get Contact List request as well. Generally speaking, making a request with a missing or invalid token will return a 401 status code:

The screenshot shows a REST API testing interface. At the top, there are tabs for 'Body' (selected), 'Cookies (1)', 'Headers (8)', and 'Test Results'. On the right, a status bar displays 'Status: 401 Unauthorized' with a red border around it. Below the tabs, there are buttons for 'Pretty', 'Raw', 'Preview', 'Visualize', 'JSON' (with a dropdown arrow), and a copy icon. The main area contains the number '1' followed by a JSON response: { "error": "Please authenticate." }.

You now understand the basics of GET requests and how to test them. In the next chapter we'll move on to POST requests.

# Chapter 34: Testing POST Requests

POST requests are perhaps the most important of the REST requests because they add new records to your application's database. It's very important to test your POST requests well because they will have a direct impact on the quality of data in your database.

You learned to do a POST request in Chapter 32; you created a POST request to log in with your user account. In this chapter, you'll learn how to do a POST request to create a new contact.

Let's take a look at the documentation for the Contact List API to see what a POST request will look like. Navigate to <https://documenter.getpostman.com/view/4012288/TzK2bEa8> in your web browser. The documentation shows what all the endpoints in the Contact List API are like: what the URL should be, what should be passed in the body of the request, and what the response should be.

The first endpoint in the documentation is the Add Contact POST request, which is exactly what we are looking for:

## POST Add Contact

```
https://thinking-tester-contact-list.herokuapp.com/contacts
```

### HEADERS

**Authorization**      Bearer {{token}}

### BODY raw

```
{
  "firstName": "John",
  "lastName": "Doe",
  "birthdate": "1970-01-01",
  "email": "jdoe@fake.com",
  "phone": "8005555555",
  "street1": "1 Main St.",
  "street2": "Apartment A",
  "city": "Anytown",
  "stateProvince": "K.    View More
  "postalCode": "12345".
```

Let's set up a request like this in Postman. Click on the plus (+) button to create a new request; then click on the HTTP verb dropdown and choose POST. Next, add this URL: <https://thinking-tester-contact-list.herokuapp.com/contacts>. You may have noticed that this is the same URL we used for our Get Contact List request. The only difference is the HTTP verb: that request used GET, and now we are using POST.

Now we'll add the body of the request. Click on the Body tab and then click the "raw" button. Change the "text" dropdown to read "JSON". Your request should look like this:

POST https://thinking-tester-contact-list.herokuapp.com/contacts

Params Authorization Headers (9) **Body** Pre-request Script Tests Settings

none  form-data  x-www-form-urlencoded  raw  binary  GraphQL **JSON** ▾

---

1

Go to the body of the request in the documentation, copy the text in the body, and paste it into the body section of your Postman request:

The screenshot shows the Postman interface with a POST request to `https://thinking-tester-contact-list.herokuapp.com/contacts`. The Body tab is selected, displaying a JSON object with 13 properties representing a contact record:

```
1 {
2   "firstName": "John",
3   "lastName": "Doe",
4   "birthdate": "1970-01-01",
5   "email": "jdoe@fake.com",
6   "phone": "8005555555",
7   "street1": "1 Main St.",
8   "street2": "Apartment A",
9   "city": "Anytown",
10  "stateProvince": "KS",
11  "postalCode": "12345",
12  "country": "USA"
13 }
```

Finally, add the authorization. Click on the Authorization tab and choose Bearer Token from the Type dropdown. Then paste in your token from Chapter 32.

Now you are ready to run the request. Click the Send button, and you should get a 201 response code with the contact that was added in the body of the response.

Next, save the request: click the Save button, name the request “Add Contact”, and save it to your Contact List API collection.

Let’s check to see whether the contact was really added to the database by doing a GET request to get the contact. Copy the ID from the response you just got, return to your Get Contact request, and replace the ID in the URL with your new contact ID. Click Send and you should see your contact returned in the response:

The screenshot shows the Postman interface with a successful API call. The top bar indicates a GET request to the URL <https://thinking-tester-contact-list.herokuapp.com/contacts/60ada775d90edc0015ebe4aa>. The Headers tab is selected, showing the content-type as application/json. The Body tab is shown below, stating "This request does not have a body". The response section shows a status of 200 OK with a JSON response body:

```
1 {  
2   "_id": "60ada775d90edc0015ebe4aa",  
3   "firstName": "John",  
4   "lastName": "Doe",  
5   "birthdate": "1970-01-01",  
6   "email": "jdoe@fake.com",  
7   "phone": "8005555555",  
8   "street1": "1 Main St.",  
9   "street2": "Apartment A",  
10  "city": "Anytown",  
11  "stateProvince": "KS",  
12  "postalCode": "12345",  
13  "country": "USA",  
14  "owner": "608a0fa49c2dd10015d2a051",  
15  "__v": 0  
16 }
```

In Chapter 11 you learned that there are many different ways to test forms. This is true when testing POST requests as well. First, you can test for required fields. In the Contact List app, only the first and last names are required. So you can try to send a request with just those two fields, and verify that you get a successful response. That request would look like this:

```
1 {  
2   ... "firstName": "John",  
3   ... "lastName": "Doe"  
4 }
```

Then you can try sending a request with the first name missing and verify that you get an error message. You could try sending a request with the last name missing and verify that you get an error message. You could also try sending an empty request, with just the curly brackets, { }, and verify that you get an appropriate error message.

It's also a good idea to do some additional happy path testing where you are sending in various combinations of the required fields with some of the nonrequired fields. After adding each contact, you can do a GET request to make sure it has been added correctly.

Now it's time to test the limits of every field. We want to make sure the field limits of the API match the field limits of the UI. The first name field, for example, has a limit of 20 characters. So you'll want to test with just one character, with 19 characters, with 20 characters, and with 21 characters. When you send in a first name with 21 characters, you should get an error in response. For fields like the birth date, the email address, the phone number, and the postal code, you'll want to check that only valid entries are accepted and that invalid entries return an appropriate error.

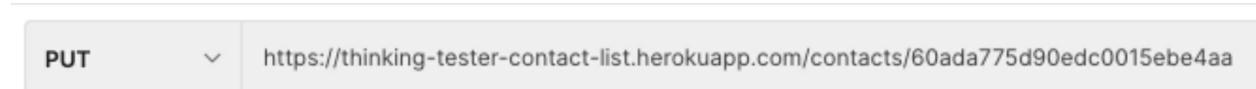
Finally, you'll want to test the security of the request. You shouldn't be able to add a contact when you are missing the bearer token, or when the token is invalid.

# Chapter 35: Testing PUT Requests

A PUT request is very similar to a POST request. The major difference is that POST requests are intended to create a new record, while PUT requests are intended to replace an existing record.

Let's create a PUT request using a contact you have already created, either through the UI or through the API. Run your Get Contact List request to see your list of existing contacts and choose one of the IDs to use.

Now click the plus (+) button to create a new request. Change the HTTP verb in the dropdown to PUT. In the URL, add <https://thinking-tester-contact-list.herokuapp.com/contacts/<your contact id>>, replacing <your contact id> with the contact ID you want to use:



Next, let's add in the authorization. Click on the Authorization tab and choose Bearer Token from the Type dropdown. Then paste in your token from Chapter 32.

When you are creating the body of the request, it's important to remember that a PUT request will replace every single value of your existing contact with whatever you put in the body of the request. Imagine for a moment that you had each contact written on a note card instead of in a database. When you do a PUT request, it's like taking that note card and replacing it with a brand-new note card.

So, if you have a contact with information like this:

```
{  
  "firstName": "John",  
  "lastName": "Doe",  
  "birthdate": "1974-07-04"
```

```
}
```

and you do a PUT request for that contact, like this:

```
{
  "firstName": "Jon",
  "lastName": "Doe"
}
```

you will have a contact with a first name and a last name but no birth date.

Now that you understand this, let's add in the body of the request. Begin by clicking on the Body tab; then choose the “raw” button and then JSON from the Type dropdown.

For our first request, let's create a body that has a value for every field. Add this to the request body:

```
{
  "firstName": "Amy",
  "lastName": "Miller",
  "birthdate": "1992-02-02",
  "email": "amiller@fake.com",
  "phone": "8005554242",
  "street1": "13 School St.",
  "street2": "Apt. 5",
  "city": "Washington",
  "stateProvince": "QC",
  "postalCode": "A1A1A1",
  "country": "Canada"
}
```

We are replacing every existing value of the contact with a completely new value. Click the Send button and you should see that the contact has all the new values you just added.

Save the request by clicking the Save button, giving the request a name

such as “Update Contact”, and saving it to your Contact List API collection.

You can verify that the new values were saved to the database by running the Get Contact request with the same contact ID you used for the PUT request.

Now that you have your PUT request working, it’s time to test it. All the tests we did in the POST request will apply to the PUT request, but in this case we are replacing the record rather than creating it. First, you could do some tests where you have a record with all the fields and you do a PUT where you are only passing in the required fields. You should get a record with just the first name and last name as the response.

Next you could try doing a PUT with no first name or last name, but with other nonrequired fields, and verify that you get the appropriate error message. Similarly, you could make a request that has some fields but is missing the first name, or has some fields but is missing the last name, and you should get the appropriate error message.

Just as with a POST request, you should check to make sure all the field validation rules are working. So you should test with values that are longer than the allowed number of characters, and with invalid birth date, email, phone number, and postal code values.

There are three different types of value changes you can have with a field:

- A null value can be replaced with a value.
- One value can be replaced with another value.
- A value can be replaced with a null value.

You’ll want to test this out with every field in the Contact record. For example, if you were testing the birth date field, you could do the following:

1. Create a record that doesn’t have a birth date. Just leave that value out when doing your POST request. Make a note of the new contact ID you have created.

2. Do a PUT request that includes a value for the birth date. Make sure you are updating the right record by using the contact ID from Step 1.
3. Do a GET request to validate that the record has been updated with the birth date.
4. Do another PUT request that does not include a value for the birth date.
5. Do a GET request to validate that the birth date is now null.

As with the GET and POST requests, you can test that you get a 401 response when you try to run the PUT request without a valid token. You can also try passing in a PUT request with a contact ID that doesn't exist; you should get a 404 response.

# Chapter 36: Testing PATCH Requests

Like PUT requests, PATCH requests modify an existing record. But rather than replacing the entire record, a PATCH request simply alters the fields that are passed into the request. PATCH requests are less common than PUT requests. In the Contact List app, I have included a PATCH request in the API, but the UI does not have any functionality that uses it. In the UI, when we modify a record it calls the PUT request.

Let's set up a PATCH request. Use the plus (+) button to create a new request, and change the HTTP verb to PATCH. Add this URL to the request:

<https://thinking-tester-contact-list.herokuapp.com/contacts/<your contact id>>, once again replacing <your contact id> with the contact ID you want to update.

Add the authorization by clicking on the Authorization tab, choosing Bearer Token from the dropdown, and adding the token you created in Chapter 32.

Add the body of the request by clicking the Body tab, selecting “raw”, and choosing JSON from the Type dropdown. In the body of the request, simply add:

```
{  
  "firstName": "Anna"  
}
```

Save the request to your collection as “Partial Update Contact”. Then send the request. When your contact is returned in the response, you’ll see that it has all the same field values it had before, except that the first name is now Anna. You can try a GET request on this contact to make sure the contact has been saved correctly to the database.

You’ll want to test doing a PATCH with every type of contact field. You can try patching every field by itself and patching several fields at once. You should also test scenarios in which one of the contact fields is null and you

are updating it to have a value.

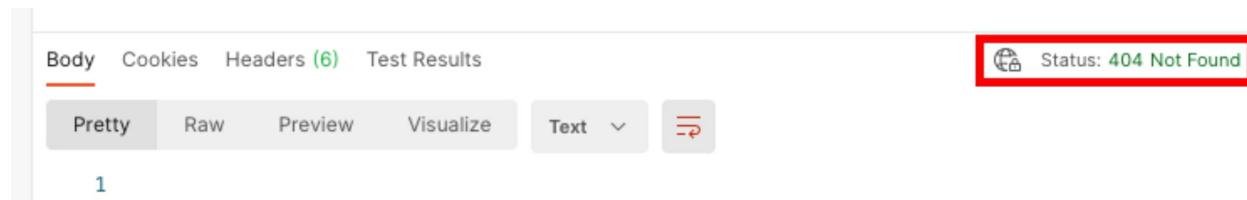
Just as you did with the POST and PUT requests, you'll want to test that validation works on all the different fields, making sure the string limits are respected and the birth date, email, phone, and postal code values are validated. Try doing a PATCH request with a contact ID that doesn't exist to make sure you get a 404 response. Finally, you'll want to make sure you can't do a PATCH request without a valid token.

# Chapter 37: Testing DELETE Requests

A DELETE request removes an entire record from a database table. To set up a DELETE request in your collection, click on the plus (+) button to add a new request, change the HTTP verb to DELETE, and add this URL: <https://thinking-tester-contact-list.herokuapp.com/contacts/<your contact id>>, replacing <your contact id> with the contact ID you want to delete.

As always, add the Authorization for your request. The request doesn't have a body, so simply save it to your collection as "Delete Contact", and your request will be ready to run.

Run the request by clicking the Send button. You should see "Contact Deleted" in the response. You can verify that the contact was deleted by running the Get Contact List request again and validating that a contact with that ID isn't in the list anymore. You can also try running the Get Contact request with that ID and you should receive a 404 Not Found response:



The screenshot shows a REST client interface with the following details:

- Header bar: Body, Cookies, Headers (6), Test Results.
- Toolbar: Pretty, Raw, Preview, Visualize, Text (dropdown), and a copy icon.
- Response status: Status: 404 Not Found (highlighted with a red box).
- Response body: A single item labeled '1'.

Testing a DELETE request is fairly straightforward. Since the only information you are passing into the request is the ID of the record you want to delete, there's not much room for variation. You can test what happens when you enter an ID for a record that doesn't exist (you should get a 404 response) and what happens when you enter an invalid ID such as "FOO". And as with the other requests in the Contact List API, you can try sending a request with a missing or invalid token and verify that you get a 401 response.

# Chapter 38: Response Codes

In the past several chapters, we looked at how to test different types of REST requests. Each time we made a request, we received a three-digit response code. In this chapter we'll look at many different types of response codes and what they mean.

Every REST request gets a three-digit code in its response. The code conveys information about the response; for example, whether the response was successful or unsuccessful, or whether another event is taking place. The response codes are grouped into levels to keep things organized. Following are some of the most common responses and what they mean.

## 100-Level Responses

A 100-level response indicates that the request should continue. The most common 100-level response type is 100 Continue. This can be used with large requests; it gives the server the opportunity to stop a large request before too much data is transmitted. You probably won't see this in your API testing, because the server response will continue and complete behind the scenes, and will then return a 200-level response.

## 200-Level Responses

A 200-level response indicates that the request was successful. The most common response is 200 OK. This simply means everything went as expected. Here are some other common 200-level responses:

- **201 Created:** This indicates that a new resource has been created as the result of the request. POST requests often return a 201 response because they often create a new record in a database.
- **202 Accepted:** This indicates that the request was accepted but is not complete yet. You could use this response for a pending change that needs additional approval before being added to the database.

- **204 No Content:** This means the request was processed successfully and no data was returned. This might be used with a PUT request, where the content is changed but the developer sees no need to return the data with the response. A 200 OK response can return no data if the developer chooses, but a 204 response should never return any data.

## 300-Level Responses

A 300-level response indicates that a resource has been moved. The most common of the 300-level responses is 301 Moved Permanently. This response should include the new URI in the header so that the client will know where to point the request next time.

## 400-Level Responses

A 400-level response indicates that there was something wrong with the client's request. The most common of these is 400 Bad Request, which is usually used when the request is malformed or inaccurate in some way. Examples of this would be a request where there is required data that is not present, or data that has some sort of validation error. Other common 400-level responses include the following:

- **401 Unauthorized:** This is usually returned when the client does not have the appropriate authentication to make the request, such as a JWT or a cookie.
- **403 Forbidden:** This is returned when the client has the appropriate authentication to make the request but does not have the permission to view the resource. For example, a user might be logged in to the system and be able to request their own data but should not be able to request another user's data.
- **404 Not Found:** This is returned when the client is making a request for a specific resource and the server cannot find it. An example of this is requesting data for a customer with an ID of 100, and there is no customer with an ID of 100 in the database. 404 is also sometimes used when the user making the request doesn't have

permission to view the resource. This is for extra security: a 403 lets the user know the resource exists and they don't have permission to view it; a 404 keeps the existence of the resource a secret.

- **409 Conflict:** This is returned when the request puts data resources in conflict with one another. One example of this is a client attempting a POST request to create a resource with an ID that is already being used.

## 500-Level Responses

A 500-level response means something has gone wrong on the server side of the request. The most common is the 500 Internal Server Error response, which can be used for a variety of problems. An example of this is a request that is attempting to add a record to a database whose database table is not equipped to handle it, because it has too many characters or is the wrong type. Other common 500-level responses include the following:

- **502 Bad Gateway:** This can happen when the responding server needs to make a request from another server and the other server is returning an invalid response.
- **503 Service Unavailable:** This is returned when the responding server is temporarily down for some reason. This response can be more helpful than the generic 500 response because it indicates that the problem is with server availability rather than with the database.

# Chapter 39: Postman Assertions

One of the great benefits of using Postman for API testing is that it allows you to create assertions on the response you receive from your requests. This enables you to create API tests which you can run manually or through automation.

In this chapter we'll look at five different types of assertions to use in API testing and create some examples of each. If you followed the steps listed in Chapters 32 through 37, you should have a collection with each type of REST request:

## ✓ Contact List API ☺

**POST** Login

**GET** Get Contact List

**GET** Get Contact

**POST** Add Contact

**PUT** Update Contact

**PATCH** Partial Update Contact

**DEL** Delete Contact

We'll use this collection to create some assertions and turn our Postman requests into Postman tests.

### Status Code Assertions

A status code assertion validates that the response you received from a request is the response you were expecting.

Let's create a status code assertion on our Get Contact List request. Click on the name of the request to select it. Then click on the Tests tab:

The screenshot shows the Postman interface with a 'GET' request to 'https://thinking-tester-contact-list.herokuapp.com/contacts'. The 'Tests' tab is highlighted with a red box. Below the tabs, there is a snippet of code: '1'.

When you click on the Tests tab, a special section called SNIPPETS opens on the right side of the Postman window:

Test scripts are written in JavaScript, and  
are run after the response is received. >

[Learn more about tests scripts](#)

## SNIPPETS

[Get an environment variable](#)

[Get a global variable](#)

[Get a variable](#)

[Get a collection variable](#)

The SNIPPETS section contains snippets of code that can be used to quickly create assertions. When you click on a snippet, code will be added to

the Tests window. Scroll down in the SNIPPETS section until you see “Status code: Code is 200” and click on it. It will add this code:



The screenshot shows the Postman interface with the 'Tests' tab selected. Below the tabs, there is a code editor containing the following JavaScript snippet:

```
1 pm.test("Status code is 200", function () {  
2     pm.response.to.have.status(200);  
3});
```

This is JavaScript, and here's a breakdown of what it means:

- pm refers to “Postman” and indicates that this is a Postman command.
- "Status code is 200" is the name of the test.
- pm.response.to.have.status(200) is the test itself; it is an assertion that the response status came back as 200.

Since we are expecting that our Get Contact List request will return a 200 response code, our test is ready to run! Save the request and then click the Send button. In the response window, you should see a Test Results tab with a (1/1) next to it in green:



This indicates that 1 of 1 tests passed. If you click on the Test Results tab, you will see the name of the test you created with the word “PASS” beside it:

Body Cookies Headers (8) Test Results (1/1)

All

Passed

Skipped

Failed

PASS

Status code is 200

Let's see what it looks like when a test fails. Go to the Authorization tab and remove your Bearer token. Then click Send to run the test again. In the Test Results tab, you'll see the following:

Body Cookies Headers (8) Test Results (0/1)

🔒 Status: 401 Unauthorized

All

Passed

Skipped

Failed

FAIL

Status code is 200 | AssertionError: expected response to have status code 200 but got 401

In addition to indicating that the test failed, it also shows what the expected and actual results were.

## Response Body Assertions

Response body assertions validate that a certain word or phrase was included in the body of the response. You can assert that the entire body of the response is correct, or you can assert that the response body contains a specific string.

Let's create an assertion that will check for a specific string. So far our collection only has happy path tests saved. Let's create a request that will result in a 400 response. Hover over the Add Contact request and click on the three-dot menu that appears:

The screenshot shows the 'Contact List API' collection in Postman. On the left, a list of requests is visible: 'Login' (POST), 'Get Contact List' (GET), 'Get Contact' (GET), 'Add Contact' (POST), 'Update Contact' (PUT), 'Partial Update Contact' (PATCH), and 'Delete Contact' (DEL). On the right, details for the 'Add Contact' request are shown: method 'POST', status '4', and a three-dot menu icon (three horizontal dots) which is highlighted with a red box. Below the menu are buttons for 'Params' and 'Auth', and radio buttons for 'none' and 'for'.

From that menu, choose Duplicate. This will create a copy of your existing request. Click on the Add Contact Copy request, choose the three-dot menu again, and choose Rename. Rename the request “Add Contact Missing First Name”.

Select the Add Contact Missing First Name test in the center pane of the Postman window, click the Body tab, and make a change to the body tab so that the firstName entry is no longer there:

[Params](#)[Authorization](#)[Headers \(10\)](#)[Body](#) none form-data x-www-form-urlencoded

```
1  {
2    "lastName": "Doe",
3    "birthdate": "1970-01-01",
4    "email": "jdoe@fake.com",
5    "phone": "8005555555",
6    "street1": "1 Main St.",
7    "street2": "Apartment A",
8    "city": "Anytown",
9    "stateProvince": "KS",
10   "postalCode": "12345",
11   "country": "USA"
12 }
```

Save the request and run it. You'll see that you are now getting a response message that says the first name field is required:

```
1  {
2      "errors": {
3          "firstName": {
4              "name": "ValidatorError",
5              "message": "Path `firstName` is required.",
6              "properties": {
7                  "message": "Path `firstName` is required.",
8                  "type": "required",
9                  "path": "firstName"
10             },
11             "kind": "required",
12             "path": "firstName"
13         },
14     },
15     "_message": "Contact validation failed",
16     "message": "Contact validation failed: firstName: Path `firstName` is required."
17 }
```

To create an assertion for this negative test, we'll want to make sure the response includes the statement that the first name is required. Click on the Tests tab, and then click on the “Response body: Contains string” snippet. The snippet will add code to the Tests window:

```
1  pm.test("Body matches string", function () {
2      pm.expect(pm.response.text()).to.include("string_you_want_to_search");
3  });
```

Let's give our test a name that makes sense. Change "Body matches string" to "Missing first name error is returned". Then change the "string\_you\_want\_to\_search" field to "Path `firstName` is required.":

```
1  pm.test("Missing first name error is returned", function () {
2      pm.expect(pm.response.text()).to.include("Path `firstName` is required.");
3  });
```

Save your request and then run it. You should see in the Test Results tab that the test has run and passed:

The screenshot shows the Postman interface with the 'Test Results' tab selected. There is one test result listed:

| Result | Description                          |
|--------|--------------------------------------|
| PASS   | Missing first name error is returned |

## JSON Data Assertions

JSON data assertions are really powerful because they let you assert that a specific value for a specific field has been saved to the database as you expected.

To create a JSON data assertion, we'll use the Get Contact request. Imagine that we've just created a new contact with the last name "Doe", and we want to assert that when we request the new contact, we get a contact with the correct last name.

Click on the Get Contact request, then click on the Tests tab. Next, click on the "Response body: JSON value check" snippet. You should see the snippet added to the Tests window:

```
1 pm.test("Your test name", function () {  
2     var jsonData = pm.response.json();  
3     pm.expect(jsonData.value).to.eql(100);  
4 });
```

To do a JSON value check, the response needs to be parsed as JSON. That's what this line of code does: `var jsonData = pm.response.json()`. It's taking the Postman response, converting it to JSON, and then saving the converted response as a variable called `jsonData`.

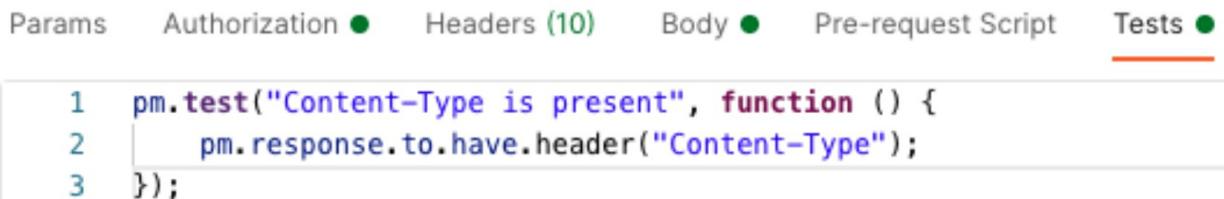
Change the test name to "Correct last name is returned". Then change value to `lastName` because this is the field we are checking. Finally, change 100 to the last name of your contact that you are testing with. Put the last name in quote marks. Your test should now look like this:

```
1 pm.test("Correct last name is returned", function () {
2     var jsonData = pm.response.json();
3     pm.expect(jsonData.lastName).to.eql("Doe");
4});
```

Save the request and click Send. You should see your test has run and passed.

## Header Assertions

Header assertions are less common than Status Code, Response Body, and JSON Data assertions, but they can be helpful when doing security testing. For example, you might want to assert that the response you are getting from a request is JSON, and not some other response type which might indicate that the request has been tampered with. We'll add this assertion to our Partial Update Contact. Click on that request and then on the Tests tab. Then click on the "Response headers: Content-Type header check" snippet. The snippet is currently checking that a Content-Type header is present:



The screenshot shows the Postman interface with the "Tests" tab selected. Below the tabs, there is a code editor containing a snippet for asserting the presence of a Content-Type header.

```
1 pm.test("Content-Type is present", function () {
2     pm.response.to.have.header("Content-Type");
3});
```

Let's change the assertion so that it's actually checking for a specific content type. First, change the test name from "Content-Type is present" to "Content-Type is application/json". Next we'll change the assertion value. Put a comma after "Content-Type" and add "application/json; charset=utf-8" because this is actually what's getting returned in the Content-Type header:

```
1 pm.test("Content-Type is application/json", function () {  
2     pm.response.to.have.header("Content-Type", "application/json; charset=utf-8");  
3 });
```

Note that the first part of the assertion names the type of header we are expecting (Content-Type), and the second part names the value of Content-Type we are expecting (application/json; charset-utf-8).

Save and send the request, and you should see it pass. In the future, if the request ever returns something other than application/json, you'll see the test fail and you'll be alerted to a potential problem.

## Response Time Assertions

This last assertion type validates that a request doesn't take too long. This is a helpful test to run to make sure your API is performing well.

Let's add a response time assertion to our Get Contact List request. We already have one assertion on this request, but we can add a second assertion as well. Click on the Get Contact List request, then click on the Tests tab. Make sure your cursor is below your first test in the Tests window. Next, click on the "Response time is less than 200ms" snippet. The response time assertion will be added to the existing test:

```
1 pm.test("Status code is 200", function () {  
2     pm.response.to.have.status(200);  
3 });  
4  
5 pm.test("Response time is less than 200ms", function () {  
6     pm.expect(pm.response.responseTime).to.be.below(200);  
7 });
```

Save and run the test. Assuming your request comes back in less than 200 milliseconds, you should see that your two tests passed. You can also adjust the test to be 500 milliseconds or anything else that seems more appropriate. Remember to change both the name of the test and the expected value:

```
pm.test("Response time is less than 500ms", function () {  
    pm.expect(pm.response.responseTime).to.be.below(500);  
});
```

---

With these five assertion types, you can create a wide variety of tests to run against an API.

# Chapter 40: Using Variables in Postman

Using variables in Postman is a really helpful way to make your collections more efficient and reliable. To learn more about variables, we'll continue using the Contact List API collection we created in previous chapters.

The first thing to understand about variables in Postman is that they are organized into environments. A Postman environment is simply a collection of variables that can be used to run against a Postman collection.

Creating an environment is easy. Click on the Environments tab in the left margin of the Postman window:

The screenshot shows the Postman application interface. On the left, there is a sidebar with the following items:

- Collections
- APIs
- Environments** (This item is highlighted with a red box.)
- Mock Servers
- Monitors
- History

The main area displays a list of API endpoints under the "Contact List API" collection. The endpoints are:

- POST Login**
- GET Get Contact List**
- GET Get Contact**
- POST Add Contact**
- POST Add Contact Missing First Name**
- PUT Update Contact**
- PATCH Partial Update Contact**
- DEL Delete Contact**

Click the Create Environment button:



## You don't have any environments.

An environment is a set of variables that allows you to switch the context of your requests.

### Create Environment

Give your environment a name, such as “Contact List- PROD”:

Contact List- PROD

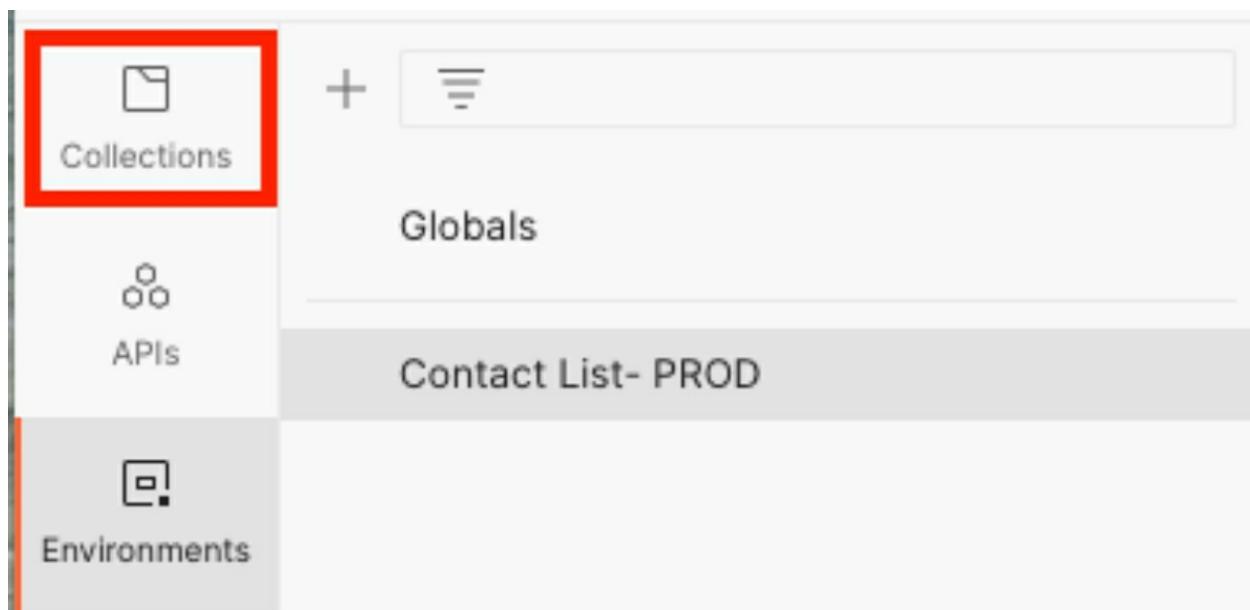
|  | VARIABLE           | INITIAL VALUE <small>①</small> | CURRENT VALUE <small>①</small> |
|--|--------------------|--------------------------------|--------------------------------|
|  | Add a new variable |                                |                                |

Add the variable “contactId” to the environment file:

### Contact List- PROD

|                                     | VARIABLE  | INITIAL VALUE <small>(i)</small> | CURRENT VALUE <small>(i)</small> |
|-------------------------------------|-----------|----------------------------------|----------------------------------|
| <input checked="" type="checkbox"/> | contactId |                                  |                                  |
| Add a new variable                  |           |                                  |                                  |

Click the Save button. We could populate the value of contactId by typing the value directly into the environment, but it's more powerful to set the variable programmatically. Return to your collection by clicking on the Collections tab in the left side of the window:



Now we'll need to set our environment to the one we just created. Click the No Environment dropdown in the upper-right of the window:

No Environment



Fork



Save



Share

...

Select your new environment from the dropdown:

Contact List- PROD



Fork



Save



Share

...

Now we'll add a code snippet to the Add Contact request which will extract and save the contactId whenever we add a new contact. Click on the Add Contact request and then on the Tests tab. (Note that extracting a variable is not actually a test; there are other scripts we can run on the Tests tab besides assertions.)

Click on the snippet called "Set an environment variable" to add it to the Tests window:

```
1 pm.environment.set("variable_key", "variable_value");
```

"variable\_key" refers to the name of our variable. Since we want to set

contactId as a variable, we'll replace "variable\_key" with "["contactId"](#)".

Next, we need to set the variable value. We will be extracting a JSON value from the response, so we actually need to add another command above the existing one:

```
var jsonData = pm.response.json\(\)
```

In the preceding chapter, we discussed how this command takes the request response, parses it as JSON, and then saves it to a variable called jsonData, enabling us to use jsonData to find the value we need.

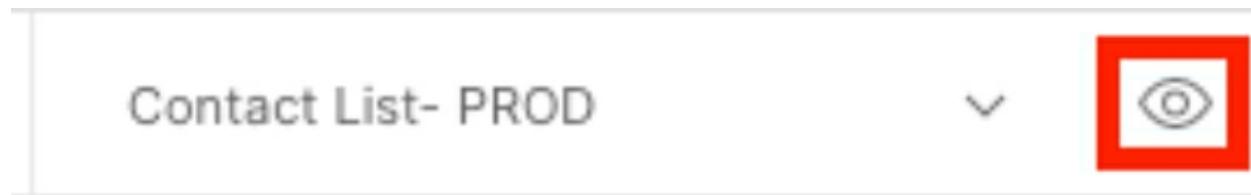
Here, we will replace "variable\_value" with [jsonData.\\_id](#) (note that it has no quotation marks around it because it's a variable, not a string). \_id is the name of the contact ID that is returned when we add a new contact. So your script should now look like this:

---

```
1 var jsonData = pm.response.json\(\);
2 pm.environment.set\("contactId", jsonData.\_id\);
```

---

Save and run your request. Now click on the eye icon in the upper-right of the window:



A window will open and you'll see that your contact ID has been saved to the contactId variable:

| Contact List- PROD |               | <a href="#">Edit</a>     |
|--------------------|---------------|--------------------------|
| VARIABLE           | INITIAL VALUE | CURRENT VALUE            |
| contactId          |               | 60b5323a4f0f1000159d5739 |

Now that we have our variable saved, let's use it in a couple of different ways. First, we'll use it in the URL of a request. Click on the Get Contact

request and replace the last part of the URL with “{{contactId}}”:

The screenshot shows the Postman interface. At the top, it says "Contact List API / Get Contact". Below that, there's a "GET" method button and a dropdown arrow. To the right of the method is the URL: "https://thinking-tester-contact-list.herokuapp.com/contacts/{{contactId}}".

Save and run the request. You'll see that the contact that is returned is the one you just added.

You can also use a variable in a test. Let's add a test to Get Contact that verifies the contactId to confirm that the correct contact has been returned. We already have a test in place that is testing that the last name is returned. Let's copy that existing test and paste it below the original:

```
1 pm.test("Correct last name is returned", function () {  
2     var jsonData = pm.response.json();  
3     pm.expect(jsonData.lastName).to.eql("Doe");  
4 });  
5  
6 pm.test("Correct last name is returned", function () {  
7     var jsonData = pm.response.json();  
8     pm.expect(jsonData.lastName).to.eql("Doe");  
9 });|  
10
```

Change the name of the test in line 6 to be "Correct contact is returned". Then change jsonData.lastName in line 8 to jsonData.\_id. Finally, change "Doe" to environment.contactId:

```
pm.test("Correct last name is returned", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData.lastName).to.eql("Doe");
});

pm.test("Correct contact is returned", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData._id).to.eql(environment.contactId);
});
```

When you run this test, it compares the `_id` value that was passed back in the body of the response to the value you have set for `contactId` in the environment. Save your test and make sure it runs and passes.

You can use variables in your Postman requests in two additional ways. One way is to pass them in with Authorization headers or other headers. Here's an example of how you can pass in a saved authorization token:

The screenshot shows the 'Authorization' tab in the Postman interface. The 'Type' dropdown is set to 'Bearer To...'. The 'Token' field contains the placeholder code `{{token}}`. A note below explains that the authorization header will be automatically generated when the request is sent. A link to 'Learn more about authorization' is also present.

You can also use variables in the body of a request. Here's an example of how you can pass in a `lastName` variable in the body of your Add Contact request:

```
{  
    "firstName": "John",  
    "lastName": "{{lastName}}",  
    "birthdate": "1970-01-01",  
    "email": "jdoe@fake.com",  
    "phone": "8005555555",  
    "street1": "1 Main St.",  
    "street2": "Apartment A",  
    "city": "Anytown",  
    "stateProvince": "KS",  
    "postalCode": "12345",  
    "country": "USA"  
}
```

Of course, for both of these examples, you'd need to first create the environment variable in your environment file.

Variables make it easy to pass values without having to copy and paste them again and again. And if you need to make a change to a variable, you can make the change in the environment file and know that it will be changed throughout your collection.

# Chapter 41: Organizing Your API Tests

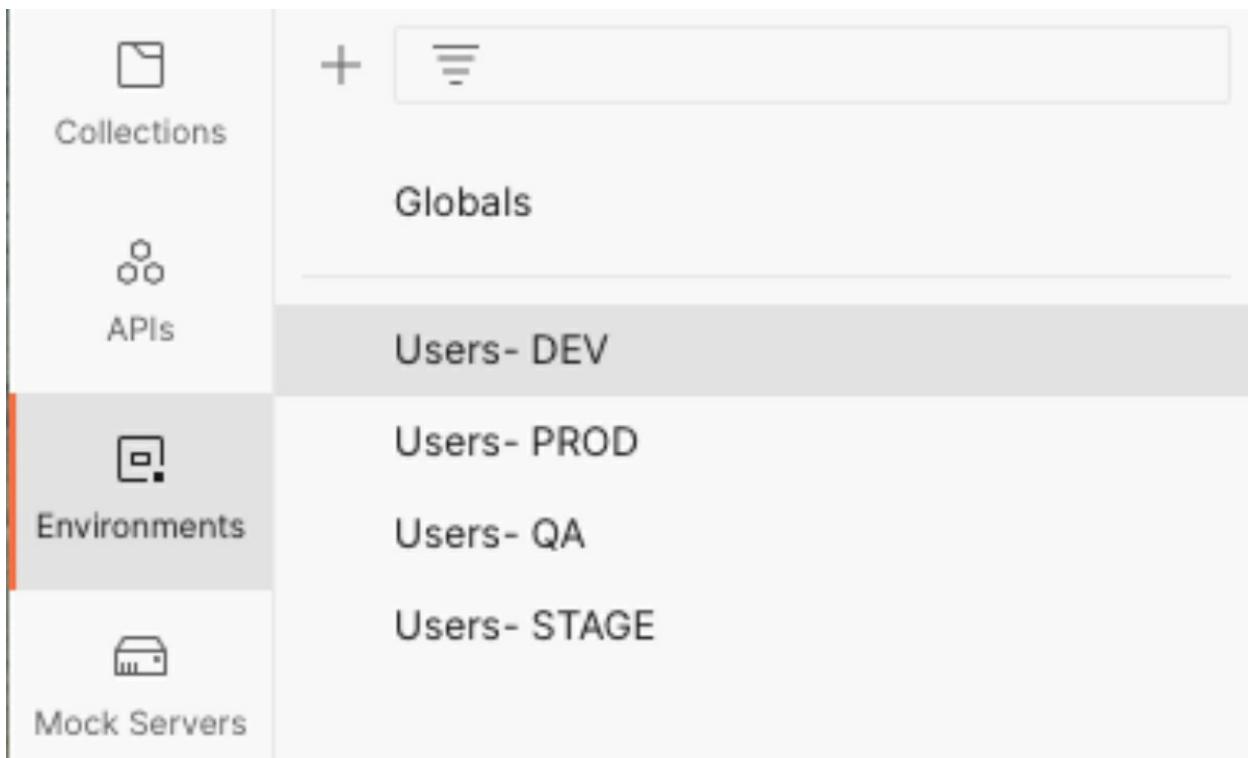
One of the things I love about API testing is how easy it is to organize tests and environment variables. I love having test suites ready at a moment's notice to run at the click of a button when regression testing is needed, or to run automatically as part of continuous integration.

This chapter covers some organizational patterns you can use for your API tests. I discuss them in the context of Postman, but the concepts are similar no matter what API testing platform you are using.

Let's begin with environments. As you'll recall from the preceding chapter, an environment is a collection of variables in Postman. There are two different ways I like to set up my Postman environments. To explain them, I'll use two scenarios. For both scenarios, let's assume I have an application that begins its deployment life cycle in development, then moves to QA, then staging, and then production.

In my first scenario, I have a Users API that gets and updates information about all the users on my website. In each product environment (development, QA, staging, and production), the test users will be different. They'll have different IDs and different first and last names. The URLs for the product environments will each be different as well. However, my tests will be exactly the same; in each product environment, I'll want to GET a user and PUT a user update.

So I will create four different Postman environments:



In each of my four environments, I'll have these variables:

- environmentURL
- userId
- firstName
- lastName

Then my test collection will reference those variables. For example, I could have a test request that looks like this:

```
GET https://{{environmentURL}}/users/{{userId}}
```

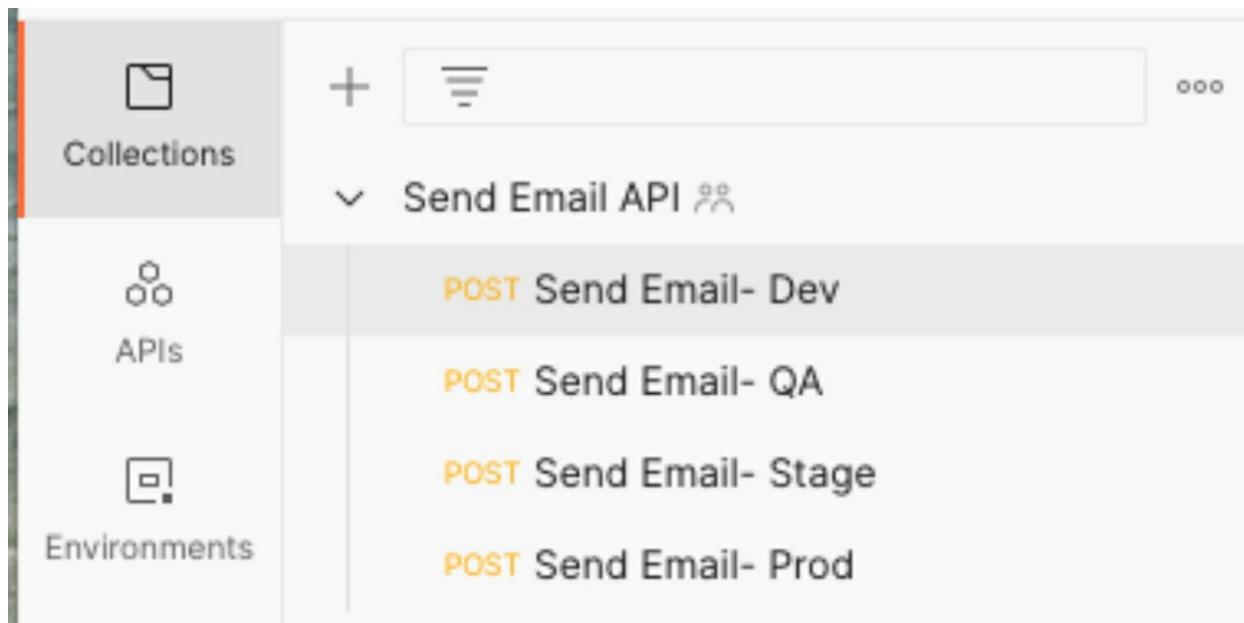
Which environmentURL is called and which userId is used will depend on which Postman environment I am using. With this strategy, it's easy for me to switch from the Dev environment to the QA environment, or to any other environment. All I have to do is change the Postman environment setting and run the same test again.

I use the second scenario when I have a function that delivers an email, and the function uses the same URL regardless of the product environment. I like to pass in a timestamp variable which will show the current time and will

stay the same regardless of what environment I am using. But I like to change the content of the email depending on what product environment I am in. In this case, I am creating only one Postman environment: “Email Test”.

My Postman test has only one variable: timestamp.

My test collection, however, has one test for each product environment. Here are my tests:



Each request includes the timestamp variable, but what is sent in the body of the email varies. For the Dev environment I use a request that contains “Dev” in the message body, for the QA environment I use a request that contains “QA” in the message body, and so on.

When deciding which of these two environment strategies to use, consider the following:

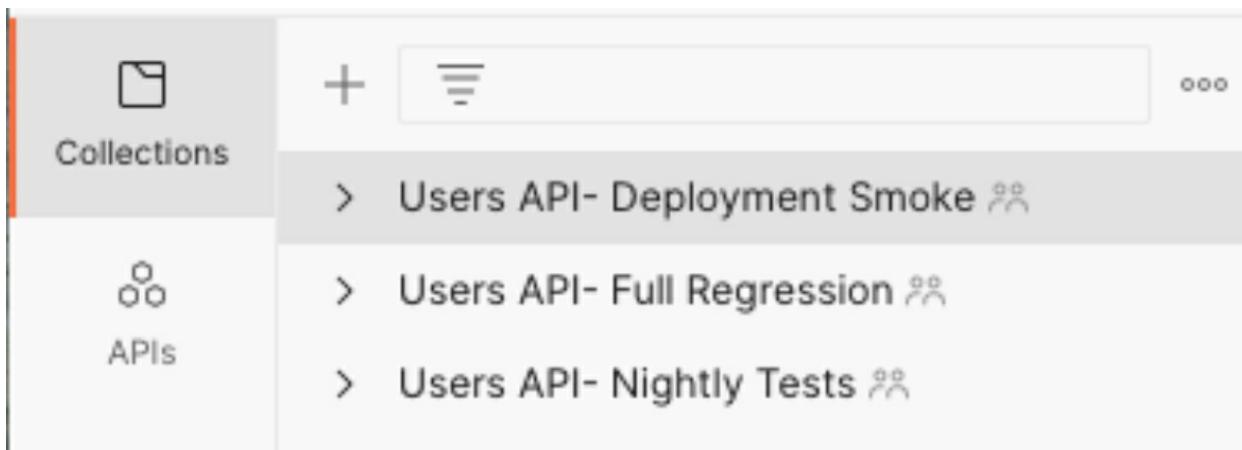
- What stays the same from one product environment to another?
- What changes from one product environment to another?

If many variables change, you may want to consider setting up multiple Postman environments, as in my first scenario.

If only one or two things change from one environment to the next, and if

the URL doesn't change, you may want to use my second scenario, which has just one Postman environment but different requests for each product environment.

Now let's talk about ways to organize our tests. First, let's think about test collections. The most obvious way to organize collections is by API. If you have more than one API in your application, you can create one collection for each API. You can also create collections based on test function. For example, if I have a Users API and I want to run a full regression suite, a nightly automated test, and a deployment smoke test, I could create three collections, like this:



Finally, let's think about test folders. Postman is so flexible in this area; you can use any number of folders in a collection, and you can also use subfolders.

Here are some suggestions for how you can organize your tests into folders:

- **By type of request:** All your POST requests go in one folder, all your GET requests go in another folder.
- **By endpoint:** GET myapp/users requests go in one folder, GET myapp/users/userId requests go in another folder.
- **By result expected:** GET myapp/users happy path requests go in one folder, GET myapp/users negative requests go in another folder.
- **By feature:** GET myapp/users requests with a sort function go in one folder, GET myapp/users requests with a filter function go in

another.

As with all organizing efforts, the purpose of organizing your tests and environments is to ensure that they can be used as efficiently as possible. By looking at the types of tests you will be running and the variations in the environments where you will be running them, you can organize your Postman environments, collections, and folders so that you have all the tests you need at your fingertips.

# Chapter 42: Understanding JSON Data

New API testers are often mystified by the assortment of curly braces, colons, and commas they see in the body of the response to their GET requests. Trying to create a valid JSON body for a POST request is even more puzzling. This chapter covers how JSON data is formed and offers some resources that will make working with JSON easier.

JSON stands for JavaScript Object Notation. It's simply a way to organize data so that it can easily be parsed by the code. The fundamental building block in JSON is the name-value pair. Here are some examples of name-value pairs:

```
"Name": "Dino"  
"Color": "Purple"
```

Multiple name-value pairs are separated by commas, like this:

```
"FirstName": "Fred",  
"LastName": "Flintstone",  
"City": "Bedrock"
```

Note that the final name-value pair does not have a comma. This is because it's at the end of the group.

An object is simply a grouping of one or more name-value pairs. The object is represented with curly braces surrounding the name-value pairs. For example, we might represent a pet object like this:

```
{  
  "Name": "Dino",  
  "Type": "Dinosaur",  
  "Age": "5",  
  "Color": "Purple"  
}
```

An array is a group of objects. The array is represented with square braces, and the objects inside the array have curly braces. For example:

```
"residents": [  
  {  
    "FirstName": "Fred",  
    "LastName": "Flintstone"  
  },  
  {  
    "FirstName": "Barney",  
    "LastName": "Rubble"  
  },  
  {  
    "FirstName": "Wilma",  
    "LastName": "Flintstone"  
  }  
]
```

Fred Flintstone's last name does not have a comma after it. This is because LastName is the last name-value pair in the object. But the object that contains Fred Flintstone does have a comma after it because there are more objects in the array. And the object that contains Wilma Flintstone does not have a comma after it, because it is the last object in the array.

Not only can an array contain objects, but an object can contain an array. When you are sending JSON in the body of an API request, it will always be in the form of an object, which means it will always begin and end with a curly brace. Also, name-value pairs, objects, and arrays can be very deeply nested. It would not be unusual to see something like this contained in a POST for city data:

```
{  
  "residents": [  
    {  
      "firstName": "Fred",  
      "lastName": "Flintstone",  
      "contactInfo": {  
        "phoneNumber": "555-867-5309",  
        "email": "fred@flintstones.com"  
      }  
    }  
  ]  
}
```

```

        "email": "fflintstone@slaterock.com"
    }
},
{
    "firstName": "Wilma",
    "lastName": "Flintstone",
    "contactInfo": {
        "phoneNumber": "555-423-4545",
        "email": "wflinstone@dailygranite.com"
    }
},
],
"pets": [
    {
        "name": "Dino",
        "type": "dinosaur",
        "color": "purple"
    },
    {
        "name": "Hoppy",
        "type": "hopperoo",
        "color": "green"
    }
]
}

```

Notice that contactInfo is deeply nested in the city object. If we were testing this API and we wanted to assert that Fred Flintstone's phone number was correct, we would access it like this:

```
residents[0].contactInfo.phoneNumber
```

The first array in the city object is the residents array, and Fred is the first resident in the array, so we access him with residents[0]. Next, we move to contactInfo, and since contactInfo is an object rather than an array, we don't need to specify a number in braces. Finally, we specify phoneNumber as the name-value pair within the contactInfo object we are looking for.

Even with the preceding explanations, you may find working with JSON objects frustrating. There are two JSON-related tools that can help you when working with JSON:

- **JSON validator:** When you paste your JSON into this tool, it will tell you whether or not it is valid JSON. A good validator will tell you exactly where your syntax is incorrect.
- **JSON pretty-printer:** If you come across JSON that is returned in a response and it is not using line breaks and indentations, you can paste it into this tool and it will insert those breaks and indentations for you, making it easier to read.

# Chapter 43: API Contract Testing Made Easy

As software becomes increasingly complex, more and more companies are turning to APIs as a way to organize and manage their application's functionality. Instead of being one monolithic application in which all changes are released at once, software can now consist of multiple APIs that are dependent upon one another but can be released separately at any time. Because of this, it's possible to have a scenario in which one API releases new functionality which breaks a second API's functionality, because the second API was relying on the first and now something has changed.

You can prevent this from happening by using API contract tests. These can seem confusing: which API sets up the tests and which API runs them? In this chapter I'll use a simple example to show you how contract testing works.

Let's imagine we have an online store that sells Super Balls. The store sells Super Balls of different colors and sizes, and it uses three different APIs to accomplish its sales tasks:

**Inventory API:** This API keeps track of the Super Ball inventory to make sure orders can be fulfilled. It has the following endpoints:

- `/checkInventory`, which passes in a color and size and verifies that the ball is available
- `/remove`, which passes in a color and size and removes that ball from the inventory
- `/add`, which passes in a color and size and adds that ball to the inventory

**Orders API:** This API is responsible for taking and processing orders from customers. It has the following endpoints:

- `/addToCart`, which puts a ball in the customer's shopping cart

- [/placeOrder](#), which completes the sale

**Returns API:** This API is responsible for processing customer returns. It has the following endpoint:

- [/processReturn](#), which confirms the customer's return and starts the refund process

Both the [Orders API](#) and the [Returns API](#) are dependent on the [Inventory API](#) in the following ways:

- When the [Orders API](#) processes the [/addToCart](#) command, it calls the [/checkInventory](#) endpoint to verify that the type of ball that's been added to the cart is available.
- When the [Orders API](#) processes the [/placeOrder](#) command, it calls the [/remove](#) command to remove that ball from the inventory so that it can't be ordered by someone else.
- When the [Returns API](#) runs the [/processReturn](#) command, it calls the [/add](#) command to return that ball to the inventory.

In this example, the [Inventory API](#) is the producer and the [Orders API](#) and [Returns API](#) are the consumers.

It is the consumer's responsibility to provide the producer with some contract tests to run whenever the producer makes a code change to its API. So in our example, the team that works on the [Orders API](#) would provide contract tests like the following to the team that works on the [Inventory API](#):

- [/checkInventory](#), where the body contains { "color": "purple", "size": "small" }
- [/remove](#), where the body contains { "color": "red", "size": "large" }

The team that works on the [Returns API](#) would provide an example like this to the team that works on the [Inventory API](#):

- [/add](#), where the body contains { "color": "yellow", "size": "small" }

Now the team that works on the [Inventory API](#) can add those examples to

its suite of tests.

Let's imagine the Super Ball store just had an update to its inventory. There are now two different kinds of bounce levels for the balls: medium and high. So the **Inventory API** team needs to make some changes to its API to reflect this. Now a ball can have three properties: color, size, and bounce.

The **Inventory API team** modifies its `/checkInventory`, `/add`, and `/remove` commands to accept the new bounce property. But the developer accidentally makes “bounce” a required field for the `/checkInventory` endpoint.

After the changes are made, the contract tests are run. The `/checkInventory` test contributed by the **Orders API team** fails with a 400 error because there's no value for “bounce”. When the developer sees this, she finds her error and makes the bounce property optional. Now the `/checkInventory` call will pass.

Without these contract tests in place, the team working on the **Inventory API** might not have noticed that the change was going to break the **Orders API**. If the change went to production, no customer would be able to add a ball to their cart!

You should now understand through this simple example the importance of contract testing and the responsibilities of each API team when setting up contracts.

## Part V: Mobile Testing

# Chapter 44: The 12 Challenges of Mobile Testing

The first iPhone was released in 2007. Today smartphones are ubiquitous. Our smartphones are like our Swiss Army knives: they are our maps, our address books, our calendars, our cameras, our music players, and of course, our communication devices. Testing software would not be complete without testing on mobile.

Following are 12 reasons why testing on mobile is difficult. I thought it would be fun to illustrate what can go wrong with mobile software by describing a bug I've found in each area. I found some of these bugs in the course of my testing career and some on my personal device as an end user.

## **Challenge #1: Carriers**

Mobile application performance can vary depending on what carrier the device is using. In the United States, the two major carriers are Verizon and AT&T, and we also have smaller carriers like T-Mobile. Some of the major carriers in Europe are Orange, Vodafone, and SFR, and in Asia they include China Mobile, NTT, and China Telecom. When testing software on mobile, it's important to consider what carriers your end users will be using and to test with those carriers.

Example bug: I once tested a mapping function within an application and discovered that while the map would update based on my location when I was using one carrier, it would not update when I was using a different carrier. This was due to how the location was cached after a cell tower ping.

## **Challenge #2: Network or Wi-Fi**

Device users have the choice of using their applications while connected to the carrier's network or while on Wi-Fi. They can even change how they are connecting in the middle of using the application, or their connection can be cut if they go out of network range. It's important to test an application

when connected to a network and when connected to Wi-Fi, and to see what happens when the connection changes or is lost.

Example bug: I have a Wi-Fi extender in my house. I have seen a recurring bug with my music player: when I switch my phone's Wi-Fi connection to use the extender's IP, the player thinks I am offline. I then have to force-close the app and reopen it for the player to recognize that I am online.

### **Challenge #3: Application Type**

Mobile applications can be web based, native, or a hybrid of the two (developed like a web app but installed like a native app). Some of your end users will choose not to use a native or hybrid app and will prefer to interact with your application in their phone's browser. A variety of mobile browsers could also be used, such as Safari, Chrome, or Opera. So it's important to make sure your web application works well on a variety of mobile browsers.

Example bug: Many times I've gone to a "mobile-optimized" site that doesn't have the functionality I need. I've had to choose to go to the full site, where all the text is tiny and navigation is difficult.

### **Challenge #4: Operating System**

Mobile applications will function differently depending on the operating system. The two major operating systems are iOS and Android. It's important to test on whatever operating systems your end users will be using, to make sure all the features in the application are supported in all systems.

Example: This is not a bug, but a key difference between Android and iOS. Android devices have a back button and iOS devices do not. Applications written for iOS need to have a back button included on each page so that users will have the option to move back to the preceding page.

### **Challenge #5: Version**

Every OS is updated periodically, with new features designed to entice users to upgrade. But not every user will upgrade their phone to the latest

version. It's important to use analytics to determine which versions your users are most likely to have, and make sure you are testing on those versions. Also, every version update has the potential to create bugs in your application that weren't there before.

Example bug: Often when the version was updated on my phone, I could no longer use the speaker function when making phone calls. I could hear the voice on the other end, but the caller couldn't hear me.

### **Challenge #6: Manufacturer**

While all iOS devices are manufactured by Apple, Android devices are not so simple. Samsung is one of the major Android device manufacturers, but there are many others, including Huawei, Motorola, Asus, and LG. It's important to note that not every Android user will be using a Samsung device, and therefore to test on other Android devices as well.

Example bug: I once tested a tablet application whose keyboard function worked fine on some makes but not others. The keyboard simply wouldn't pop up on those devices, so I wasn't able to type in any form fields.

### **Challenge #7: Model**

Similar to versioning, new models of devices are introduced annually. While some users will upgrade every year or two to the latest device, others will not. Moreover, some devices will not be able to upgrade to the latest version of the OS, so they will be out of date in two ways. Again, it's important to find out what models your end users are using so that you can decide which models to test on and support.

Example: This is not a bug, but it was an important consideration. When Apple released a new model of the iPad that would allow a signature control for users to sign their name in a form, the software I was testing included this feature. But older versions of the iPad weren't able to support this, so the application needed to account for this and not ask users on older versions to sign a document.

### **Challenge #8: Tablet or Smartphone**

Many of your end users will be interacting with your application on a tablet rather than a smartphone. Native applications will often have different app versions depending on whether they are designed for a tablet or a phone. An application designed for a smartphone can often be downloaded to a tablet, but an application designed for a tablet cannot be installed on a smartphone. If a web app is being used, it's important to remember that tablets and smartphones sometimes have different features. Test your application on both tablets and phones.

Example bug: I have tested applications that worked fine on a smartphone and simply gave me a blank screen when I tried to test them on a tablet.

### **Challenge #9: Screen Size**

Mobile devices come in many different sizes. While iOS devices fit into a few sizing standards, Android devices have dozens of sizes. Although it's impossible to test every screen size, it's important to test small, medium, large, and extra-large sizes to make sure your application draws correctly in every resolution.

Example bug: I have tested applications on small phones whose page elements were overlapping each other, making it difficult to see text fields or click on buttons.

### **Challenge #10: Portrait or Landscape**

When testing on smartphones, it's easy to forget to test in landscape mode because we often hold our phones in a portrait position. But sometimes smartphone users will want to view an application in landscape mode, and this is even more true for tablet users. It's important to not only test your application in portrait and landscape modes, but also to switch back and forth between modes while using the application.

Example bug: I tested an application once that looked great on a tablet when it was in portrait mode, but all the fields disappeared when I moved to landscape mode.

### **Challenge #11: In-App Integration**

One of the great things about mobile applications is that they can integrate with other features of the device, such as the microphone and camera. They can also link to other applications, such as Facebook and Twitter. Whatever integrations the application supports, be sure to test them thoroughly.

Example bug: I tested an application that allowed users to take a picture of an appliance in their home and add it to their home's inventory. When I chose to take a picture, I was taken to the camera app correctly and was able to take the picture, but after I took the picture I wasn't returned to the application.

### **Challenge #12: Integration with Other Apps**

Even if your application isn't designed to work with any other apps or features, it's still possible there are bugs in this area. What happens if the user gets a phone call, a text, or a low-battery warning while they are using your app? It's important to find out.

Example bug: For a while, if the device timer on my phone went off while I was on a call, as soon as I got off the phone the timer would sound and wouldn't stop.

These descriptions and examples should show just how difficult it is to test mobile applications. In the next chapter, we'll take a look at writing mobile test plans and assembling a portfolio of physical devices on which to test them.

# Chapter 45: Manual Mobile Testing

I firmly believe that no matter how great virtual devices and automated tests are, you should always do some mobile testing with a physical device in your hand. But none of us has the resources to acquire every possible mobile device with every possible carrier! So this chapter discusses how to assemble a mobile device portfolio that meets your minimum testing criteria and how to get your mobile testing done on other physical devices. It also covers the manual tests that should be part of every mobile test plan.

Every company is different and will have a different budget available for acquiring mobile devices. Here is an example of how I would decide which phones to buy if I were allowed to purchase no more than 8. I would want to make sure I had at least one of the top three carriers for my service area (United States) in my portfolio. I would also want to have a Wi-Fi-only device, and I would want to have at least one iOS device and at least one Android device. For OS versions, I'd want to have both the latest OS version and the next-to-latest OS version for each operating system. For Android devices, I'd want to have the three most popular manufacturers for my service area. Finally, I would want to make sure I had at least one tablet for each operating system.

With those stipulations in mind, I would create a list of devices similar to this:

| Model             | Carrier   | Phone/Tablet | OS              | Version                    | Screen Size (inches) |
|-------------------|-----------|--------------|-----------------|----------------------------|----------------------|
| iPhone 12         | AT&T      | Phone        | iOS             | iOS 14                     | 6.1                  |
| iPad              | Verizon   | Tablet       | iOS             | iOS 14                     | 10.2                 |
| iPhone 12 mini    | T-Mobile  | Phone        | iOS             | iOS 13                     | 5.4                  |
| Galaxy S21 5G     | AT&T      | Phone        | Android         | Android 11                 | 6.2                  |
| Galaxy Tab S5e    | AT&T      | Tablet       | Android         | Android 9                  | 10.5                 |
| Motorola One 5G   | Verizon   | Phone        | Android         | Android 10                 | 6.7                  |
| LG VELVET 5G      | T-Mobile  | Phone        | Android         | Android 10                 | 6.8                  |
| Amazon Fire HD 10 | Wifi only | Tablet       | Fire OS/Android | Fire OS 7.1.1./Android 9.0 | 10.1                 |

This portfolio includes three iOS devices and five Android devices. All three carriers I wanted are represented, as well as one Wi-Fi-only device. The portfolio also includes three tablets and five smartphones; the latest and next-

to-latest iOS and Android versions; and a variety of screen sizes.

The benefit of having a physical device portfolio is that you can add to it every year as your budget allows. Each year you can purchase a new set of devices with the latest OS version, and you can keep your old devices on the older OS versions, thus expanding the range of OS versions you can test with.

Once you have a device portfolio, you'll need to make sure you are building good mobile tests into your test plans. In addition to the general application functionality, you should add the following tests:

- Test the application in the mobile browser, in addition to testing the native app.
- Test in portrait and landscape modes, switching back and forth between the two.
- Change from using the network, to using Wi-Fi, to using no service, and back again.
- Test any in-app links and social media features.
- Set the phone or device timer to go off during your testing.
- Set text messages to arrive and low-battery warnings to activate during your testing.

What about testing on the hundreds of devices that you don't have? This is where device farms come into play. A device farm consists of many physical devices housed in one location which you can access through the Web. From your computer, you can access device controls such as the home and back buttons, swipe left and right on the screen, and click on the controls in your application. You may even be able to do things like rotate the device and receive a phone call.

With a device farm, you can expand the range of devices on which you are testing. Good ideas for expanding your test plan would be adding devices with older OS versions and adding devices from manufacturers you don't have in your portfolio.

What if your mobile application is designed for people to use all over the world? You probably won't be able to get devices or carrier plans from other countries. How can you make sure users everywhere are having a good

experience with your app? You can use crowdsourced testing!

Crowdsourced testing companies specialize in using testers from many countries who are using devices with their local carriers. They can test your application in their own time zone with a local carrier and a local device.

With a mobile device portfolio, a mobile test plan, a device farm, and a crowdsourced testing service in place, you will be able to execute a comprehensive suite of tests on your application and ensure a great user experience worldwide.

# Chapter 46: Seven Tips for Mobile Automated Testing

Walk into any mobile carrier store and you will see a wide range of mobile devices for sale. Of course you want to make sure your application works well on all of those devices, in addition to the older devices that some users have. But running even the simplest of manual tests on a phone or tablet takes time. Multiply that time by the number of devices you want to support and you've got a huge testing burden!

This is where automated mobile testing comes in. Here are seven tips to help you be successful with mobile automated testing.

## **Tip #1: Don't Test Things on Mobile That Could Be More Easily Tested Elsewhere**

Mobile automation is not the place to test your backend services. It's also not the place to test the application's general logic, unless your application is mobile only. Mobile testing should be used for verifying that elements appear correctly on the device and function correctly when used.

For example, let's say you have a sign-up form in your application. In your mobile testing, you'll want to make sure the form renders correctly, all fields can be filled in, error messages display appropriately, and the Save button submits the form when the user clicks it. But you don't want to test that the error message has the correct text or that the fields have all been saved correctly. You can save those tests for standard web browser or API automation.

## **Tip #2: Decide Whether You Want to Run Your Tests on Real Devices or Emulators**

The advantage of running your tests on real devices is that the devices will behave like the devices your users own, with the possibility of having a low battery, connectivity issues, or other applications running. But because of

this, it's possible that your tests will fail because a phone in the device farm froze or was being used by another tester. Annoyances like these can be avoided by using emulators, but emulators can't completely mimic the real user experience. It's up to you to decide which option is more appropriate for your application. You can also use both!

### **Tip #3: Test Only One Thing at a Time**

Mobile tests can be flaky due to the issues we just discussed as well as other issues such as the variations found in different phones and tablets. You may find yourself spending a fair amount of time diagnosing your failed tests. Therefore, it's a good strategy to keep your tests small. For example, if you were testing a login screen, you could have one test for a successful login and a second test for an unsuccessful login, instead of putting both scenarios into the same test.

### **Tip #4: Be Prepared to Rerun Tests**

As mentioned in tip #3, you will probably encounter some flakiness in your mobile tests. A test can fail simply because the service hosting the emulator loses connectivity for a moment. So you may want to set up a system in which your tests run once and then rerun the failed tests automatically. You can then set up an alert that will notify you only if a test has failed twice.

### **Tip #5: Don't Feel Like You Have to Test Every Device in Existence**

As testers, we love to be thorough. We love to come up with every possible permutation in testing and run through them all. But in the mobile space, this can quickly drive you crazy! The more devices on which you are running your automated tests, the more failures you will have. The more failures you have, the more time you will have to spend diagnosing those issues. This is time taken away from new-feature testing or exploratory testing. Research which devices your users own and create a list of devices to test with that covers most of those devices.

### **Tip #6: Take Screenshots**

Nothing is more frustrating than seeing that a test failed and not being able to figure out why. Screenshots can help you determine whether you were on the correct screen during a test step and whether all the elements are visible. Some mobile testing companies take a screenshot of every test step as the test progresses. Others automatically take a screenshot of the last state of the application when the test fails. You can also code your test to take screenshots of specific test steps.

### **Tip #7: Use Visual Validation**

Visual validation is essential in mobile testing. Many of the bugs you will encounter will concern elements not rendering correctly on the screen. You can test for the presence of an element, but unless you have a way to compare a screenshot with one you have on file, you won't really be verifying that your elements are visible to the user. Many tools are available that build visual verification right into your tests and save a collection of screenshots from every device you test with to use for image comparison.

By heeding these tips, you will ensure that you are comprehensively testing your application on mobile without spending a lot of time debugging.

# Part VI: Security Testing

# Chapter 47: Introduction to Security Testing

Until a few years ago, security testing was thought of as something separate from traditional software testing; something an Application Security team would take care of. But massive data breaches have demonstrated that security is everyone's responsibility: from CEOs to product owners, DBAs, developers, and yes, software testers. Testers already verify that software is working as it should so that users will have a good user experience; it is also crucial for testers to verify that software is secure so that users' data will be protected.

The great news is that much of what you already do as a software tester helps with security testing. This chapter covers how testers can use the skills they already have to start testing with security in mind, as well as the new skills testers can learn to help secure their applications.

## **Things you are probably already testing:**

- **Field validation:** You likely are already ensuring that fields only accept the data types they are expecting and that the number and type of characters are enforced. This helps ensure that SQL injection and cross-site scripting can't be entered through a data field.
- **Authentication:** Everyone knows it's important to test an application's login page. You are probably already testing to make sure that when a login fails the UI doesn't provide any hints as to whether the username or password failed; and to make sure the password isn't saved after logout or displayed in clear text. This helps make it more difficult for a malicious user to figure out how to log in to someone else's account.
- **Authorization:** You likely are paying attention to which user roles have access to which pages. By verifying that only authorized users can view specific pages, you are helping to ensure that data does not fall into the wrong hands.

## **Things you can learn for more comprehensive security testing:**

- **Intercepting and manipulating requests:** It is easy to intercept web requests with free tools that are available to everyone online. Since attackers do this regularly, you must ensure that they can't get access to information they shouldn't have.
- **Cross-site scripting (XSS):** This involves entering scripted code that will be executed when someone navigates to a page or retrieves data. Any text field or URL represents a potential attack point for a malicious user to insert a script.
- **SQL injection:** This is exploiting potential security holes in communication with the database to retrieve more information than the application intended. As with cross-site scripting, any text field or URL can potentially be used to extract data.
- **Session hijacking:** It's important to learn whether usernames, passwords, tokens, or other sensitive information is displayed in clear text or poorly encrypted. Malicious users can use this information to log in as someone else.

Security testing involves a shift in mindset from traditional testing. When we test software, we are usually thinking like an end user. For security testing, we need to think like a malicious user. End users take the happy path because they are using the software for its intended purpose, whereas hackers are trying to find possible security holes and exploit them.

# Chapter 48: Using Dev Tools to Find Security Flaws

A common misconception is that all security testing is complicated. While some testing certainly requires learning new skills and understanding things like networks, IP addresses, and domain names, other testing is extremely simple. In this chapter, you'll learn about three security flaws you can find in an application by simply using your browser's developer tools. These flaws could be exploited by an average user of your application, not just a well-trained malicious hacker.

To access the developer tools in Chrome, simply click on the three-dot menu in the upper-right corner of the browser and choose More Tools, then Developer Tools. The tools will open on the right side or the bottom of the screen.

To access the developer tools in Firefox, click on the three-line menu in the upper-right corner of the browser and choose Web Developer, then Web Developer Tools.

## **Flaw #1: Editing Disabled Buttons**

When you are on a web page that has a disabled button which is only enabled when certain criteria are met, such as filling out all the fields in a form, it may be possible to enable it without meeting the criteria. I included instructions on how to do this in Chapter 10.

Users of your application might use this flaw to avoid submitting a form that includes required fields they don't want to fill in. Or they might enable an Edit button and submit edits to data they shouldn't be able to edit.

## **Flaw #2: Viewing Hidden Data**

I was once shown a security flaw in an application that was listing contact details for various members of the site. Depending on the user's access rules,

there were certain fields for the members, such as their personal address, that were not displayed on the page. But when the developer tools were opened, all the hidden field values were displayed in the Elements section! Any user of this application could open the developer tools and search through them to find personal data for any of the site's members.

### Flaw #3: Finding Hidden Pages

It's possible to find links that are not displayed on a web page by looking in the Elements section of the developer tools. If you right-click on an element in a web page and choose Inspect, you'll be taken to the HTML for the page. See whether you can find an element that is marked "ng-hide" or "hidden":

```
▼<li class="dropdown ng-hide" ng-show="isLoggedIn()">
  ►<a href="#/recycle">...</a>
</li>
```

If you append the href value to the URL you are on, you may be able to get to a page that is supposed to be hidden from you.

This is why it is important to do authorization checks when a user navigates to a page. It's not enough to simply hide a link, because it's so easy to find hidden links by looking in the developer tools. Any user could find hidden links in your application and navigate to them, which could give them access to an admin page or a page with other users' data.

As you can see, testing for these types of security flaws is quick and easy. I recommend checking for these three flaws whenever you test a web page.

# Chapter 49: Testing for IDOR Vulnerabilities

IDOR stands for Insecure Direct Object Reference, and it refers to a situation in which a user can successfully request access to a web page, data object, or file they should not have access to. In this chapter, I'll describe four ways this vulnerability might appear. Then we'll exploit this vulnerability in a test application using Chrome's DevTools and Postman.

One easy way to look for IDOR is in a URL parameter. Let's say you are an online banking customer of a really insecure bank. When you go to your account page, you log in and are taken to this URL: `http://mybank/customer/27`. Looking at this URL, you can tell that you are customer #27. What would happen if you changed the URL to `http://mybank/customer/28`? If you are able to see customer #28's data, you have definitely found an instance of IDOR!

Another easy place to look is in a query parameter. Imagine that your name is John Smith and you work for a company that conducts annual employee reviews. You can access your review by going to `http://mycompany/reviews?employee=jsmith`. You are very curious about whether your co-worker, Amy Jones, received a better review than you did. You change the URL to `http://mycompany/reviews?employee=ajones`, and voilà! You now have access to Amy's review.

A third way to look for IDOR is by trying to get to a page that your user should not have access to. If your website has an admin page with a URL of `http://mywebsite/admin`, which is normally accessed by a menu item that is only visible when the user has admin privileges, see what happens if you log in as a non-admin user and then manually change the URL to point to the admin page. If you can get to the admin page, you have found another instance of IDOR.

Finally, it's also possible to exploit an IDOR vulnerability to access files that a user shouldn't have access to. Let's say your site has a file called

userlist.txt with the names and addresses of all your users. If you can log in as a non-admin user and navigate to <http://mywebsite/files?file=userlist.txt> by typing it into the URL field, your files are not secure.

Let's take a look at IDOR in action using Postman, Chrome DevTools, and an awesome website called the OWASP Juice Shop. The OWASP Juice Shop is an application created by Björn Kimminich to demonstrate the most prevalent security vulnerabilities. You can find it at <http://juice-shop.herokuapp.com>.

Once you have navigated to the site on Chrome, create a login for yourself. You can use any email address and password to register (don't use any real ones!). Log in as your new user, and click on any of the juices on the Search page to add it to your shopping basket.

Before you take a look at your basket, open DevTools by clicking on the three dots in the upper-right corner of the browser, selecting More Tools, and then Developer Tools. A new window will open on either the right or the bottom of your browser. In the tools' navigation bar, you should see a Network option; click on it. This network tool will display all the network requests you are making in your browser.

Click on the Shopping Cart icon. You will be taken to your shopping cart and you should see the juice that you added to the basket. Take a look in the Network section of DevTools. The request you are looking for is one that is named simply with a number, such as "6" or "7". That number represents your account ID. Click on this request and you should see that the request URL is <http://juice-shop.herokuapp.com/rest/basket/<whateverYourAccountIdIs>> and the request type is a GET. (If you don't see this URL, make sure that you have clicked on the "Headers" tab.) Scrolling down a bit, you'll see that in the Request Headers, the Authorization is set to Bearer. Then you'll see a long string of letters and numbers. This is the auth token. Copy everything in the token, including the word "Bearer".

Next, we'll re-create the request in Postman. Click on the plus (+) tab to create a new request. The request should already be set to GET by default. Enter <https://juice-shop.herokuapp.com/rest/basket/<yourAccountId>> into the

URL, making sure to replace <yourAccountId> with your actual ID. Now go to the Headers section; underneath the Key section type “Authorization”, and underneath the Value section paste the string you copied. Click to Send the request, and if things are set up correctly, you will be able to see the contents of your shopping basket in the response.

Now for the fun part! Change the account ID in the URL to a different number, such as something between 1 and 5, and click Send. You will see the contents of someone else’s basket. Congratulations! You just exploited an IDOR vulnerability!

# Chapter 50: Introduction to Cross-Site Scripting

Cross-site scripting (XSS) is an attack in which a malicious user finds a way to execute a script on an unsuspecting user's website. In this chapter you'll learn about two types of XSS attacks, walk through a hands-on demo of each, and learn why they are harmful to the end user.

## Reflected XSS

Reflected XSS is an attack that is executed through the web server but is not stored in the code or the database. Because the attack is not stored, the site owner may have no idea the attack is happening.

To demonstrate this attack, we'll go to a great training site from Google called the XSS Game: <https://xss-game.appspot.com>. This site has a series of challenges in which you try to execute XSS attacks. The challenges become increasingly difficult as they progress. Let's try the first challenge.



On this page, you'll see a simple search field and Search button. To execute the attack, all you need to do is type `<script>alert("XSS here!")`

`</script>` into the text field and click Search. You will see your message, “XSS here!” pop up in a new window.

You just sent a script to execute a pop-up alert to the server. The client-side code does not have appropriate safeguards in place to prevent a script from executing, so the site executed the script.

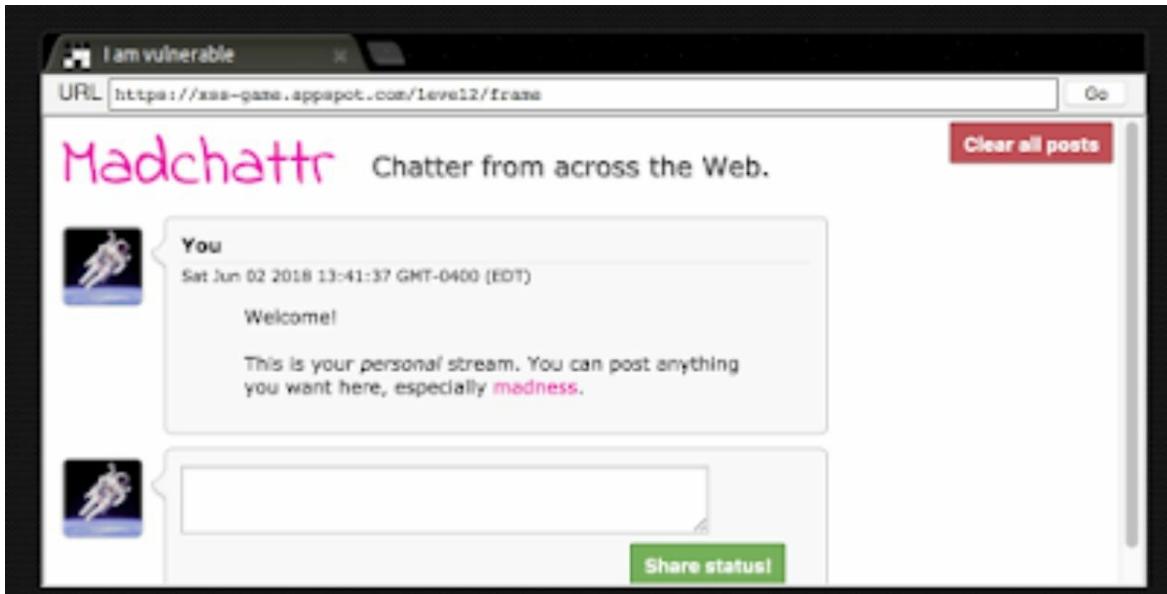
You might be thinking: “This is a fun trick, but how could a malicious user use this to hack me? I’m typing into my own search window.” One way this is used is through a phishing link. Let’s say you are the owner of a website. A malicious user could create a link that goes to your site but appends a script to the end of the URL, such as

`?query=%3Cscript%3Ealert%28%22XSS%22%29%3C%2Fscript%3E.` (This is simply the attack we used earlier, with HTML encoding.) The malicious user could send this link in an email to an unsuspecting visitor to your site, making the email look like it came from you. When the person clicks on the link, the script will navigate to your site and then execute the pop-up script. The malicious user will craft the script so that instead of containing the message “XSS here!”, it contains a script that encourages the visitor to interact with it to obtain the user’s account number or other sensitive information.

## Stored XSS

Stored XSS is an attack in which the malicious script is stored in the database or code of a website, so it executes whenever a user navigates to the page or link. This could happen if the site’s creator did not put adequate input sanitization in the backend database.

We’ll look at how to craft this attack by working through the second challenge in the XSS Game. (To see this challenge, you’ll need to have solved the first challenge, so follow the instructions I provided earlier.)



In the second challenge, you are presented with a chat app. To solve the challenge, you need to add some text to the application that will execute a script. You can do this by typing `<img src='foobar' onerror='alert("xss")'>`.

As soon as you submit this entry, you should see a pop-up window with the “XSS alert!” message. Furthermore, if you navigate away from this page and return to it, you should see the pop-up window again. The attack has been stored in your comment on the chat page, where it will cause a pop-up for any users who navigate to it.

Let’s parse through the script we entered to see what it’s doing:

- `<img src='foobar' onerror='alert("xss")'>`  
The items in red indicate that we are passing in an image element.
- `<img src='foobar' onerror='alert("xss")'>`  
The section in blue is telling the server what the source of the image should be. And here’s the trick: there is no URL of 'foobar', so the image cannot load.
- `<img src='foobar' onerror='alert("xss")'>`  
The section in green is telling the server that if there is an error, a pop-up window should be generated with the “xss” text. Because we have set things up so that there will always be an error, this pop-up will always execute.

One way that stored XSS might be used is to spoof a login window. When a user navigates to a hacked site, they will be presented with a login window that has been crafted to look authentic. When they enter their login credentials, their credentials will be sent to the malicious user, who can now use them to log in to the site, impersonating the victim.

# Chapter 51: Three Ways to Test for Cross-Site Scripting

In the preceding chapter, I explained what cross-site scripting is and demonstrated a couple of examples. But knowing what it is isn't enough: we need to be able to verify that our application is not vulnerable to XSS attacks. In this chapter you'll learn three different strategies to test for XSS.

## Strategy #1: Manual Opaque Testing

This is the strategy to use when you don't have access to an application's code and when you want to manually try XSS. To implement this strategy, you'll need to think about the places where you could inject a script into an application:

- An input field
- A URL
- The body of an HTTP request
- A file upload area

You'll also need to think about what attacks you will try. You may want to use an existing list, such as this one: [https://www.owasp.org/index.php/XSS\\_Filter\\_Evasion\\_Cheat\\_Sheet](https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet).

This cheat sheet includes lots of different ways to get scripts past any validation filters, including:

- On error and on mouseover alerts
- URL encoding
- Using upper- and lowercase letters (to evade a filter that's just looking for "javascript" in lowercase letters)
- Putting a tab or space into a script so that it won't be detected
- Using a character to end an existing script and then appending your own

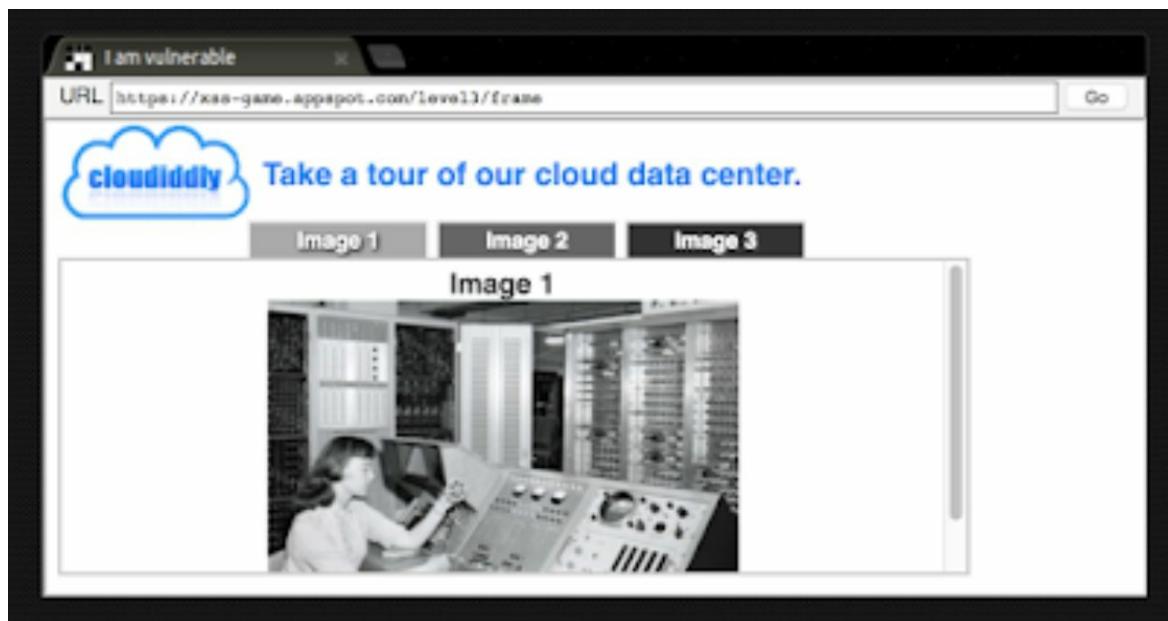
If you are testing manually, a systematic approach is best. Locate all the

places where you could inject a script, choose a list of attacks you'd like to try, and try each attack in each place. While you are testing, you may also gain some insight on how you could change the attack by what kind of response you get from the attack. For example, if your script tag is stripped out by validation, you could try to encode it.

## Strategy #2: Look at the Code

This is the strategy to use if you want to test manually and you have access to your application's code. By looking at the code, you can determine the best way to craft an attack script. This also works well for testing file uploads; for example, if your application's code lists file types that are not allowed, you may find some types that have not explicitly been disallowed and you can try uploading a script using one of those types.

Let's take a look at how you can use an application's code to craft an attack. We'll use the third challenge in the XSS Game; to access it you need to have solved the first and second challenges, so follow the directions in the preceding chapter to see how to do that.



As you look at the website in the third challenge, you can see that there are three different tabs, each of which displays a different image when clicked. Take a look at what happens in the URL each time you click on one

of the tabs. When you click on the Image 2 tab, “#2” is appended to the URL. When you click on the Image 3 tab, “#3” is appended to the URL.

What happens when instead of clicking on a tab, you type “[#2](#)” into the URL? Unsurprisingly, you are taken to the Image 2 tab. What happens when you type “[#5](#)” into the URL? There is no Image 5 tab, but you can see that the page displays the words “Image 5”. What happens when you type “[#FOO](#)”? The page displays “Image NaN” (short for “Not a Number”). You have probably figured out by now that the end of the URL is the place you are going to inject your malicious script.

Now let’s take a look at the code: click on the “toggle” link next to the words Target Code. This will display the code used for the web page. Look at line 17; it shows how the URL for the image tag is created:

```
"<img src='/static/level3/cloud" + num + ".jpg' />";
```

The num part of this image tag is a variable. The value of the variable is taken from what we are sending in the URL. If you send in a URL with “#3”, the image tag will be cloud3.jpg. If you send in a URL with “#FOO”, the image tag will be cloudFOO.jpg.

Our task now is to see how we can inject a script using this num variable. Recall that in the preceding chapter we did some cross-site scripting that made it look like we were uploading an image, and we included an alert that would display when there was an error uploading the image. And we also set things up so that there would always be an error, because we weren’t really uploading an image at all. We are going to do the same thing here.

Let’s craft our URL. We will begin with <https://xss-game.appspot.com/level3/frame> because this is how the URL always starts.

Next, we’ll add <https://xss-game.appspot.com/level3/frame#3> because we want to make it look like we are following the pattern of choosing an image number.

Now we’ll add <https://xss-game.appspot.com/level3/frame#3> because we want to trick the code into thinking the image URL is complete. This means the code will try to load an image called “cloud3” instead of “cloud3.jpg,”

which will generate an error.

Now we can add our on-error script:

[https://xss-game.appspot.com/level3/frame#3' onerror='alert\('Hacked!'\)'](https://xss-game.appspot.com/level3/frame#3' onerror='alert('Hacked!')')

When the alert is triggered, a pop-up window will appear with the “Hacked!” message.

Let's try it! Paste the entire URL, [https://xss-game.appspot.com/level3/frame#3' onerror='alert\('Hacked!'\)'](https://xss-game.appspot.com/level3/frame#3' onerror='alert('Hacked!')') into the URL window and click the Go button.

You should see the pop-up window appear, and you have solved the challenge!

### **Strategy #3: Use a Security Testing Tool**

As you can see from the preceding example, crafting an XSS attack takes a little time. You may have many places in your application that you need to test for XSS, and not much time to test them. This is where automated tools come in. With an automated tool such as Burp Suite, you can send hundreds of XSS attacks in less than a minute.

# Chapter 52: Introduction to SQL Injection

SQL injection is another type of security attack that can do serious damage to your application. It's important to find SQL injection vulnerabilities before a malicious user does.

In SQL injection, a malicious user sends a SQL query through a form field which interacts with the database in an unexpected way. Here are four things a malicious user might do with SQL injection:

- Drop a table
- Change another user's records
- Return records the user shouldn't have access to
- Log in without appropriate credentials

To understand how a SQL injection attack is crafted, let's look at an example. Say our application has a form with a username field. When the username field is populated with a name such as "testerguy" and is submitted to the server, the following SQL query is run:

```
SELECT * from users where username = 'testerguy'
```

If this username exists in the database, results for the users table are returned to the application.

A malicious user will try to trick the database by doing the following:

- Making it think the entry has terminated, by passing in **testerguy'**
- Adding an additional clause, such as **OR 1=1**
- Adding a terminating statement such as **;** to make sure no other SQL statement will be run

In the preceding example, the user would add the following to the username field:

**testerguy' OR 1=1;**

The database would then execute the following:

```
SELECT * from users where username = 'testerguy' OR 1=1;
```

Take a moment to think about the `1=1` clause. `1=1` is always true, so the database interprets this as selecting everything in the table! Therefore, this select statement is asking for all the values for all the users in the table.

Let's see some SQL injection in action, using the OWASP Juice Shop at <http://juice-shop.herokuapp.com>. We are going to use SQL injection to log in without valid credentials. Make sure you have logged out from your previous session, and then click the Login button.

We'll assume that when the login request happens, a request like this goes to the database:

```
SELECT * from users where username = 'testerguy' AND password = 'mysecretpass'
```

If the request returns results, it's assumed that the user is valid, and the user is logged in.

What we want to do is try to terminate the statement so that all usernames will be returned and the password isn't viewed at all.

So we will send in the following:

1. Any username at all, such as “`foo`”
2. A single quote mark, `'`, to make it look like our entry has terminated
3. The clause `OR 1=1` to make the database return every username in the table
4. A terminating string of `--` (two dashes) to make the database ignore everything after our request

Taken together, the string we will add to the username field is:

`foo' OR 1=1--`

You may notice that the Submit button is not enabled yet. This is because we haven't added a password. The UI expects both a username and a password in order to submit the login. You can add any text you want into the password field because we are ensuring that it will be ignored. Let's add "bar".

Now when you submit the login request, this is what will be executed on the database:

```
SELECT * from users where username = 'foo' OR 1=1-- AND password = 'bar'
```

The first part of the request is returning all users because `1=1` is always true. And the second part of the request, indicated here in gray, will be ignored because in SQL everything after the dashes is commented out. So when the code sees that all users have been returned, it logs us in!

If you click on the person icon at the upper-right of the screen, you will see that you have actually been logged in as the admin! The admin's email address was the first address in the database, so this is the credential that was used. Because you are logged in as the admin, you now have elevated privileges on the website that you would not have as a regular user.

Obviously, you would want to avoid this sort of scenario in your application! So it's a good idea to try SQL injection on your application's test database to make sure queries are being properly sanitized.

# Chapter 53: Introduction to Session Hijacking

We all know that session hijacking is bad and that we should protect ourselves and our applications against it. But it's difficult to get easy-to-understand information about what it is and how to test for it. This chapter covers the different types of session hijacking and then walks you through the steps to test for it using the OWASP Juice Shop and Burp Suite.

Session hijacking refers to when a malicious user gets access to authentication information and uses it to impersonate another user or gain access to information they should not have. There are several types of session hijacking:

- **Predictable session token:** This happens when the access tokens an application is generating follow some kind of pattern. For example, if the log-in token granted for one user was “APP123” and the login token granted for a second user was “APP124”, a malicious user could assume the next token granted would be “APP125”. This is a pretty obvious vulnerability, and there are many tools in use today that create nonsequential tokens, so it’s not a very common attack.
- **Session sniffing:** This takes place when a malicious user finds a way to examine the web traffic that is being sent between a user and a web server, and copies the token for their own use. This is classified as a passive attack because the malicious user is not interfering with the operation of the application or with the request.
- **Client-side attack:** In this scenario, the malicious user is using XSS to cause an application to display a user’s token. Then they copy the token and use it.
- **Man-in-the-middle attack:** This attack is similar to session sniffing in that the malicious user gains access to web traffic. But this is an active attack because the malicious user uses a tool such as Burp

Suite or Fiddler to intercept the request and then manipulate it for their purposes.

- **Man-in-the-browser attack:** This takes place when a malicious user has managed to get code into another user's computer. The implanted code will then send all request information to the malicious user.

Now we'll discuss how to test for session hijacking by using a man-in-the-middle attack. But first, it's important to note that we will be intercepting requests by using the same computer we are using to make the requests. In a real man-in-the-middle attack, the malicious user would be using some sort of packet-sniffing tool to gain access to requests that someone was making on a different computer.

First, we'll need to download Burp Suite. The link to install the free Community edition is available at <https://portswigger.net/burp/communitydownload>. Don't open Burp Suite yet; we'll do that after we've done some work in the Juice Shop.

Navigate to the Juice Shop at <http://juice-shop.herokuapp.com>, and create an account. After you've created the account, you'll be prompted to log in, but don't log in just yet.

Now open Burp Suite. Click Next, and then Start Burp. Click on the Proxy tab and then click "Open browser". A Chromium browser window will open. Enter <http://juice-shop.herokuapp.com> into the URL field. The page will start trying to load, but it won't load, because Burp Suite is intercepting the request. Click the Forward button in Burp Suite until the page is completely loaded.

In the Juice Shop's Chromium window, click the Account button and then the Login button. Enter your login credentials and click "Log in". Go to Burp Suite and click the Forward button. Notice that your credentials are displayed in plain text in the intercept window! This is one example of how the Juice Shop is vulnerable. Keep clicking the Forward button until the login has completed.

Next, return to the Juice Shop and click on the cart symbol for the first juice listed to add it to your shopping cart. Return to Burp Suite and click the Forward button to forward the request. Continue to click the Forward button until no more requests are intercepted.

Go to the HTTP History tab in Burp Suite, and scroll down through the list of requests until you see a POST request with the api/BasketItems endpoint. Right-click on this request and choose Send to Repeater. This will send the request to the Repeater module where we can manipulate the request and send it again. Return to the Intercept tab and turn the Intercept off by clicking on the “Intercept is on” button.

Click on the Repeater tab, which is in the top row of tabs. The request we intercepted when we added a juice to the shopping cart is there. Let’s try sending the request again, by clicking on the Send button. In the right panel of the page, we get a validation error, with the message that the Basket Id and Product Id must be unique. So, let’s return to the request in the left panel. We can see that the body of the request is {"ProductId":1,"BasketId":"<whatever your basket id is" , "quantity":1}. Let’s change ProductId from 1 to 2, and send the request again by clicking the Send button. We can see in the response section that the request was successful.

Let’s return to the Juice Shop and see whether we were really able to add an item to the cart by sending the request in Burp Suite. Click on the Your Basket link. You should see two juices in your cart! This means that if someone were to intercept your request to add an item to your cart, they could manipulate the request and use it to add any other item they wanted to your cart.

When you set up Burp Suite to intercept requests in your own application, you will be able to test for session hijacking vulnerabilities like this one.

# Chapter 54: An Introduction to Mobile Security Testing

Mobile security testing can be problematic for a software tester because it combines the challenges of mobile with the challenges of security testing. Here are some of the difficulties:

- Mobile devices are designed to be more secure than traditional web applications because they are personal to the user. Therefore, it's harder to look "under the hood" to see how an application works.
- Mobile security testing often requires tools that the average tester might not have handy, such as Xcode Tools or Android Studio.
- Security testing on a physical device usually means using a rooted or jailbroken phone. (A rooted or jailbroken phone is one that is altered to have admin access or user restrictions removed. An Android phone can be rooted; an iPhone can be jailbroken. You will not want to do this with your personal device.)

It's difficult to find instructions for mobile security testing when you are a beginner; most documentation assumes that you are already comfortable with advanced security testing concepts or with developing mobile applications.

This chapter is intended to serve as a gentle introduction for testers who are not already security testing experts or mobile app developers. Let's first take a look at the differences between web application security testing and mobile app security testing:

- Native apps are usually built with the mobile OS's development kit, which has built-in features for things like input validation so that SQL injection and cross-site scripting vulnerabilities are less likely.
- Native apps often make use of the data storage capabilities on the

device, whereas web applications will store everything on the application's server.

- Native apps will be more likely than web applications to use biometric data, such as a fingerprint, for authentication.

However, there are still a number of vulnerabilities that you can look for in a mobile app which are similar to the types of security tests you would run on a web application. Here are some examples:

- For apps that require a username and password to log in, you can check to make sure a login failure doesn't give away information. For example, you don't want your app to return the message "invalid password", because that lets an intruder know they have a correct username.
- You can use a tool such as Postman to test the API calls the mobile app will be using and verify that your request headers are expected to use "https" rather than "http".
- You can test for validation errors. For example, if a text field in the UI accepts a string that is longer than what the database will accept, this could be exploited by a malicious user for a buffer overflow attack.

If you are ready for a bigger testing challenge, here are a couple of mobile security testing activities you can try:

- You can access your app's local data storage and verify that it is encrypted. On an Android device, you can do this with a rooted phone or an Android emulator and the Android ADB (Android Debug Bridge) command-line tool. On an iPhone, you can do this with Xcode Tools and a jailbroken phone or an iPhone simulator.
- You can use a security testing tool such as Burp Suite to intercept and examine requests made by the mobile app. On an Android phone, you'll need to do this with an emulator. On an iPhone, you can do this with a physical device or a simulator. In both instances,

you'll need to install a CA certificate on the device that allows requests to be intercepted. This CA certificate can be generated from Burp Suite itself.

These two testing tasks can prepare you to be a mobile security testing champion! If you are ready to learn even more, I recommend that you check out the online book “A Guide to the OWASP Mobile Security Project” (<https://info.nowsecure.com/Managers-Guide-OWASP-Mobile-Security-Project.html>). This is the definitive reference guide to making sure your application is free of common security vulnerabilities. Happy hacking!

## Part VII: Performance Testing

# Chapter 55: Introduction to Performance Testing

Performance testing measures how an application behaves when it is used.

This includes reliability:

- Does the page load when a user navigates to it?
- Does the user get a response when they make a request?

And speed:

- How fast does the page load?
- How fast does the user get a response to their request?

Depending on the size of the company you work for, you may have performance engineers or DevOps professionals who are already monitoring metrics like these. But if you work for a small company or you simply like to be thorough in your testing, it's worth learning how to capture some of this data to find out how well your application is behaving in the real world. I have stopped using an application simply because the response time was too slow. You don't want your end users to do that with your application!

Here are several ways that you can monitor the health of your application.

## Latency

This is the time it takes for a request to reach a server and return a response. The simplest way to test this is with a ping test. You can run a ping test from the command line on your computer by simply entering the word “ping” (minus the quote marks) followed by a website’s URL or an IP address. For example, you could run this command:

`ping www.google.com`

And you'd get a response like this:

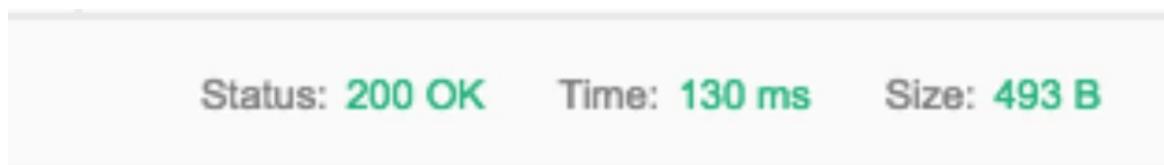
```
$:~$ ping www.google.com
PING www.google.com (172.217.10.132): 56 data bytes
64 bytes from 172.217.10.132: icmp_seq=0 ttl=57 time=19.291 ms
64 bytes from 172.217.10.132: icmp_seq=1 ttl=57 time=20.708 ms
64 bytes from 172.217.10.132: icmp_seq=2 ttl=57 time=21.975 ms
64 bytes from 172.217.10.132: icmp_seq=3 ttl=57 time=23.557 ms
64 bytes from 172.217.10.132: icmp_seq=4 ttl=57 time=19.579 ms
64 bytes from 172.217.10.132: icmp_seq=5 ttl=57 time=21.407 ms
64 bytes from 172.217.10.132: icmp_seq=6 ttl=57 time=22.263 ms
^C
--- www.google.com ping statistics ---
7 packets transmitted, 7 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 19.291/21.254/23.557/1.405 ms
```

To stop the ping test, simply press **CTRL+C**.

Let's take a look at the response times. Each ping result shows how long it took in milliseconds to reach that server and return a response. At the bottom of the test results, we can see the minimum response time, average response time, maximum response time, and standard deviation in response time. In this particular example, the slowest response time was 23.557 milliseconds.

## API Response Time

This is a really helpful measurement because so many web and mobile applications are using APIs to request and post data. Postman has response time measurements built into the application. When you run a request, you will see a Time entry next to the Status of the response:



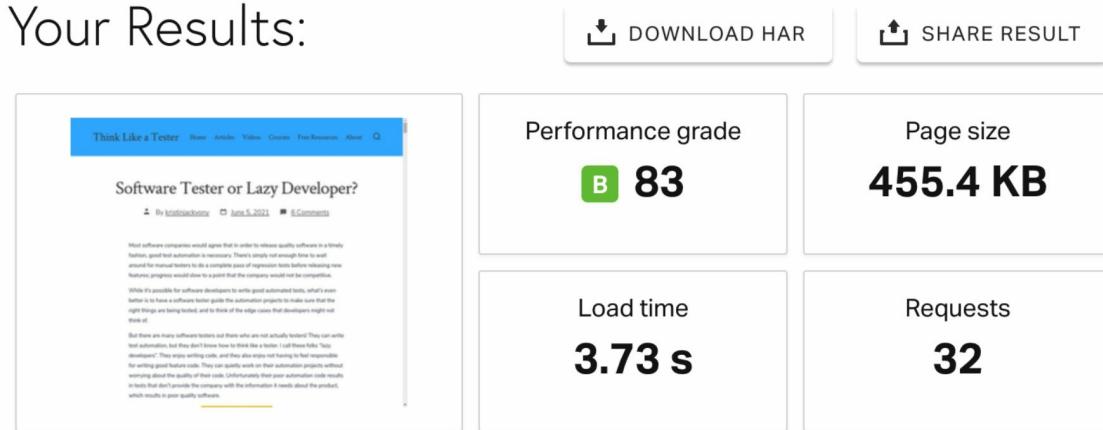
In this example, the request took 130 milliseconds to return a response.

You can include an assertion in your Postman tests which will verify that your response was returned in less than a selected time, such as 200 milliseconds. The assertion would look like this:

```
pm.test("Response time is less than 200ms", function () {  
  pm.expect(pm.response.responseTime).to.be.below(200);  
});
```

## Web Response Time

Even if your API is responding beautifully, you'll also want to make sure your web page is loading well. Nothing is more frustrating to a user than sitting around waiting for a page to load! There are a number of free tools that you can use to measure how long it takes your application's pages to render. With these tools you can enter your website's URL and the tool will crawl through your application, measuring load times. Here are the results I got with Pingdom when I used my website's URL and requested that it be tested from Melbourne, Australia:



Pingdom also provided suggestions for improving my site's performance, such as adding browser caching and minimizing redirects. Paid customers of applications like Pingdom can set up monitoring and alerting so that you can be notified if your page loading times slow down.

## Mobile Application Monitoring

If you have a native mobile application, you'll want to make sure it's

responding correctly and quickly. Tools like Firebase Crashlytics can be added to your app to provide statistics about why your app crashed. Other tools like New Relic offer paid mobile monitoring for your app, allowing you to see data about which mobile devices are working well with your app and which might be having problems.

## **Application Performance Monitoring (APM) Tools**

For more advanced monitoring of your application, you can use an APM tool such as Elasticsearch or Datadog. These tools track every transaction your application processes and can provide insights on CPU usage, server response times, and request errors.

Whether you work for a big company with a web application that has millions of users or a small startup with one little mobile app, performance testing is important. It may mean the difference between happy customers who keep using your application and disappointed users who uninstall it.

# Chapter 56: How to Design a Load Test

Load testing is simply measuring how your application will perform during times of high demand. This could mean testing during normal loads, or it could mean testing during high loads to find the limits of the application.

It's easy to find a load testing tool, create a few tests, and run them with a few hundred users to create metrics. But this isn't particularly useful if you don't know why you are testing or how your results will help you.

So, before you begin a load test, it's important to ask the following questions:

- What kinds of scenarios are you testing for?
- What is the expected behavior in those scenarios?

Let's imagine that you have a new website that sells boxes of chocolates. You have noticed that your site is most active on Saturday mornings. Also, Valentine's Day is coming, and you anticipate that you will have many more orders in the week leading up to that day. In March, your company will be reviewed by a popular cable TV show, and you are hopeful this will lead to thousands of visits to your site.

With this in mind, you come up with the following expected behaviors:

- You would like your web pages to load in less than two seconds under a typical Saturday morning load.
- You would like your site to be able to process 500 orders per hour, which will get you through the busy Valentine's Day season.
- You would like your site to be able to handle 10,000 visitors per hour, which is how many people you expect will visit right after the TV show airs.

The next step is to figure out what test environment you will use. Testing in your production environment would provide the most realistic results, but it would be a bad idea if the tests caused your site to crash! A better option is

a test environment that accurately mimics your production environment in terms of the number of servers used and the size of the backend database. Ideally, this test environment would only be used for your load testing, but this is not always an option; you may need to share this environment with other testers, in which case you'll need to be aware of what kinds of tests they are running and how they will impact you. You'll also want to let other testers know when you are conducting your load tests so that they won't be surprised if response times increase.

Once your test environment is ready, you can conduct some simple baseline tests. You can use some of the strategies mentioned in Chapter 55 to find out how long it takes for your web pages to load and what the typical response times are for your API endpoints. Knowing these values will help you gauge how large an impact a load scenario will have on your application.

Now it's time to design your tests. There are a couple of different strategies to use in this process:

- You can test individual components, such as loading a web page or making a single request to an API.
- You can test entire user journeys: browsing, adding an item to a cart, and making a purchase.

You'll probably want to use both of these strategies, but not at the same time. For instance, you could measure how long it takes to load your website's home page when you have 10,000 requests for the page in one hour. In a separate test, you could create a scenario in which hundreds of users are browsing, adding items to their cart, and making a purchase, and you could monitor the results for any errors.

For each test you design, you'll want to determine the following:

- How many users will be interacting with the application at one time?
- Will the users be added all at once, or every few seconds?
- Will the users execute just one action and then stop, or will they execute the action continuously for the duration of the test?
- How long will the test last?

Let's design a test for the Valentine's Day scenario I mentioned earlier. We'll assume you have created test steps that will load the web page, browse through three product pages, add one product to the cart, and make a purchase. We already mentioned that you'll want to be able to handle 500 orders per hour, so we'll set up the test to do just that. It's unlikely that in a real-world scenario all 500 users would start the ordering process at the same time, so we'll set the test to add a new user every five seconds. Each user will run through their scenario once and then stop. The test will run for one hour, or until all 500 users have completed the scenario.

Before you run your test, be sure that your test steps will return errors if they don't result in the expected response. When I first got started with load testing, I ran several tests with hundreds of requests before I discovered that all my requests were returning an empty set. Because the requests were returning a 200 response, I didn't notice that there was anything wrong! Make sure your steps have assertions that will validate that your application is really behaving as you expect it to.

Once you have the test steps in place, you've made sure the steps have good assertions, and you have your test parameters set up with the number of users, the ramp-up time (how frequently a new user will be added to the test), and the test duration, it's time to run the test. While the test is running, watch your application's response times and CPU usage. If you start seeing errors or high CPU spikes, you can stop the test and note how high the load was when the spikes occurred.

Regardless of whether you need to stop the test early or whether the test completed successfully, you'll want to run a few test passes to make sure your data is fairly consistent. At the end of your testing, you'll be able to answer the question: can my website handle 500 orders in one hour? If all the purchases were completed with no errors and all the response times were reasonable, then the answer is yes. If you started seeing errors or if the response times increased to several seconds, then the answer is no. If the answer is no, you can take the data you collected and share it with your developers, showing them exactly how many users it took to slow down the system.

Load testing is an important step in the performance testing process. To

ensure that load testing is a productive and informative activity, take the time to consider what behaviors you want to measure, how you want your application to behave, what scenarios you can run to test those behaviors, and how you can analyze your results.

# Part VIII: Usability and Accessibility Testing

# Chapter 57: Localization and Internationalization Testing

Internationalization means designing your application so that it can be used in different countries. Localization means adapting software for use in a specific location.

If your app is used anywhere outside your country of origin, chances are it uses some kind of localization strategy. Many people assume that localization simply means translating the text to another language, but this is not the case. Here are some examples of localization that your application might use:

- **Language:** Different countries speak different languages, but different regions can speak different languages as well. Canada is an example of this: in the province of Quebec the primary language spoken is French, and in the other provinces the primary language spoken is English.
- **Spelling:** Even when two areas speak the same language the spelling of words can be different; for example, it's "color" in the US, as opposed to "colour" in Canada and the UK.
- **Words and idioms:** Words can vary, even in a language common to multiple countries. In the UK, a truck is a lorry and a car's trunk is a boot. In the US, to "table" a topic means to stop talking about it until a later meeting. But in the UK and Canada, to "table" a topic means to start talking about it in the current meeting—the opposite of what it means in the US!
- **Currency:** Different countries use different currencies. But this doesn't just mean using a different symbol in front of the currency, like \$ or £; the currencies can also be formatted differently. In the US, fractions of a dollar are separated with a dot and amounts over 1,000 are separated with a comma, while in Spain, it's the opposite.

So what would be written as 1,000.00 in the US would be written as 1.000,00 in Spain.

- **Date and time formats:** In the US, dates are written as month/day/year, but in many other countries dates are written as day/month/year. The US generally writes times using AM and PM, but many other countries use 24-hour time, so what would be 1:00 PM in the US would be 13:00 elsewhere.
- **Units of measure:** The US usually uses US customary units; for example, pounds for weight, and feet and inches for height. Most other countries use the metric system for these measurements. Most countries measure air temperature in Celsius, while the US uses Fahrenheit.
- **Postal codes and phone numbers:** These vary widely from country to country. See the chapters on international phone numbers and postal codes for some examples.
- **Images:** Pictures in an application might need to be varied from country to country. For example, if your application was to be used internationally, you might not want to include a picture of a building with an American flag on the front. Or if your app were to be used in religiously conservative countries, you might not want a picture of a person in a sleeveless shirt.

The first step in localization testing is to determine exactly what will be localized. Your company may decide to localize for date and time, postal codes, and phone numbers, but not for language. Or a mobile app may choose to only use other languages that are built into the device so that the app's text would be in one language but its buttons would be in the user's language.

If your app will be using other languages, gather all the texts you will need to be checking. For example, if your app has menu items such as Home, Search, Your Account, and About Us and your app will be localized for US English, French, and Spanish, find out what those menu items should be in all three languages. It should go without saying that whoever has done the translations should have consulted with a native speaker or a translation

service to make sure the translations are correct.

Next, create a test plan. The simplest way to do this is to create a spreadsheet in which the leftmost column lists the different localization types you need to test and the top row lists the different countries. Here is a very basic example:

|               | United States | Canada- Quebec | Mexico   |
|---------------|---------------|----------------|----------|
| Home Button   | Home          | Accueil        | Inicio   |
| Back Button   | Back          | Retour         | Regresar |
| Save Button   | Save          | Enregistrer    | Guardar  |
| Delete Button | Delete        | Supprimer      | Eliminar |

Once your matrix is created, it should be very simple to run through your tests. If you are testing on mobile, make sure that when you switch your mobile device to a different language you know exactly how to switch it back if you don't recognize the words in the language you are switching to. When I was testing localization for Mandarin this was especially important; since I didn't know any of the characters, I had no idea what any of the menu items said. I memorized the order of the menu items so that I knew which items I needed to click on to get back to English.

Another important thing to watch for as you are testing is that translated items fit well in the app. For example, your Save button might look perfectly fine in English, but in German it could look like this:



Once you have completed your localization testing, you'll want to automate it. Using UI automation tools, you can create a separate test suite for each language in which the setup step would be to set the desired country on the browser or device and each test would validate one aspect of localization, such as that button texts are in the correct language or that you can enter a postal code in that country's format. It would be very helpful to use a visual validation tool to validate that buttons are displaying correctly or that the correct flag icon is displaying for the location.

Localization is a tricky subject, and just like software, it's hard to make it perfect. But if you and your development team clarify exactly what you want to localize and you are methodical in your testing, you'll ensure that a majority of your users will be satisfied with your application.

# Chapter 58: User Experience Testing

User experience (UX) testing refers to whether the app is intuitive and easy to use. Some larger companies have dedicated UX designers on staff whose goal is to make their company's application pleasing to customers. Even if you have UX designers at your company, it's still a good idea to test your application with the user experience in mind.

Here are some ways you can do that.

## **Learn What the Expected User Journeys Are**

Usually when we are testing an application we use it in ways that users won't, focusing on one feature or page at a time. A good strategy is to find out how real users will be using the application and run through those scenarios. For example, if you had an application that allowed users to order tickets for a movie theater, a user journey might be to log in, browse through the movies, select a movie, look at the showtimes for the movie, select a showtime, click a button to place the order, add credit card information, and complete the sale. By running through scenarios like this, you'll discover which areas might not be offering the best user experience.

## **Look for Tasks in the User Journey That Require Many Clicks or Steps**

Could those tasks be completed with fewer clicks? My husband was once searching online for a new car. He went to a website for a car manufacturer and was browsing through the different models. Every time he changed to a new page, he was prompted to enter his zip code again. That's not a great user experience!

## **Test a New Feature Before You Know What It's Supposed to Do**

This is a strategy that doesn't get used much anymore, since these days testers are included in feature planning meetings (as they should be). But I have found that it is sometimes helpful to look at a feature while knowing

very little about it. That is what your customers will be doing, so anything you find frustrating or complicated will probably also be frustrating or complicated for your customers. An alternative to testing without knowing anything about the feature is to get someone who has never used the application to try it out. Spouses, roommates, friends, and people from nontechnical teams in your company are good candidates for these tests. By watching them navigate through your site, you will find the tasks that might not be intuitive.

## **Test Using Only the Keyboard**

People who use applications extensively want to be able to use them as quickly as possible. A good example of this is a customer service representative who has to fill out a form for every person who calls them. If they are typing into each field and they have to keep moving their hand over to the mouse to click the Submit button, this is a waste of their time. If they can instead submit the form with the Enter key, their hands don't need to leave the keyboard.

There are many applications out there, and your application may have dozens of competitors. By ensuring that your app is easy and intuitive, you will make it more likely that customers will choose to use your app over others.

# Chapter 59: Accessibility Testing

Accessibility testing refers to how easy it is for users with limited ability to use the application. This kind of testing is important because 15% of the population has some kind of disability, and we want our applications to be as inclusive as possible. The three main types of accessibility testing you will want to conduct are visual, dexterity, and auditory. Here are some strategies for each.

## Visual Testing

- Is the text large enough for most users? Can users zoom in and enlarge the text if needed?
- Do the images have text descriptions so that visually impaired users who use text-to-speech features will know what the image is?
- Are the colors in the application distinctive enough that colorblind users won't be confused by them? There are helpful websites that allow you to load a screenshot of your application and see what it will look like to a colorblind person.

## Dexterity Testing

- Does your app require any complicated click-and-drag or highlight-and-click scenarios? Imagine how difficult these would be to a person who has limited use of their hands. Can you change the app so that these tasks can be accomplished in a simpler way?
- Are your buttons and links easy to click on? If the buttons are too small, it may be difficult for someone with limited dexterity to click in the correct place.

## Auditory Testing

- Does your application include videos? If so, do the videos have

captions so that those who are deaf or hard of hearing will know what is being said?

- Do parts of your application rely solely on sound effects to tell the user what is happening? Try running the application with the sound turned off. Do you miss any information while you are running through your test scenarios?

A really easy way to check your software for accessibility compliance is to use the WAVE plug-in for Chrome and Firefox: <https://wave.webaim.org>. Simply install the plug-in, navigate to the page you want to test, and turn on the plug-in. You'll get a pop-up window that shows all the accessibility errors, and those errors will be displayed on the page as well. Click on any error to learn more about the accessibility standard.

As software testers, we want our users to have as pleasant an experience as possible. Accessibility testing will help us ensure that our users will be able to accomplish what they want with our apps efficiently and easily.

## Part IX: Software Development Basics

# Chapter 60: Code Like a Developer

I'll be honest: I don't love coding.

Don't get me wrong. I love the feeling of solving a technical challenge and coming up with a great way to automatically assert that software is doing what it's supposed to be doing. I love maintaining and updating my automated test suites. But actually writing code is not my favorite thing to do. Whenever I find myself having to write another nested "for" loop, I sigh inwardly.

However, with all the coding I've done over the years, I've come to really appreciate the work that software developers do! Software is complex stuff, and developers have come up with great ways to set standards, share repositories, and review one another's work.

The test automation code we write is just as important as the code software developers write. Therefore, we should write our code with the same standards the developers use. Here are a few suggestions for coding practices you should adopt.

## **Your Code Should Live in the Same Repository as the Developers' Code**

This is for a few reasons. First, the developers' unit tests reside with the code, so it makes sense to have your integration and UI tests in the same place. Second, it's easier to maintain one repository than it is to maintain two. And finally, having your code in the same place serves to remind the whole team that test automation is everyone's responsibility.

## **Write Clean Code**

When I first got started with test automation, I had absolutely no idea what I was doing. All I had was my manual testing experience and a couple of courses in Java and C++. I did a lot of googling and a lot of guessing as I put together my first Selenium tests. After much work, they ran and (mostly)

passed, but boy, were they lousy! I didn't know anything about how to write clean code. Fortunately, I had great developers around to teach me how to make my code better.

Here are some of the principles of writing clean code:

- **Keep it simple.** Always review your code and ask yourself whether there's a simpler way to do what you are trying to do. Sometimes the obvious solution to a testing problem becomes clear only after you have solved it in a complicated way; now it's time to go back and solve it more elegantly.
- **Don't repeat yourself.** If there's something you're doing in more than one test—for example, logging in to the application—write a method that you can call instead of putting those steps into every test. Similarly, create a file where you save all your variables and element locators, and have all your tests refer to that file. This way, if a variable or a locator changes, you can make the change in one place rather than several.
- **Be consistent.** Consistent code is easier to read. For example, if you use “firstName” as a variable for the user's first name, don't use “LastName” as the variable for the user's last name. Also, follow the conventions your developers are using. If they indent with two spaces, you should too. If they put their opening curly braces on a separate line, you should too.

## Solicit Feedback

Like me, you may not have had a thorough grounding in good coding principles. Some of the best software testers I've had the pleasure of working with did not major in software engineering in college. If you did not go through rigorous training in software development, it's important to get feedback from the developers you work with. On my team, the software testers often review and approve one another's code, but I also like to have my code checked by developers to make sure I'm not doing anything unusual or creating steps that could possibly result in a race condition.

Test automation helps the whole team by speeding up the feedback process and freeing testers to do more exploratory testing. We owe it to our entire team to write quality code that is readable, runs quickly and consistently, and provides valuable feedback.

# Chapter 61: Command-Line Basics

The command line is hugely helpful when you want to navigate through your system's folder structure, create new folders or files, or execute runnable files. In this chapter, I'll walk you through some simple commands that can help you get started using the command line like a pro. Most of the commands I'll share will work in both Mac and Windows environments; when there are differences between the two, I'll point them out.

First, let's look at some useful keys.

## **The Up Arrow**

The up arrow copies whatever command you just ran, and if you click on the up arrow more than once, you can cycle back through all the commands you have run so far.

For example, if you ran these three commands:

```
ls  
cd Documents  
cd ..
```

and then you clicked the up arrow, you'd see `cd ..`. If you clicked the up arrow again, you'd see `cd Documents`, and if you were to click it a third time, you'd see `ls`.

The up arrow is really helpful when you need to run a complicated command more than once but you don't feel like retyping it. Simply click the up arrow until you've returned to the command you want, and then click Return to run the command again.

## **The Tab Key**

The tab key has auto-complete functionality. To see how this works, let's imagine you have a folder called MyFolder that contains three subfolders:

```
LettersToDad  
LettersToMom  
graduationPics
```

If you wanted to navigate from MyFolder to graduationPics using the **cd** command (more on this later), you could simply type:

```
cd grad
```

and then press the tab key. The folder name will auto-complete to [graduationPics](#).

This command is helpful when you don't feel like typing out an entire folder name. Typing just the first few letters of the folder and hitting tab, then Return, is a really fast way to navigate.

For the auto-complete to work, you need to type enough letters that there's only one possible option left when you hit the tab key. For example, when you type

[cd LettersTo](#) and then hit the tab key, the command line doesn't know whether you mean [LettersToDad](#) or [LettersToMom](#). The Windows command line will cycle through the possible options as you repeatedly hit the tab key. In macOS, if you hit the tab key a second time, it will return your possible options.

Next, let's learn some navigation skills.

## The Command Prompt

The command prompt is a symbol indicating that the command line is ready to receive commands. On a Mac, the command prompt looks like this: [\\$](#). In Windows, the command prompt looks like this: [>](#). The command prompt is preceded by the working directory.

## The Working Directory

The term "working directory" refers to whatever directory (folder) you are in when you are using the command line. When you first open the

command-line window, you'll be in your home directory. This is your own personal directory. For example, on my Windows machine, my working directory is `C:/users/kjackvony`. On my Mac, my working directory is `/Users/K.Jackvony`, but the directory location displays only as `~`, which is a symbol that means the home directory.

## The `ls /dir` Command

This command—`ls` (the first letter is a lowercase L) in Mac and `dir` in Windows—will list all the files and folders in your working directory.

### `cd <folder name>:`

This command will change your working directory to the folder you specify. For example, if you entered `cd Documents`, your working directory would change to the `Documents` folder.

### `cd ..:`

This command moves you up one level in the directory. Let's look at an example. I am using my Mac, so I'll use `ls` rather than `dir`.

1. I begin in my home directory:

`~$`

2. I type `ls` to see what's in my home directory and I get this response:

`Desktop`  
`Documents`  
`Pictures`  
`Projects`

3. I type `cd Documents`, and my working directory is now the `Documents` folder:

`Documents$`

4. I type `ls` to see what's in my `Documents` folder and I get this response:

Blog Post Notes  
Images for Testing  
ProfilePhoto.jpg

5. I type `cd "Blog Post Notes"` (I'm using quote marks because the directory name has spaces in it), and my working directory is now the **Blog Post Notes** folder:

Blog Post Notes\$

6. I type `cd ..` and I'm taken back to the **Documents** folder:

Documents\$

7. I type `cd ..` again and I'm taken back to my home folder:

~\$

Now let's try an exercise to practice everything you've learned.

Let's start by creating a new folder. Open the command window and enter `mkdir MyNewFolder`. You won't get any kind of response, just a new command prompt. But if you type `ls` (on Mac) or `dir` (on Windows), you'll now see **MyNewFolder** listed in the contents of your home directory.

To navigate to your new directory, type `cd MyNewFolder`. Your command prompt will now look like **MyNewFolder\$** on a Mac and **MyNewFolder>** on Windows. If you type `ls` or `dir` now, you'll get no files in response because your folder is empty.

Let's put something in that new folder. If you are using a Mac, type `nano MyNewFile`. A text editor will open in the command window. If you are using Windows, type `notepad MyNewFile.txt`. Notepad will open in a separate window.

Type **This is my new file** in the text editor (Mac) or in Notepad (Windows). If you are using Windows, just save and close the Notepad file. If you are on a Mac, click `Control+X`, type `Y` when asked whether you want to save the file, then press the Return key to save the file.

If you are on a Windows machine, return to the command line; Mac users should already be there and the text editor should have closed. Your working directory should still be `MyNewFolder`. Now when you type `ls` (Mac) or `dir` (Windows), you should get this response: `MyNewFile` (Mac) or `MyNewFile.txt` (Windows). You have successfully created a new file from the command line.

We can now read the contents of this file from the command line. If you are on a Mac, type `cat MyNewFile`. If you are on Windows, type `MyNewFile.txt`. Your new file should open.

Now let's learn how to delete a file. Simply type `rm MyNewFile` if you are on a Mac or `del MyNewFile.txt` if you are on Windows. If you've deleted correctly, an `ls` or `dir` command will now give you an empty result.

Finally, let's delete the folder we created. You can't have the folder you want to delete as your working directory, so we need to move one level up by typing `cd ..`. Now you should be in your home directory. If you are on a Mac, type `rm -r MyNewFolder`. If you are on Windows, type `rmdir MyNewFolder`. You won't get any response from the command line, but if you enter `ls` or `dir`, you'll see that the folder has been deleted.

Now you know how to create and delete files and folders from the command line. I'll close by adding two bonus commands: one for Mac and one for Windows.

**For Mac users:** `sudo` (which stands for superuser do) allows you to run a command as an administrator. At times, you may need to do an installation or edit that requires administrator access. By putting `sudo` before the command, you can run the command as the system admin. For example, if you typed `sudo rm -r MyNewFolder`, you'd be removing the folder as the system admin. Think carefully before you use this command, and make sure you know what you are doing. There are many commands that require a superuser to execute them because they are dangerous. You don't want to delete your entire filesystem, for example!

**For Windows users:** A handy command is `explorer`. Typing this in your

command window will bring up the File Explorer. This is useful when you want to switch from navigating the folder structure in the command line to navigating in the Explorer window. For example, if you knew there was a folder called MyPictures with images in it, you might want to open the Explorer window to take a look at the thumbnails of those images.

Have fun using your newly learned command-line skills!

# Chapter 62: Coding Definitions

One of the things I struggled with when learning to code was knowing the difference between a class, an object, and a method. Through personal experience and web searches, I have come up with the following explanation.

A class is the basic building block of software. It consists of variables and methods.

Here's an example of a class:

```
class Cat { //this is the class  
    //these are the variables in the Cat class  
    var name;  
    var age;  
    var weight;  
    var color;  
  
    //these are the methods in the Cat class  
    eat() {  
    }  
    sleep() {  
    }  
    meow() {  
    }  
} //this marks the end of the Cat class
```

A variable is a little bit of memory that stores a value to be used in a program. A string is a bit of text, like "Hello!" or "rainbow". In the Cat class, the string variables we have are name (the name of the cat) and color (the color of the cat). An int is an integer, like 7 or 13. In the Cat class, the int variables we have are age (the age of the cat) and weight (the weight of the cat).

An object is an instance of a class. If we want our program to create and use a cat instance, we need to create a Cat object:

```
//this is where we create the new Cat object  
var Fluffy = new Cat() //Fluffy is the variable name given to the Cat  
object  
}
```

A method is a task or group of tasks. If you are familiar with mathematical functions, you might want to think of a method as a function. In the Cat class, the methods are eat(), sleep(), and meow(). The parentheses after the method name indicate what kind of parameter will be passed to the method. In this case, eat(), sleep(), and meow() do not require a parameter, so the parentheses are empty.

# Chapter 63: Object-Oriented Programming

Much test automation is written in an object-oriented programming (OOP) language such as Java, JavaScript, or C#. Having grown up before OOP, I had a particularly tough time learning it. I hope the following explanation will make things easier for you!

Think about the tasks you perform when you get ready in the morning. You might exercise, shower, and get dressed. If you were writing a program, or class, to tell a robot to do these things, you would probably write a little subroutine, or method, for each task. For example:

```
class getReady { //this is the class
    exercise(); //this calls the exercise method
    shower();
    getDressed();

    exercise() { //this is the exercise method
        Pushups;
        Situps;
        Squats;
    }

    shower() { //this is the shower method
        get in shower;
        Lather;
        Rinse;
        Repeat;
    }

    getDressed() { //this is the get dressed method
        put on shirt;
        put on pants;
        put on shoes;
    }
}
```

Now consider your friend, who exercises and showers at night instead of in the morning. His nighttime routine is exercise, shower, put on pajamas. If you were programming a robot to do your friend's nighttime activities, your code might look like this:

```
class getReadyForBed { //this is the class
exercise();
shower();
putOnPajamas(); //this calls the put on pajamas method

exercise() {
    Pushups;
    Situps;
    Squats;
}
shower() {
    get in shower;
    Lather;
    Rinse;
    Repeat;
}

putOnPajamas() { //this is the put on pajamas method
    put on pajama top;
    put on pajama bottoms;
}
}
```

Notice that the two classes have two methods that are exactly the same. Why not share the code instead of copying and pasting it into each class? This way, if you ever need to make a change to your exercise method, you will only have to change it once.

Let's make a base class that contains all the methods we might need:

```
class Ready { //this is our base class
exercise() {
    Pushups;
```

```
    Situps;  
    Squats;  
}  
  
shower() {  
    get in shower;  
    Lather;  
    Rinse;  
    Repeat;  
}  
  
getDressed() {  
    put on shirt;  
    put on pants;  
    put on shoes;  
}  
  
putOnPajamas() {  
    put on pajama top;  
    put on pajama bottoms;  
}  
}
```

Now we can change our two programs to call the existing methods instead of copying and pasting them:

```
class getReady extends Ready { //this is your routine  
exercise();  
shower();  
getDressed();  
}
```

```
class getReadyForBed extends Ready { //this is your friend's routine  
exercise();  
shower();  
putOnPajamas();  
}
```

See how much space this saves? Object-oriented programming makes code much more reusable and maintainable.

# Chapter 64: Passing Parameters

When I first started learning object-oriented programming, I was fairly perplexed by the parentheses found after every method. Sometimes they were empty and sometimes they had values in them, and I didn't understand why. I've found it's easiest to think about parameters and methods in terms of input and output. Here's an example:

```
Square (input) {  
}
```

`Square` is the name of the method. The word `input` is what I'm calling the variable that I'm going to pass into the method. This method is going to take a number, square it, and return the result of the squaring.

Now let's add the steps for the method:

```
Square (input) {  
var output = input * input;  
return output;  
}
```

This function takes the variable called `input`, squares it, sets it to equal a variable called `output`, and then returns the output.

It's also possible to have more than one input parameter:

```
Square (input, name) {  
var output = input * input;  
var result = "Hi " + name + "! Your result is " + output;  
return result;  
}
```

This is a little more confusing, so I will explain it in more detail. The first line of the method is taking the `input` variable, squaring it, and setting it to a variable called `output`, just as it did before. But in the second line, we're

creating a string. The characters in between the quote marks are entered into the string. These are concatenated (added) with our int output and the string name that we entered in the parameters, using plus signs to add the characters:

```
"Hi " + name + "! Your result is " + output
```

So if I called the function like this:

```
Square(2, Kristin)
```

**t**he result string would be set to:

```
Hi Kristin! Your result is 4
```

If my program wanted to output the result, I would add the instruction:

```
console.log(result)
```

And my result would be printed to the console.

Is it possible to have a method that doesn't pass in any parameters, but still returns something? Sure!

```
MyParameterlessMethod () {  
    var result = 4;  
    return result;  
}
```

This method assigns the number 4 to the variable called **result**, and then returns the result.

# Chapter 65: Setting Up Node

The hands-on activities found in Part IX and Part X all require Node.js. I'm a big fan of Node because JavaScript, which Node is based on, is such a versatile language, and because Node provides an easy way to install modules that can be used in a Node program.

You can check whether you have Node installed by opening a command window and typing `node --version`. If you get a version number as a response, you already have Node installed.

If you don't have Node installed, you can go to <https://nodejs.org/en/download> and download the appropriate version for your operating system. Or if you have a Mac, you can install Node with this command: `brew install node`.

Once you have Node installed, you should make sure it can be found on your PATH. The PATH is a set of directories that show your computer's command line how to locate programs you have installed. If you are using a Mac, type `sudo nano /etc/paths` and make sure `/usr/local/bin` is listed in the file. If it isn't, you can add it to the file, then save and exit. If you are using Windows, go to Advanced System Settings, click Environment Variables..., select the Path variable in the System Variables, click Edit, and add the path to Node. The path may be `C:\Program Files\nodejs`, but this might vary, so check to see whether that is where Node is installed.

Next, reboot your computer to make sure your path settings have taken effect, then confirm that you have Node installed by typing `node --version`. You should now see the version number for Node in the response. You can also check to make sure npm (the Node Package Manager) has been installed by typing `npm --version`.

You are now ready to start using Node!

# Chapter 66: Arrow Functions

Arrow functions have been around for a few years now, but I've always been confused by them because they weren't around when I was first learning to write code. You may have seen these functions, which use the symbol `=>`. They seem so mysterious, but they are actually quite straightforward! Arrow functions are simply a way to notate a function to save space and make code easier to read. I'll walk you through an example. We'll start with a simple traditional function:

```
const double = function(x) {  
    return x + x  
}
```

`double` is the name of the function. When `x` is passed into the function, `x + x` is returned. So, if I called the `double` function with the number `3`, I'd get `6` in response.

Now we're going to replace the function with an arrow:

```
const double = (x) => {  
    return x + x  
}
```

Note that the arrow comes after the `(x)` rather than before it. Even though the order is different, `function(x)` and `(x) =>` mean the same thing.

Now we're going to replace the body of the function `{ return x + x }` with something simpler:

```
const double = (x) => x + x
```

When arrow functions are used, it's assumed that what comes after the arrow is what will be returned. So, in this case, `x + x` means the same thing as `{ return x + x }`.

You can try running these three functions for yourself if you have Node installed. Using a code editor, simply create a file called `app.js` with the first version of the function:

```
const double = function(x) {  
  return x + x  
}
```

and in the next line, add a logging command:

```
console.log(double(3))
```

Navigate in the command line to the file's location, run the file from the command line with `node app.js`, and the number `6` will be returned in the console.

Then replace version 1 of the function with version 2:

```
const double = (x) => {  
  return x + x  
}
```

Run the file, and you should get a `6` again. Finally, replace version 2 with version 3:

```
const double = (x) => x + x
```

and run the file; you should get a `6` once again.

It's even possible to nest arrow functions! Here's an example:

```
const doublePlusTen = (x) => {  
  const double = (x) => x + x  
  return double(x) + 10  
}
```

The `const double = (x) => x + x` is our original function. It's nested inside a `doublePlusTen` function. The `doublePlusTen` is using curly braces and a `return` command because there's more than one line inside the function

(including the double function). If we were going to translate this nested function into plain English, it would look something like this:

“We have a function called doublePlusTen. When we pass a number into that function, first we pass it into a nested function called double, which takes the number and doubles it. Then we take the result of that function, add 10 to it, and return that number.”

You can try out this function by calling adding the line `console.log(doublePlusTen(3))` and running the app again, and you should get `16` as the response.

Hopefully this information will help you understand what an arrow function is doing the next time you encounter it in code.

# Chapter 67: Promises

Have you ever written an automated UI test that uses JavaScript, and when you went to assert on a response, you got **Promise {pending}** instead of what you were expecting? This really frustrated me when I first encountered it! A developer I was working with explained that this is because JavaScript processes commands asynchronously through the use of promises. I sort of understood what he meant, so I tried to work with it as best I could, but I didn't really get it. Since then, I've gotten a better grasp on the concept. In this chapter, I'll explain why JavaScript needs promises and show an example of how they work.

JavaScript needs promises because it is a single-threaded language, meaning it can only do one thing at a time. For example, if we wanted a program to do three things, such as make an HTTP request, alphabetize a list, and update a record in a database, we wouldn't want to have to wait around for each of those tasks to finish before we went on to the next one, because our program would be very slow! So JavaScript is designed to be asynchronous—it can start a task, and then while it's waiting for that task to complete, it can start the next task.

Our example program from the preceding paragraph might actually run like this:

- Start the HTTP request.
- Start alphabetizing the list.
- Start updating the record in the database.
- Finish alphabetizing the list.
- Finish updating the record in the database.
- Finish the HTTP request.

JavaScript would manage this through the use of promises. Let's take a look at a promise:

```
const sumChecker = new Promise((resolve, reject) => {
```

```
if (a+b==c) {
    resolve('You are correct!')
}
else {
    reject('Sorry, your math is wrong.')
}
})
```

The function called `sumChecker` is a promise. It's going to have two possible results: `resolve` and `reject`. If the sum is correct it resolves the promise, and if it's incorrect it rejects it. All promises behave in this way; there will be an option to resolve the promise and an option to reject the promise.

When the promise is called, either `resolve` or `reject` will be returned; you can never return both. Let's look at an example of calling the promise:

```
sumChecker.then((result) => {
console.log('Success!', result)
}).catch((error) => {
console.log('Error!', error)
})
```

The result that is returned will be either `resolve` or `reject`. If the result is `resolve`, then the program knows to continue and will return the `resolve` message. If the result is `reject`, then the program knows to throw an error and will return the `reject` message.

You can try this for yourself if you have Node installed! Simply copy the promise and the call and paste them into your favorite code editor. Then, at the beginning of the file, add these lines:

```
var a = 1
var b = 2
var c = 3
```

Your complete program should look like this:

```
var a = 1
var b = 2
var c = 3

const sumChecker = new Promise((resolve, reject) => {
if (a+b==c) {
    resolve('You are correct!')
}
else {
    reject('Sorry, your math is wrong.')
}
})

sumChecker.then((result) => {
console.log('Success!', result)
}).catch((error) => {
console.log('Error!', error)
})
```

Save the file with the name `myfile.js`, navigate in the command line to the file's location, and run the file with the command `node myfile.js`. You should see this response:

**Success! You are correct!**

If you make a change to the `c` variable and set it to `4`, and then you save and run the command again, you'll see this response:

**Error! Sorry, your math is wrong.**

Let's put a `log` statement in between the promise and the call to the promise that looks like this, `console.log(sumChecker)`, so that we can see the state of `sumChecker` before we call it. Your program should now look like this:

```
var a = 1
var b = 2
var c = 3
```

```

const sumChecker = new Promise((resolve, reject) => {
  if (a+b==c) {
    resolve('You are correct!')
  }
  else {
    reject('Sorry, your math is wrong.')
  }
})

console.log(sumChecker)

sumChecker.then((result) => {
  console.log('Success!', result)
}).catch((error) => {
  console.log('Error!', error)
})

```

If we change the value of `c` back to `3` so that we'll get a positive result, save the file, and run the program with `node myfile.js` now, we'll get the result `Promise { 'You are correct!'}` in addition to the response we got earlier. That seems easy! But the reason why we get the promise resolved so quickly is because the `sumChecker` promise executes quickly. Let's see what happens if we make the `sumChecker` work more slowly, like a real promise would.

Update the `sumChecker` promise to look like this:

```

const sumChecker = new Promise((resolve, reject) => {
  if (a+b==c) {
    setTimeout(() => {
      resolve('You are correct!')
    }, 2000)
  }
  else {
    reject('Sorry, your math is wrong.')
  }
})

```

All we're doing here is adding a two-second timeout to the resolved

promise. Save the file, and run the program again with `node myfile.js`. This time you'll first get the result `Promise { <pending> }`, and after two seconds you'll get the result `Success! You are correct!`

Now it should be clear why you get `Promise { <pending> }` when you are making JavaScript or Node calls. It's because the promise hasn't been completed yet. This is why we use the `.then()` command. We wait for the response to the call to come back, and then we do something with the response. If we're writing a test, at that point we can assert on our result.

I hope you'll take the time to try running this file with Node because there's nothing quite like doing hands-on work to generate understanding. You can try changing the variables or any of the response messages to get a feel for how it's working. Here's the final version of the file if you'd like to copy and paste it:

```
var a = 1
var b = 2
var c = 3

const sumChecker = new Promise((resolve, reject) => {
  if (a+b==c) {
    setTimeout(() => {
      resolve('You are correct!')
    }, 2000)
  } else {
    reject('Sorry, your math is wrong.')
  }
})

console.log(sumChecker)

sumChecker.then((result) => {
  console.log('Success!', result)
}).catch((error) => {
  console.log('Error!', error)
})
```

# Chapter 68: Async/Await

In the preceding chapter, I explained promises. The basic idea of promises is that JavaScript functions happen asynchronously, so promises are like place savers that wait for either a resolve or reject response.

Here's an example of a function with a promise:

```
const add = (a, b) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(a+b)
    }, 2000)
  })
}
```

In this function, we're simply adding two numbers. A `setTimeout` for two seconds is added to make the function take a little longer, as a real function would.

Here's what calling the function might look like:

```
add(1, 2).then((sum) => {
  console.log(sum)
})
```

The `add` function is called, and then we use the `then()` command to log the result. Using `then()` means we're waiting for the promise to be resolved before we go on.

But what if we needed to call the function a second time? Let's say we wanted to add two numbers, then we wanted to take that sum and add a third number to it. For this, we'd need to do promise chaining.

Here's an example of promise chaining:

```
add(1, 2).then((sum) => {
```

```
add(sum, 3).then((sum2) => {
    console.log(sum2)
})
})
```

The add function is called, and we use the `then()` command with the sum that is returned. Then we call the function again, and use the `then()` command once again with the new sum that is returned. Finally, we log the new sum.

This works just fine, but imagine if we had to chain a number of function calls together. It would start to get pretty tricky with all the `then()`s, curly braces, and indenting. This is why `async/await` was invented!

With `async/await`, you don't have to chain promises together. You create a new `async` function call, and then you use the `await` command when you are calling a function with a promise.

Here's what the chained promise call would look like if we used `async/await` instead:

```
const getSum = async () => {
const sum = await add(1, 2)
const sum2 = await add(sum, 3)
console.log(sum2)
}
getSum()
```

We're creating a new `async` function called `getSum`, by using the `async` command

`const getSum = async () =>`. In that function, we're first calling the `add` function with an `await`, and we're setting the result of that call to the variable called `sum`. Then we're calling the `add` function again with an `await`, and we're setting the result of that call to the variable called `sum2`. Finally, we're logging the value of `sum2`. Now that the `async` function has been created, we're calling it with the `getSum()` command.

It's pretty clear that this code is easier to read with `async/await`! Keep in mind that promises are still being used here; the `add()` function still returns a

promise. But `async/await` provides a way to call a promise function without having to add in a `then()` statement.

# Chapter 69: Debugging for Testers

Wikipedia defines debugging as “the process of finding and resolving defects or problems within a computer program that prevent correct operation of computer software or a system.” Often we think of debugging as something that only developers need to do, but this isn’t the case. Here are two reasons why: first, investigating the cause of a bug when we find it can help the developer fix it faster; and second, since we write automation code ourselves and we want to write code that is of high quality just like developers do, we ought to know how to debug our code.

Let’s take a look at three different strategies we can employ when debugging code.

## Strategy #1: Console Output

Code that is executed in a browser or on a device generally outputs some information to the console. You can easily see this by opening DevTools in Chrome or the Web Console in Firefox. When something goes wrong in your application, you can look for error messages in the console. Helpful error messages like “The file ‘address.js’ was not found” can tell you exactly what’s going wrong.

Often an error in an application will produce a stack trace. A stack trace is simply a series of error statements that go in order from the most recent file that was called all the way back to the first file that was called. Here’s a very simple example: let’s say you have a Node application that displays cat photos. Your main *app.js* page calls a function called *getCats* which will load the user’s cat photos. But something goes wrong with *getCats*, and the application crashes. Your stack trace might look something like this:

```
Error: cannot find photos
at getDocs.js 10:57
at app.js 15:16
at internal/main/run_main_module.js:17:47
```

The first line of the stack trace is the error: the main cause of what went wrong.

The next line shows the last thing that happened before the app crashed: the code was executing in *getCats.js*, and when it got to line 10, column 57, it couldn't find the photos.

The third line shows which file called *getCats.js*: it was *app.js*, and it called *getCats* at line 15, column 16.

The final line shows what file was called to run *app.js* in the first place: an internal Node file that called *app.js* at line 17, column 47.

Stack traces are often longer, harder to read, and more complicated than this example, but the more you practice looking at them, the better you will get at finding the most important information.

## Strategy #2: Logging

Much of what you see in the console output can be called logging, but there are often specific log entries set up in an application's code which record everything that happens in the application. I'm fortunate to work with great developers who are adept at creating clear log statements that make it easy to figure out what happened when things went wrong.

Log statements often come with different levels of importance, such as Error, Warning, Info, and Debug. An application can sometimes be set to only log certain levels of statements. For example, a production version of an application might be set to only log Errors and Warnings. When you're investigating a bug, it may be possible to increase the verbosity of the logs so that you can see the Info and Debug statements as well.

You can also make your own log statements, simply by writing code that will output information to the console. I do this when I'm checking to make sure my automation code is working like I'm expecting it to. For example, if I had a do-while statement like this:

```
do {  
    counter++  
}
```

```
while (counter < 10)
```

I might add a logging statement that tells me the value of counter as my program progresses:

```
do {  
    console.log ("The value of counter right now is: " + counter)  
    counter++  
}  
while (counter < 10)
```

The great thing about creating your own log statements is that you can set them up in a way that makes the most sense to you.

### Strategy #3: Breakpoints

A breakpoint is a place you set in the code that will cause the program to pause. Software often executes very quickly, and it can be hard to figure out what's happening as you're flying through the lines of code. When you set a breakpoint, you can take a look at what all your variable values are at that point in the program. You can also step through the code slowly to see what happens at each line.

Debuggers are generally available in every language in which you can write code. Here are some examples:

- Python uses the *pdb* library
- JavaScript uses the debugger statement
- Powershell uses breakpoints
- C# has debugging tools in Visual Studio
- Java has debugging tools in Eclipse and IntelliJ

By using the console output, logging statements, and breakpoints, you will get all kinds of information that will help you debug code.

# Chapter 70: Seven Steps to Solve Any Coding Problem

I am not the world's greatest coder, although I am getting better every year. One thing I'm really improving on is my ability to solve coding problems. I'm not talking about the coding challenges you'd get online or in a job interview; I'm talking about real-world problems, like "How are we going to create an automated test for this?" Here are the seven steps I use to solve any coding problem.

## **Step #1: Remember What Problem You Are Trying to Solve**

When you're trying to figure out how to do something, it can be easy to forget what your original intent was. For example, let's say you are trying to access a specific element on a web page, and you're having a really tough time doing so; perhaps the element is in a pop-up that you can't reach or it's blocked by another element. It's easy to get so bogged down in trying to solve this problem that you forget your original intent: to add a new user to the system. When you remember this, you realize you could have added a new user to the system by calling the database directly, avoiding the issue you were stuck on!

## **Step #2: Set Small Steps**

I often have what I want to do in my code figured out long before I know how I'm going to do it. And I used to just write a whole bunch of code even if I wasn't sure it was going to work correctly. Then when I tried to run the code, it didn't work; but I wrote so much code that I didn't know whether I had one problem or many. This is why I now set small steps when I code. For example, when trying to write an email test, I will first set for myself the goal of just reaching the Gmail API. I won't care what kind of token I use or what information I get back; I just want a response. Once I solve that, I will work on trying to get the specific response that I want. This strategy also keeps me from getting frustrated or overwhelmed.

## **Step #3: Change One Thing at a Time**

This step is similar to Step #2, but it's good for times when your code isn't working. It's very tempting to thrash around and try a number of different solutions, sometimes all at once, but that's not very helpful. Even if you get your code to work by following this method, you won't know which change caused the code to work; therefore, you won't know which changes were superfluous. It's much better to make one small change, see whether it works, remove that change and try a different change, and so on. Not only will you solve your problem faster this way, but you'll be learning as you go, and what you learn will be very valuable for the next time you have a problem.

## **Step #4: Save All Your Work**

I learned this one the hard way when I was first writing UI automation. I had absolutely no idea what I was doing, and sometimes I'd try something that didn't completely work and then delete it and try something else. Then I'd realize I needed some of the lines of code from the first thing I tried, but I had deleted them, so I had to start from scratch to find them again. Now when I'm solving a new coding challenge I create a document that I call my scratch pad, and when I remove anything from my code I copy it and paste it in the scratch pad, just in case I'll need it again. This has helped me solve challenges much more efficiently.

## **Step #5: See What Others Have Done**

People who are good at solving coding problems are usually also masters of Google Fu: the art of knowing the right web search to use to get them the answers they need. When I first started writing test automation, I was not very good at Google Fu, because I often wasn't sure what to call the thing I wanted to do. As I've grown in experience, I've become better at knowing the terminology of whatever language I'm using, so if I've forgotten something like whether I should be using a static method, I can structure my search so that I can quickly find the right answer. The answers you find on the internet are not always the right ones, and sometimes they aren't even good ones, but they often provide clues that can help you solve your problem.

## **Step #6: Level Up Your Skills**

In 2020 I took an online course on NodeJS, and it helped me gain a deep understanding of how JavaScript works. Now that I understand more, writing code is so much easier. Rather than just copying and pasting examples from someone on Stack Overflow, I can make good decisions about how to set things up; and when I understand what's going on, I can write code so much faster. Take some time to really learn a coding language; it's an investment that will be worth it!

## **Step #7: Ask for Help**

If you've finished all the other steps and still haven't solved your problem, it's time to ask for help. This should definitely not be Step #1 in your process. Running for help every time something gets hard will not make you a better coder. Instead, imagine there's no one who can help you and see how far you can get on your own. See what kinds of lessons you can learn from the process. Then, if you do need to ask for help, you'll be able to accurately describe the problem in such a way that your helper will probably be able to give you some answers very quickly. You'll save them time, which they will appreciate.

Coding is not magic: while there are all sorts of complex and weird things out there in the world of software, an answer exists for every question. By using these seven steps, you'll take some of the mystery out of coding and become a better thinker in the process!

# Chapter 71: Introduction to Git

For a software tester who has just started writing test automation, using version control software such as Git can seem daunting and confusing. But being able to pull down the latest code, update it, and submit a pull request is very important for any team project! In this chapter, I'll provide a gentle introduction to the basics of Git.

## **What Is Git?**

Git is a version control system, which is a system that allows a group of people to collaborate on code without accidentally overwriting one another's work. It also allows the group to keep track of who changed the code and when it was changed so that it's easy to trace back to the source of a problem.

## **Why Is Git Needed?**

Consider what file editing is like when you don't use a version control system. Let's say you have a recipe for brownies. You send the recipe to your friend, and she decides to change the amount of cocoa in the recipe. When she makes that change, it is only in her version of the file, not yours. Your files are now different. If you make a change to add more vanilla to the recipe, now your versions have diverged even further.

You can see how this would be unacceptable for software code! In a version control system, there is one "main version" which is the accepted version of the code. This main version lives in GitHub (or another version control hosting service) and can be "pulled down" by any user. When someone wants to make a change to the code, they pull down the main version of the code, create a "branch" that is a copy of the main version, make their changes to the branch, push the branch up to GitHub, and then do a "pull request," which is asking for someone to review their code and merge it into the main branch.

Confused? Don't worry, this will look much simpler with an example. Let's imagine we have a source code repository called "The Thinking Tester

Guestbook". We'll take a look at what would happen if Prunella Prunewhip wanted to add her name to the guestbook. (These instructions assume that Prunella has already installed [Git](#) on her computer and has already created a [GitHub account](#).)

### **Step #1: Prunella Clones the Source Code Repository**

This is often called "cloning the repo" or "pulling down the repo." Prunella does this by going to the URL in GitHub that has the source code, and clicking the green Code button. A pop-up window appears with the URL she will need to clone the source code. She clicks the little clipboard button to the right of the URL to copy the URL text.

Prunella opens a command window and navigates to the folder where she would like to put the source code. Once she's there, she types `git clone`, pastes the URL text next to those words, and hits Return. The repository is copied from GitHub into a new folder.

Now that the repository is in a folder on her computer, she can open the folder in her file browser and take a look at what's in there. She sees that there is one text file, called *guestBook.txt*. The text file reads:

[Kristin Jackvony was here on May 11, 2021](#)

### **Step #2: Prunella Makes a New Branch and Adds Her Changes to That Branch**

Before Prunella makes any changes to *guestBook.txt*, she should create a new branch and switch to it. So, in the command line, she navigates to the new folder that was cloned earlier by typing `cd ThinkingTesterGuestBook`. She can verify that she's in the main branch by typing `git status`, and she will get a response like this: [On branch main](#).

Now she can create a new branch and switch to it by typing `git checkout -b NewEntry`. The command `-b` tells Git to create a new branch. `NewEntry` is what Prunella has chosen to name her branch. And the command `checkout` is what causes Git to switch to the new branch.

If Prunella types `git status` at this point, she will get `On branch NewEntry` as a response.

Now that Prunella is in the correct branch, she's going to make a change to the `guestBook.txt` file by adding one line so that the file now reads:

```
Kristin Jackvony was here on May 11, 2021  
Prunella Prunewhip was here on June 13, 2021
```

### Step #3: Prunella Commits Her Changes and Pushes Them to GitHub

Now that Prunella has made the change she wanted, she needs to commit and push her change. First, she can run `git status` and she'll get this response:

```
On branch NewEntry  
modified: guestBook.txt
```

This shows that the `guestBook.txt` file has been modified. Next, Prunella needs to add the file to the commit by typing `git add guestBook.txt`. Now if she types `git status`, she'll see this response:

```
On branch NewEntry  
Changes to be committed:  
modified: guestBook.txt
```

Next, Prunella commits her change by typing `git commit -m "Adding a new entry"`. The `-m` in this command stands for “message”. The “`Adding a new entry`” text is the message she is adding to explain what she is committing. The command line will respond with how many files and lines were changed.

Once the change has been committed, Prunella can push the change up to the GitHub repository by typing `git push origin NewEntry`. The `NewEntry` value explains that the code should go up to the `NewEntry` branch, which doesn't exist yet in the GitHub repository but will be created with this command. `Origin` refers to the GitHub repository (this is also referred to as “remote”). The command line will respond with several lines, the final of

which will be \* [new branch] NewEntry -> NewEntry, which shows that a new branch called NewEntry has been created in the origin and that it was copied from the local branch Prunella created, which was also called NewEntry.

### **Step #4: Prunella Creates a Pull Request in GitHub**

Now that her new branch has been pushed up to GitHub, Prunella can submit a pull request to ask that her changes be merged with the main branch. She does this by going to the GitHub repository and choosing the Pull Requests tab, then clicking the New Pull Request button. This takes her to the Compare page. She makes sure the left side of the comparison is the main branch, and then chooses the NewEntry branch from the branch dropdown. She can see how the *guestBook.txt* file has changed; the new line she added is highlighted in green, illustrating the difference between the two files. (If she had deleted a line, the line she removed would be highlighted in red.) Finally, she clicks the Create Pull Request button.

### **Step #5: Prunella's Pull Request Is Approved and Merged**

The final step in the file change process is that the owner of the repository (or any teammates who have approval permissions) will review the change, approve it, and merge it. Now if Prunella changes directories to the main branch by doing `git checkout main`, pulls down the changes by doing `git pull origin main`, and takes a look at the *guestBook.txt* file, she will see that her entry has been added:

[Kristin Jackvony was here on May 11, 2021](#)

[Prunella Prunewhip was here on June 13, 2021](#)

And that's all there is to it! In the next two chapters you'll learn some more Git tips and tricks.

# Chapter 72: Six Tips for Git Success

Even when you understand how Git works, it can still be a bit mysterious because there is so much happening that you don't see. The command line does not offer a visual interface to show you what branch you are on or when you last pulled from the main branch. So, in this chapter I'll provide six tips that make using Git easier.

## **Tip #1: Run git status Frequently**

A common mistake that Git users make is to do a bunch of work and commit while on the wrong branch. Because I am not a Git expert and I'm never sure how to recover from this mistake, I run `git status` frequently. I always run it as soon as I've opened the command line and navigated to my repository so that I won't be surprised by what branch I'm on.

Running `git status` is also a good way to find out what files have been changed in your branch. Sometimes we make a change to a file for debugging purposes and we forget to change it back. If you run `git status`, it will alert you that a file you didn't want to change has been altered.

## **Tip #2: Pull from the Main Branch Before You Do Anything**

Before you start changing code in your branch, you'll want to make sure your branch has the very latest code in it. Otherwise, you may be updating code that doesn't exist anymore, or you may be duplicating work. By making sure you have the latest code, you will also avoid creating a merge conflict, a situation in which your branch and the main branch have differences that the version control system doesn't know how to handle.

Once you have pulled from the main branch, remember to switch to your own branch! If you are running `git status` frequently, you'll notice whether or not you've forgotten to make the switch.

## **Tip #3: Add All Your Changed Files at the Same Time**

If you have changed a number of files, you'll find it tedious to type `git add <insert long filename here>` over and over again. A quick way to add all your changed files at the same time is to use `git add -A`. Just be sure when using this that you really want to add all your files.

#### **Tip #4: Give Your Commits a Useful Name**

When you commit your files, adding a commit message is optional, but most companies expect their employees to do it. Make it easier on yourself and everyone else by naming your commits something that will make sense later. “One-line code change” is not a very helpful commit message. “Adding test for new contact info endpoint” provides much more detail.

#### **Tip #5: View Your Git Logs in a Single Line**

It can be hard to remember what you've done in Git because there's no UI to show you. This is where `git log` is helpful. The log in its full version will show you the last several commits made, who made them, and the date and time they were made. I find it's easier to read the logs when they are condensed to a single line; to do this, type `git log --pretty=oneline`. To exit the log, type `q`.

#### **Tip #6: View the “Diff” of Your Files Before You Do a Pull Request**

If you are running `git status` frequently, you probably won't commit and push any files that you didn't mean to push. But it's still possible to accidentally commit code you didn't mean to commit in a file that has the changes you want. So, before you do a pull request and ask someone to review your code, view the “diff” of your files, which simply means looking in GitHub and comparing the files in your branch with the files in the master branch to see which lines were changed. Make sure there are no code changes in the file that you didn't want to commit, such as commented-out code or debugging statements.

If you find that you've accidentally pushed something you didn't mean to push, simply change the file to what you want it to be and add, commit, and push it again.

# Chapter 73: Merge Conflict Resolution

Anyone working with version control software such as Git will eventually come across a merge conflict. If you are new to working with Git, here is a simple example.

The main branch contains a file with this text:

**Kristin Jackvony was here on May 22, 2021**

Prunella and Joe each check out a version of this main branch. Prunella makes a branch called “Prunella” and Joe makes a branch called “Joe”.

Joe updates the file in his branch to read:

**Kristin Jackvony was here on May 22, 2021**

**Joe Schmoe was here on May 23, 2021**

Joe does a pull request for his file changes, and they are approved and merged into the main branch.

Shortly thereafter, Prunella updates the file in her branch to read:

**Kristin Jackvony was here on May 22, 2021**

**Prunella Prunewhip was here on May 23, 2021**

She also does a pull request for her file changes, but because she no longer has the latest version of the main branch, there is a merge conflict. Git sees that she wants to add her name to the second line of the file, but her version of the main branch doesn’t have anything on the second line, whereas the latest version of the main branch has Joe’s name on it. Prunella will need to resolve the conflict before her changes can be merged.

By using the tips in the preceding chapter, especially tips #1, #2, and #6, you can avoid creating a merge conflict. It’s always a good idea to do a pull from main before you do anything code related.

However, if you do create a merge conflict, here are six tactics to keep in mind.

### **Tactic #1: Don't Panic**

When you have a merge conflict, it's important not to thrash around trying "git this" and "git that". You might make things worse this way. A merge conflict will eventually be resolved, even if you have to resort to asking for help (see tactic #6). And you can't possibly have done anything irreversible; that's the beauty of version control!

### **Tactic #2: Resolve the Conflict from Within GitHub**

GitHub has an easy interface that allows you to resolve merge conflicts. Simply click the Resolve Conflicts button to see the conflict. Our example conflict would look something like this:

```
>>>>HEAD  
Joe Schmoe was here on May 23, 2021  
=====  
Prunella Prunewhip was here on May 23, 2021  
<<<< Prunella
```

In our example, all we need to do is decide which entry we want on line 2 and which entry we want on line 3 and make those edits, deleting the extraneous symbols and branch names along the way. When we are done, our file will look like this:

```
Kristin Jackvony was here on May 22, 2021  
Joe Schmoe was here on May 23, 2021  
Prunella Prunewhip was here on May 23, 2021
```

Now we just click the Mark as Resolved button and the pull request should be ready for merging.

### **Tactic #3: Resolve the Conflict from the Command Line**

If you are using Git as a version control system but you are not using

GitHub to host your repositories, you may not have a nice UI to work with in order to resolve your conflict. In this case, you may need to use the command line.

What I do in this scenario is open the file with the merge conflict in a text editor such asTextEdit or Notepad++, remove all the extraneous symbols and branch names, and then do a git add with the filename, and then a git commit.

### **Tactic #4: Forget About Your Existing Branch and Make a New One**

In this scenario, I copy the text of the entire file, including all my changes, and paste it somewhere outside the code. Then I go to the master branch and do a `git pull origin main`. Now I should have all the latest changes. Next, I create a brand-new branch, switch to that branch, and paste in all my file changes. Then I do a new pull request, which won't have the merge conflict.

### **Tactic #5: The Nuclear Option**

If tactic #4 fails to work, I exercise the nuclear option, which is to delete the entire repository from my machine. Before I do this, however, I make a copy of my file with all my changes and paste it somewhere outside the code, as I did in tactic #4. Then I delete the entire repository and clone it again. At that point, I create a new branch, switch to it, make my changes, and do a brand-new pull request.

### **Tactic #6: Ask for Help**

If all else fails, ask someone for help. There's no shame in asking for help with a particularly thorny merge conflict! If you followed tactic #1 and didn't panic, your helper will be able to see exactly what you've done so far and can help you arrive at a solution.

Git purists may argue that merge conflicts should be resolved the right way (using tactic #2 or #3), and they are probably right. But using tactic #4 or #5 can keep you from wasting time trying to resolve the conflict, and these tactics will also keep you from wanting to throw your laptop out the window!

# Chapter 74: A Gentle Introduction to Regex

In my experience, working with regex makes everyone's head hurt. No one wants to have to look at `^(19|20)\d\d[- /.](0[1-9]|1[012])[- /.](0[1-9]|[12][0-9]|3[01])$` and figure out what it means!

However, regex is a very powerful tool and it's good to know how to use it, even if (like most people) you're not an expert. This chapter serves as a gentle introduction to regex so that when you encounter it in your testing you'll feel more comfortable with it.

The first thing you should know about regex is that it stands for “regular expression,” which is simply a sequence of characters that define a search pattern. Regexes are very useful for doing things like editing a string or checking to see whether a phone number, date, or postal code fits the accepted pattern.

The second thing you should know about regex is that it's a lot easier to use when you have a regex tester available! Many free regex testers are available on the Web.

Regex varies slightly depending on what language you are using, but the basic building blocks are the same in each variation. Here are 10 regex symbols that will help you get started:

- `^` : This indicates that you want to match the beginning of a word. For example, if you were using a pattern that started with `^ball`, you could match the word `ball` or the word `balloon`, but you could not match the word `football`, because it doesn't begin with `ball`.
- `$` : This indicates that you want to match the end of a word. So, if you were using a pattern that ended with `ball$`, you could now match the word `football`, but you couldn't match the word `balloon`, because it doesn't end with `ball`.
- `.` : This matches any character. You can use this when one of the

characters in a string is going to vary. So, if you had the pattern `foo.ball`, you could match `football` or `foosball`.

- `*`: This indicates that the character should be matched one or more times. In the pattern `fo*tball`, the letter `o` can be matched one or more times. So with this pattern, you could match `fotball`, or `football`, or even `oooooooootball`.
- `\d`: This matches any numeric digit. The pattern `football\d` will match `football1`, `football2`, `football3`, and so on, but not `football` or `football!`.
- `\w`: This matches any character from the basic Latin alphabet. So, if you were looking for a pattern of `12345\w`, this symbol would match `12345a`, `12345b`, `12345z`, and so on, but not `123456` or `12345!`.
- `\s`: This matches a space. If you had a pattern of `foot\sball`, it would match `foot ball` but not `football`.
- `[ ]`: These indicate a character set. So, if you wanted to match any number from 1 to 5, you could use `[12345]`. A pattern of `football[12345]` would match `football1` but not `football6`.
- `|`: This is an either/or pattern. A regex pattern of `cat|dog` will match `cat`, and will also match `dog`.
- `( )`: These group pattern items together, the same way they do in mathematical expressions. Let's say you were trying to find a match for `November` or `December` but not for `September`. You couldn't just use a regex pattern of `ember`, because that would match all three months. But you could use `(Nov|Dec)ember`; using the parentheses combined with the pipe character shows that the month could either have `Nov` or `Dec`, and then should continue with `ember`.

I kept these examples very simple because there is so much to regex that you could spend months learning it, and it is very easy to get confused! But these commonly used symbols should be enough to get you feeling a bit more comfortable with it. Take some time to play around with a regex tester to

practice what you've learned, and if you'd like to learn more, try an interactive tutorial on the Web.

# Chapter 75: Logging, Monitoring, and Alerting

A bug-free application doesn't mean a thing if your users can't get to it because the server crashed! For this reason, it's important to understand logging, monitoring, and alerting so that you can participate in ensuring the health of your applications.

## Logging

Logging is simply recording information about what happens in an application. The information is written to either a file or a database. Often, developers will include logging statements in their code to help them determine what's going on with the application behind the UI. This is especially helpful in applications that make calls to a number of servers or databases.

Recently I tested a notification system that passed a message from a function to a number of different channels. The logging statements were so helpful in this case because they enabled me to follow the message through the channels. Without them, I wouldn't have been able to figure out where the bug was when I didn't get the message I was expecting.

Good log messages should be easy to access and search. You shouldn't have to log on to some obscure remote desktop and sift through tens of thousands of entries with no line breaks. There are many helpful logging applications that let you search and sort through logs in an easy-to-read format.

Good log messages should also be easy to understand, and provide helpful information. It's so frustrating when a log message about an error reads "An unknown error occurred" or "Error TSGB-45667". Ask your developer to provide log messages that clearly state what went wrong and where it happened in the code.

Another helpful tactic for logging is to give each event a specific GUID as an identifier. The GUID will stay associated with everything that happens with the event so that you can follow it as it moves from one area of an application to another.

## **Monitoring**

Monitoring means setting up automatic processes to watch the health of your application and the servers that run it. Good monitoring ensures that any potential problems can be discovered and dealt with before they reach the end user. For example, if it becomes clear that a server's disk space is reaching maximum capacity, additional servers can be added to handle the load.

Things to monitor include:

- Server response times
- Load on the server
- Server errors, such as 500-level response errors
- CPU usage
- Memory usage
- Disk space

One way to monitor application health is with a periodic health check or a ping. In this case, a job is set up to make a request to the server every few minutes and record whether the response was positive or negative. Monitoring can also happen through a tool that watches the number of requests made to the server and records whether those requests were successful. Data points such as response times and CPU usage can also be recorded and examined to see whether there are any trends that might indicate the application is unhealthy.

## **Alerting**

All the logging and monitoring in the world won't be helpful if no one pays attention to the logs and monitors! This is where alerting comes in. Alerts can be set to notify the appropriate people so that immediate action can be taken when there is a problem.

Following are some situations that might call for an alert:

- CPU or memory usage goes above a certain threshold.
- Disk space goes below a certain threshold.
- The number of 500 errors goes above a certain level.
- A health check fails twice in a row.
- Response times are slower than expected.
- Load is higher than normal.

There are a number of ways to alert people of problems. For example, you can set up alerts that will send emails, text messages, or phone calls. Note that you should only set off-hours alerts for serious cases in which users might be affected. No one wants to be awakened in the middle of the night by an alert that says the QA servers are down! However, a problem in the QA environment could indicate an issue that might be seen in the production environment in the future. So a less invasive alert, such as a message to a team chat room, could be set up for this situation.

You may be saying to yourself at this point: “But I’m a software tester! It’s not my job to set up logging, monitoring, and alerting for the company.” The health of your application is the responsibility of everyone who works on it, including you! While you might not have the clout to purchase server monitoring software, you still have the power to ask questions of your team, such as:

- How can we troubleshoot user issues?
- How do we know we have enough servers to handle our application’s load?
- How do we know our API is responding correctly?
- How will we know whether a DDoS attack is being attempted on our application?
- How will we know whether our end users are experiencing long wait times?
- How will we know whether we are running out of disk space?

Hopefully these questions will motivate you and your team to set up logging, monitoring, and alerting that will ensure the health and reliability of your application.

## Part X: Automated Testing

# Chapter 76: Why Automate?

For some of you, your initial answer to the question “Why automate?” may be “Because my manager wants me to.” This is, of course, a valid reason. But after you have a few automated tests written, you will discover other, more important reasons.

## **It Saves You Time**

At first it will seem as though automation is a huge time-suck because it will take several weeks to get your automation system up and running. But once it’s in place, it will save you valuable testing time. Even without looking at your application, you will know whether something went wrong with the latest build.

## **It Freed You from Some Repetitive Tasks**

A feature I once tested had a search form with four editable fields: Last Name, Company, Email, and Phone Number. To test this feature well, I would need to test with every possible permutation: just one field at a time, two fields at a time, three fields at a time, and four fields at a time. This adds up to 15 searches. It was rather tedious to do it every time I tested the search function. But my automated test was happy to run through these 15 searches as many times as I told it to.

## **It Helps You Think More Critically About Your Testing**

When I’m manually testing I tend to follow a written plan, and add a few exploratory tests as well. But when I’m writing an automated test I need to think about why I’m running the test. The test engine will not be exploring, so I’m going to need to tell it to do something very specific. I ask myself: What are the minimum steps required to ensure that the feature works? Will I need to vary those steps for broader coverage, and what is the most efficient way to do that?

## **It Helps You Understand Your Development Team**

Before I started writing automated tests, I knew a bit about coding from my college courses but I'd never had hands-on experience. Writing usable code helped me have more sympathy for what developers go through in order to create software; it also taught me how programs are structured and how the build process works. This has helped me become a better tester.

The path to learning automation won't be easy, but it will teach you important skills, streamline your testing process, help you understand the development lifecycle, and improve your thinking about your tests.

# Chapter 77: When to Automate

The first step in automation success is knowing when to automate and when not to. Here are some guidelines.

## **DO Automate Repetitive Tasks**

I once needed to add over 600 records to an application in order to fully test a feature. Rather than add just a few records to partially test it, I created a simple script that would add as many records as I specified. Another great thing to automate is a feature with a number of different permutations.

## **DO Automate Basic Smoke-Level Tests**

I like to think of smoke-level tests as tests of features whose failure in the field would really embarrass us. One company I worked for had a search feature broken for weeks, and no one noticed because we hadn't run a test on it. Unfortunately, the bug was pushed out to production and seen by customers. Automating these tests and running them at every build or early in the morning can help catch major problems quickly.

## **DON'T Automate Fragile Features or Features Going Through Churn**

There may be a feature of your company's software that seems to break every time someone looks at it in a funny way. Automating tests of this feature practically guarantees that the tests will fail daily. Similarly, a feature that is going through a lot of revisions (perhaps because the team is not yet sure what the customer wants) will have many changes that will cause you to make frequent code changes in your tests. It's better to keep testing the feature manually and reporting bugs on it until it becomes more stable.

## **DO Automate Things Users Will Do Every Day**

What is the primary function of your software? What is the typical path that a user will take when using your software? These are the kinds of things

that should be automated. Rather than running through this manual path every day at 9 AM, you can set your automated test to do it at 8 AM and you'll know right away if there is a problem.

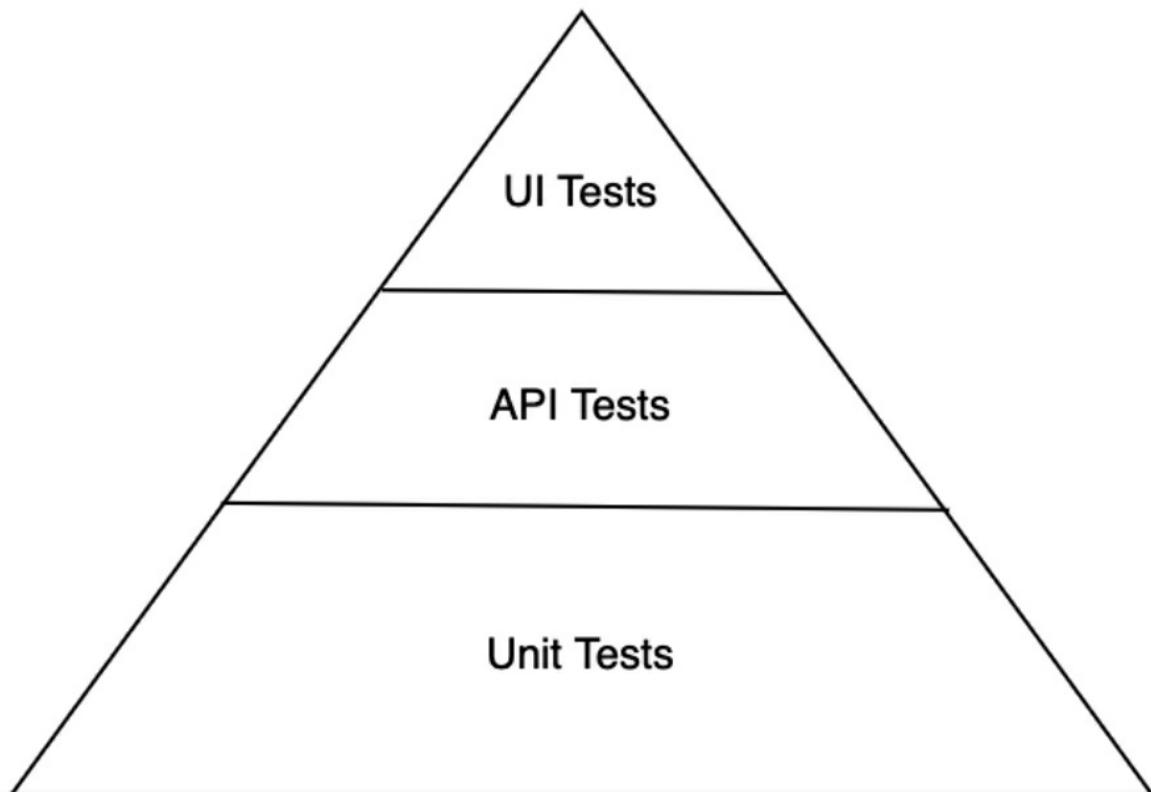
### **DON'T Automate Weird Edge Cases**

There will always be bugs in software, but some will more likely be seen by users than others. You may be fascinated by the bug that is caused by going to a specific sequence of pages, entering non-UTF-8 characters, and then clicking the back button three times in a row, but since it's very unlikely that an end user will do this, it's not worth your time to design an automated test for it.

Knowing when to automate will ensure that you are spending time writing tests that will save you time rather than writing tests that will need constant attention.

# Chapter 78: Rethinking the Pyramid: The Automation Test Wheel

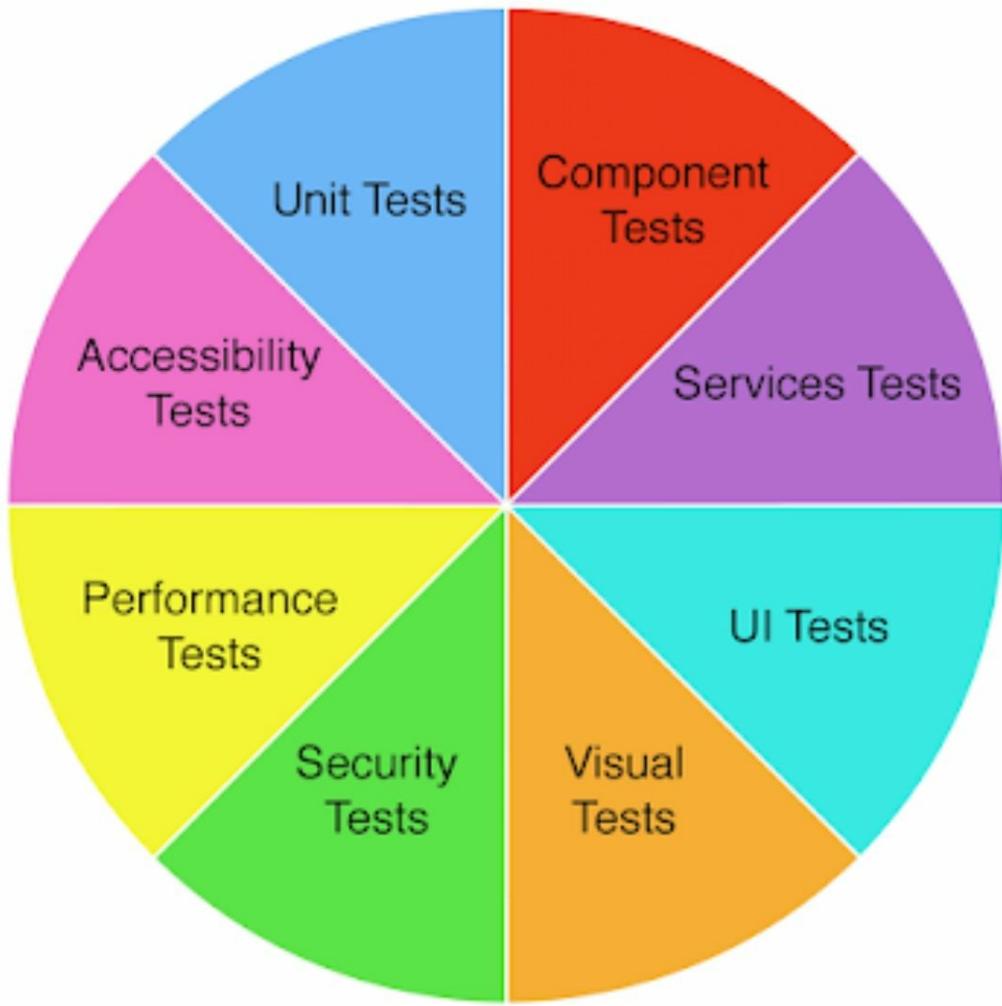
Anyone who has spent time working on test automation has likely heard of the test automation pyramid. The pyramid typically comprises three horizontal sections: UI Tests, API Tests, and Unit Tests.



The bottom section is the widest and is for the unit tests. There should be more unit tests than API or UI tests because unit tests run very quickly and are extremely reliable since they don't rely on any data store. The middle section is for the API tests. There should be fewer API tests than unit tests because API tests are slower to run and rely on integration to a data store or

other resource. The top section is for the UI tests. The fewest number of tests run should be UI tests because they take the most time and are the most fragile.

While the pyramid is very helpful for teaching teams that they should test as close to the code as possible, it doesn't include many types of automated tests and it assumes that the number of tests is the best indicator of appropriate test coverage. I propose a new way of thinking about automated testing: the automation test wheel.



Each test type in the automation test wheel can be thought of as a spoke in a wheel; all are necessary and equally important. The size of each section of the wheel does not indicate the number of the tests to be automated; each test type should have the number of tests required to verify quality in that area. Let's take a look at each test type.

**Unit tests:** A unit test is the smallest automated test possible. It tests the behavior of just one function or method. For example, if I had a method that tested whether a number was 0, I could write these unit tests:

- A test that passes a 0 to the method and validates that it is identified as a 0
- A test that passes a 1 to the method and validates that it is identified as nonzero

- A test that passes a string to the method and validates that the appropriate exception is thrown

Because unit tests are independent of all other services and run so quickly, they are a very effective way of testing code. They are often written by the developer who wrote the method or function, but they can also be written by others. Each method or function should have at least one unit test associated with it.

**Component tests:** These tests check the various services the code depends on. For example, if we had code that called the GitHub API, we could write a component test that would make a call to the API and verify that we got a response. Other examples of component tests include pinging a server or making a call to a database and verifying that a response was received. There should be at least one component test for each service the code relies on.

**Services tests:** These tests check the web services that are used in the code. In today's applications, web services often use API requests, so these are usually API tests. For example, if we have an API that accepts POST, GET, PUT, and DELETE requests, we will want to have automated tests that check each request type. We will want to have both happy path tests that check that a valid request returns an appropriate response, and negative tests that verify that an invalid request returns an appropriate error code.

**User interface (UI) tests:** UI tests verify that end-user activities work correctly. These are the tests that will fill out text fields and click on buttons. As a general rule, anything that can be tested with a unit, component, or service test should be tested by those methods instead. UI tests should focus solely on the user interface.

**Visual tests:** Visual tests verify that elements are actually appearing on the screen. This is slightly different from UI tests, because the UI tests are focusing on the functionality of the user interface rather than the appearance. Examples of visual tests include verifying that a button's label is rendered correctly and verifying that the correct product image is appearing on the screen.

**Security tests:** These tests verify that security rules are being respected. Security tests can overlap with services tests, but they should still be considered separately. For example, a security test could check to make sure an authorization token cannot be generated with an invalid username and password combination. Another security test could make a GET request with an authorization token for a user who should not have access to that resource, and verify that a 403 response is returned.

**Performance tests:** Automated performance tests can verify that response times for a request happen within an appropriate period. For example, if your company has decided that GET requests should never take longer than two seconds, tests can be created to return a failure state if the request takes longer than that amount of time. Web page load times can also be measured with performance tests.

**Accessibility tests:** Automated accessibility tests can check a variety of things. When combined with UI tests, they can verify that images have text descriptions for those with low vision. Visual tests can also be used to verify that the text on the screen is the correct size.

You may have noticed that the preceding descriptions often overlap. For example, security tests might be run through API testing and visual tests might be run through UI testing. What is important here is that each area is tested thoroughly, efficiently, and accurately. If a spoke is missing from the wheel, you will never be comfortable relying on your automation when you are doing continuous deployment.

In the next chapter, I'll discuss how we can fit these tests into a real-world application testing scenario.

# Chapter 79: The Automation Test Wheel in Practice

Using the Contact List app as an example, here's how I would create tests for every section of the automation test wheel.

**Unit tests:** I will make sure every function of my code has at least two unit tests. Remember that unit tests don't connect to any dependency, such as a server, so I'll run these tests using mock responses. For each function in my app, I will create a positive and negative mock response. Then I'll make both a valid call to the function and an invalid call, and I'll verify that I get the appropriate mock response.

**Component tests:** My application is very simple and relies on just one database. The database is used for authentication and for retrieving the contact data. I will include one test for each function; I'll send an authentication request for a valid user and verify that the user is authenticated, and I'll make one request to the database to retrieve a known contact and verify that the contact is retrieved.

**Services tests:** My application has an API that allows me to do CRUD (Create, Read, Update, Delete) operations on my contacts. I have two GET endpoints: one allows me to retrieve the list of contacts and the other allows me to retrieve one specific contact. I also have a POST endpoint that allows me to add a contact to the contact list; PUT and PATCH endpoints that allow me to update the data for an existing contact; and a DELETE endpoint that allows me to delete an existing contact.

My API also has endpoints that allow me to create a user with a POST, GET a user, update a user with a PATCH, DELETE a user, and do a POST to log in and log out as the user.

For each of these endpoints, I will have a series of tests. The tests will include both happy paths and error paths. I'll verify that for each request the response code and the response body are correct. For example, with the GET

endpoint where I retrieve one contact, I'll verify that a GET on an existing contact returns a 200 response and the correct data for the contact. I'll also verify that a GET on a contact that doesn't exist returns a 404 Not Found response.

**User interface (UI) tests:** This is where I will be testing in the browser, validating that all the buttons and fields work as they are supposed to. I will also be validating that error messages appear on the page appropriately. For example, on the Add Contact page, I'll validate that I can add text to all the form fields and click on the Submit button. I'll also validate that the Cancel button works correctly, and that when I have a field that fails validation rules, an error message is returned on the page.

**Visual tests:** This is where I will verify that elements are actually appearing on the page the way I want them to. I will navigate to the list page and verify that all the columns are appearing on the page. I will navigate to the Add Contact page and verify that all the form fields and their labels are appearing appropriately on the page. I will trigger all possible error messages (such as the one I would receive if I entered an invalid phone number) and verify that the error appears correctly on the screen. And I will verify that all the buttons needed to use the application are rendering correctly.

**Security tests:** I will run security tests at both the Services layer and the UI layer. I will test the API operations relating to authenticating a user, verifying that only a user with the correct credentials will be authenticated. I will test every request endpoint to make sure only those requests with a valid token are executing; requests without a valid token should return a 401. I'll also use API tests to verify that one user cannot view another user's contacts. For the UI layer, I will conduct a series of login tests that validate that only a user with correct credentials is logged in.

**Performance tests:** I will set benchmarks for both the server response time and the web page load time. To measure the server response time, I will add assertions to my existing services tests that will verify that the response was returned within that benchmark. To measure the web page load time, I will run a UI test that will load each page and assert that the page was loaded within the benchmark time.

**Accessibility tests:** I want to make sure my application can be used by those with low vision. So I will run a set of UI and visual tests on each page where I validate that I can increase the text size and that scroll bars appear and disappear depending on whether they are needed. For example, if I zoom in on the contact list I will now need a vertical scroll bar, because some of the contacts will now be off the page.

With this series of automated tests, I will feel confident that I'll be able to deploy changes to my application and discover any problems quickly.

# Chapter 80: Unit Tests

Unit tests are usually written by developers, but it's a good idea for all software testers to understand what they are and how they work. Unit tests test just one method or function, and they aim to exercise as many paths of that method or function as possible.

The major benefit of unit tests is that they provide extremely fast, accurate feedback.

In this chapter I'll use Node.js and Jest to demonstrate how unit tests work. You can find this code at my GitHub repository: <https://github.com/kristinjackvony/unitTestExample>.

The function I am testing is called **isDozen**, and you can find it in the *dozen.js* file. Given a number, the function first checks to see whether it is 12. If it is, it returns a “Yup, it’s a dozen!” message. If it is not 12, it checks to see whether it is more or less than 12, and returns appropriate messages for those cases. Then it checks to see whether the number is 0 or negative, and returns an appropriate message if it is. Finally, if the number doesn’t meet any of these criteria, it returns a “This is not a number” message.

If I were testing this function manually, assuming it had a UI interface, I’d check to make sure it recognized 12 as one dozen and that it recognized when a number was more than or less than one dozen. I’d also try some of the usual testing tricks, like passing in a negative number or “FOO”.

This is exactly what we’ll do with our unit tests! I am using Jest for my unit testing, and you can find the code in my *dozen.test.js* file. I have five test cases here, aptly named as follows:

- Recognizes one dozen
- Recognizes more than one dozen
- Recognizes less than one dozen
- Recognizes a negative number
- Recognizes a non-number

In each test, I call the `isDozen` function with the number I want to test with. Then I assert that the result I got matches the result I was expecting.

If you'd like to run these tests yourself, make sure you have Node.js installed, navigate to the location of the `unitTestExample` folder, run `npm install` to install Jest, and then run the tests with the `npm run test` command.

Once you have the tests running, try making the first test fail by changing the assertion statement to this: `expect(isDozen(12)).toBe("This is not the real response")`. Run the tests again and watch the assertion fail.

Then try changing a test so that you are passing in a different value. For example, you could pass the number 15 into the “Recognizes more than a dozen” test instead of 13.

Once you are familiar with how these unit tests work, try taking another simple code example and writing unit tests for it. Remember that unit tests do not call a data store; they are only testing the function itself.

# Chapter 81: Component Tests

A component test is a test for a service that an application is dependent on. For example, an application might need to make calls to a database, so a component test would make a simple call to that database and verify that it received data in response. Another example of a component is an API that the application doesn't own. In this scenario, a component test would make a simple call to the API and verify that it got a 200-level response.

Let's imagine I have a simple app that is dependent on two things: a Mongo database that has a list of users, and the PokéAPI, which is an API that catalogs information about every type of Pokéémon character. For my component tests, I want to test those two dependencies: that I can make a request to the database and get a positive response, and that I can make a call to the API and get a positive response.

To test the dependencies, I've decided to use Jest and Supertest. Supertest is a library that extends Jest testing by making it easier to call APIs.

Here is what my tests will look like:

```
const request = require('supertest');

describe('Database Connection Test', () => {
  it('Returns a 200 with a call to the DB', async () => {
    const res = await request('http://localhost:3000')
      .get('/userlist')
      .expect(200)
  });
  });

describe('PokeAPI Connection Test', () => {
  it('Returns a 201 with a health check', async () => {
    const res = await request('https://pokeapi.co/api/v2')
      .get('/')
      .expect(200)
```

```
})  
})
```

The first line of my file is invoking Supertest, so I will be able to do HTTP requests. Let's take a look at each part of the second test so that we can see what it's doing.

The “describe” section comprises the entire test:

```
describe('PokeAPI Connection Test', () => {  
  it('Returns a 200 with a call to the API', async () => {  
    const res = await request('https://pokeapi.co/api/v2')  
    .get('/')  
    .expect(200)  
  })  
})
```

‘PokeAPI Connection Test’ is the title of the test:

```
describe('PokeAPI Connection Test', () => {  
  it('Returns a 200 with a call to the API', async () => {  
    const res = await request('https://pokeapi.co/api/v2')  
    .get('/')  
    .expect(200)  
  })  
})
```

The “it” section is where the assertion is called:

```
describe('PokeAPI Connection Test', () => {  
  it('Returns a 200 with a call to the AP', async () => {  
    const res = await request('https://pokeapi.co/api/v2')  
    .get('/')  
    .expect(200)  
  })  
})
```

‘Returns a 200 with a call to the API’ is the title of the assertion:

```
describe('PokeAPI Connection Test', () => {
  it('Returns a 200 with a call to the API', async () => {
    const res = await request('https://pokeapi.co/api/v2')
      .get('/')
      .expect(200)
  })
})
```

Here is where we are making the call to the API:

```
describe('PokeAPI Connection Test', () => {
  it('Returns a 200 with a call to the API', async () => {
    const res = await request('https://pokeapi.co/api/v2')
      .get('/')
      .expect(200)
  })
})
```

And here is where we are expecting that we will get a 200 response:

```
describe('PokeAPI Connection Test', () => {
  it('Returns a 200 with a call to the API', async () => {
    const res = await request('https://pokeapi.co/api/v2')
      .get('/')
      .expect(200)
  })
})
```

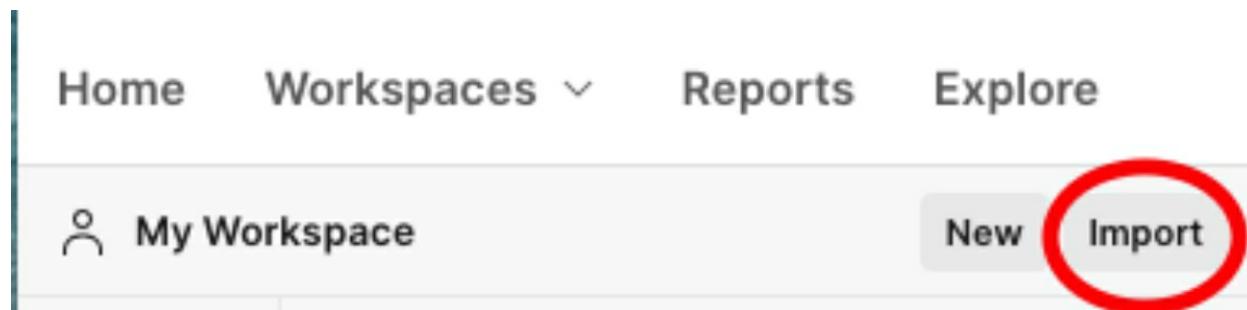
How many tests you have in this area will depend on how many external systems your application is dependent on. It may be a good idea to create a “health check” that will run your component tests whenever your code is deployed. This way, you will be alerted if there is an error when calling your external systems.

# Chapter 82: Services Tests

There are many types of services, but the most widely used service is the REST API. And my favorite way to test a REST API is with Postman.

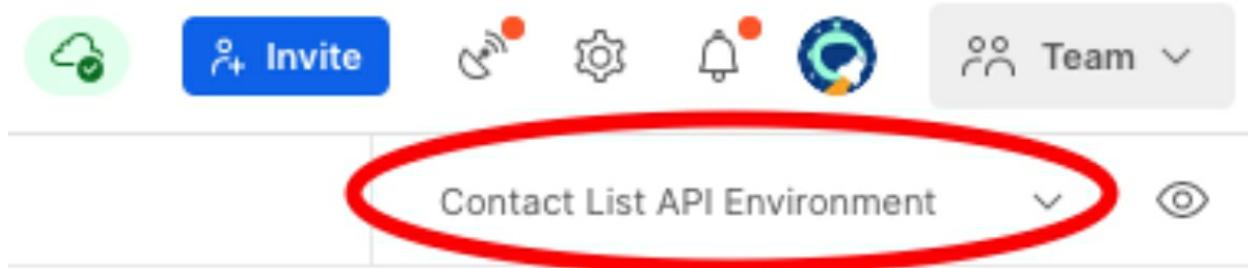
To demonstrate automating services tests, I've created a collection of Postman tests that you can download from GitHub here: <https://github.com/kristinjackvony/apiTestsExample>

Once you have downloaded the file from GitHub, you can upload it in Postman to see what it does. To upload the collection, click on the Import button on the upper-left of the Postman window:



When the Import window pops up, drag the downloaded file into the window and it will be uploaded to Postman. When you choose the Collections tab on the left you should see a list of your collections, and Contact List API Smoke Tests should be one of the collections.

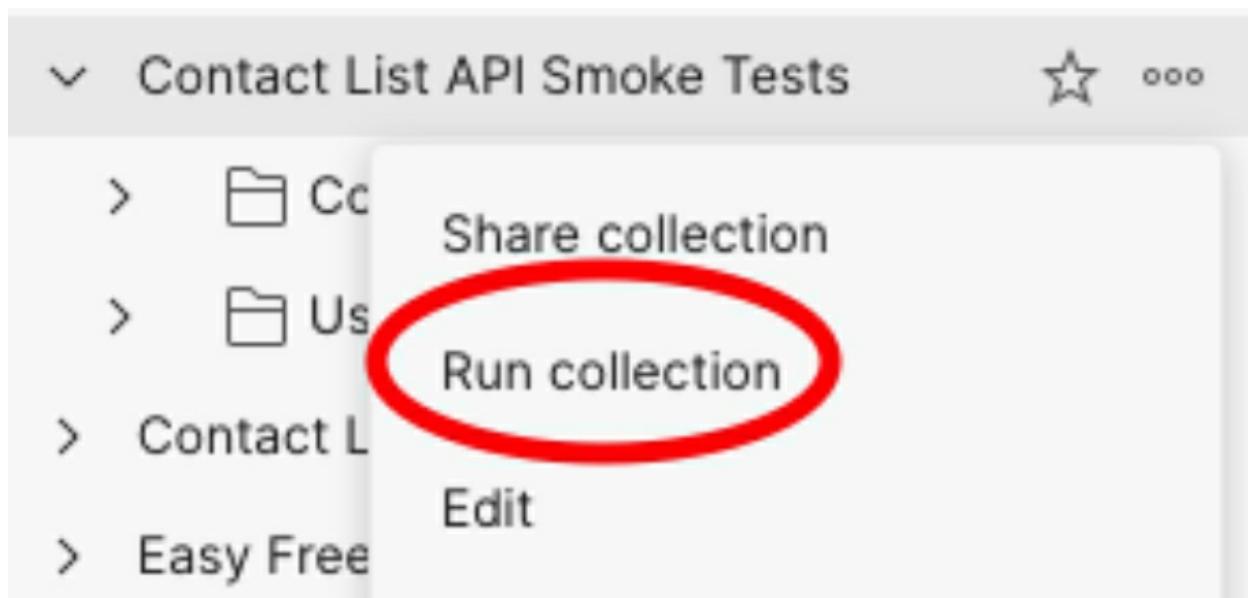
Next, create an environment called “Contact List API Environment”. If you don’t remember how to create an environment, see the instructions in Chapter 40. You may need to create one empty variable in order to save the environment. Once your environment is created, make sure you have selected the environment in the upper-right corner:



You don't have to put any other variables in this environment; scripts within the collection will be adding variables for you.

You are now ready to run the requests from within Postman. Keep in mind that the tests are not idempotent: some of the requests depend on previous requests for setting variables and creating users and contacts, so you should run the requests in order.

To run all the tests in the collection, choose the Collections tab, click on the Contact List API Smoke Tests collection, and click on the three-dot menu to the right of the collection name. From the dropdown that appears, choose "Run collection":



A new Runner tab will appear in the main window. Click on the Run Contact List API Smoke Tests button:

Contact List API S... Runner + ... Contact List API Environment

RUN ORDER Deselect All | Select All | Reset

- ✓  > POST Setup: Add User
- ✓  > POST Add Contact
- ✓  > GET Get Contact List
- ✓  > GET Get Contact
- ✓  > PUT Update Contact
- ✓  > PATCH Update Contact
- ✓  > DEL Delete Contact
- ✓  > GET Verify Contact Deleted
- ✓  > DEL Cleanup: Delete User
- ✓  > POST Add User
- ✓  > GET Get User Profile
- ✓  > PATCH Update User
- ✓  > POST Log Out User
- ✓  > POST Log In User
- ✓  > DEL Delete User
- ✓  > GET Verify User Deleted

Iterations: 1

Delay: 0 ms

Data: Select File

Save responses

Keep variable values

Run collection without using stored cookies

Save cookies after collection run

Run Contact List API Smoke Tests

When you click the button, you should see all the requests run and all the assertions pass:

The screenshot shows the Postman interface with the 'Contact List API Environment' selected. At the top, there are tabs for 'Contact List API S...' and 'Contact List API S...'. Below the tabs, the title 'Contact List API Smoke Tests' is followed by 'Contact List API Environment, just now'. On the right, there are buttons for 'View Summary', 'Run Again', 'New', and 'Export Results'. A navigation bar at the bottom includes 'All Tests' (underlined), 'Passed (62)', 'Failed (0)', and a search bar with placeholder 'ooo'. The main area displays a list of test cases:

- POST Log Out User** https://thinking-tester-contact-list.herokuapp.com/users/logout / Users / Log Out User 200 OK 52 ms 154 B
  - Pass First name is updated
  - Pass Last name is updated
  - Pass Email is updated
- POST Log In User** https://thinking-tester-contact-list.herokuapp.com/users/login / Users / Log In User 200 OK 107 ms 757 B
  - Pass Status code is 200
  - Pass UserId is correct
  - Pass First name is correct
  - Pass Last name is correct
  - Pass Email is correct
- DELETE Delete User** https://thinking-tester-contact-list.herokuapp.com/users/me / Users / Delete User 200 OK 51 ms 154 B
  - Pass Status code is 200
- GET Verify User Deleted** https://thinking-tester-contact-list.herokuapp.com/users/me / Users / Verify User Deleted 401 Unauthorized 51 ms 286 B
  - Pass Status code is 401

Take a moment to explore the different requests and assertions in the collection. Some of the requests have a “pre-request script,” which creates some variables and sets them to random Postman-generated values. When you look at the Contact List API Environment that you created, you will see that the created variables have been added to the environment.

Now that you understand how the tests and assertions work, let’s run them from the command line! To do this you will need to have Node.js installed. Installing node will also install npm, which is the Node Package Manager. Then you can use npm to install Newman, the package that is used to run Postman tests. To install Newman, simply open your command-line window and type:

```
npm install -g newman
```

Once Newman is installed, locate the place where you downloaded the *ContactListAPISmokeTests.postman\_collection.json* file. You will want to download your environment file to this same location. To do this, go to

Postman, click on the Environments tab in the left nav menu, click on Contact List API Environment, and in the upper-right corner, click on the three-dot menu and choose Export:



F Fork

0

Save

Share

...

(i)

Set as active environment

II



Export

Duplicate

⌘D

Create a fork

Create Pull Request

Merge changes

Manage Roles

..

Remove from workspace

Delete



&gt;;J9.eyJfaWQiOiI.

When the Export window opens, choose the location where your collection is currently saved, and save your environment as:

ContactListAPIEnvironment.postman\_environment.json

From your command-line window, navigate to the folder where you saved your collection and environment. Now you can run your collection with this command:

```
newman run APISmokeTests.postman_collection.json -e  
ContactListAPIEnvironment.postman_environment.json
```

(This command should be in a line with no returns; it's OK if it wraps around to a second line.) The **-e** in this command stands for “environment”.

If all goes well, you should get a result that looks like this:

|                    | executed | failed |
|--------------------|----------|--------|
| iterations         | 1        | 0      |
| requests           | 16       | 0      |
| test-scripts       | 16       | 0      |
| prerequest-scripts | 4        | 0      |
| assertions         | 62       | 0      |

total run duration: 1726ms

total data received: 2.73KB (approx)

average response time: 82ms [min: 41ms, max: 279ms, s.d.: 55ms]

You can also run your tests from other file locations. To do this, simply specify the path to your files in your command. For example, if my files were in a folder called NightlyTests which was in my Documents folder, and if I were using a Windows computer, I would use this command:

```
newman run
C:/users/kjackvony/Documents/NightlyTests/APISmokeTests.postman_collection
-e
C:/users/kjackvony/Documents/NightlyTests/ContactListAPIEnvironment.postman_collec
```

Now that you know how to run your tests from the command line, you can set up your tests to run automatically with a cron job or in a CI/CD platform such as Jenkins. Just be aware that the machine you run your tests on will need to have Newman installed.

You can also have your test results write to a file with the -r command. For example, you can have your results written in JUnit with this command:

```
newman run ContactListAPISmokeTests.postman_collection.json -e ContactListAPIEnvironment.postman_environment.json -r junit
```

When the tests have finished running, a Newman folder will be created with your test results inside.

# Chapter 83: What API Tests to Run and When to Run Them

Knowing how to automate API tests isn't that helpful unless you make good choices about what tests to run and when to run them. Let's first think about what to automate.

Let's imagine that we have an API with these requests:

```
POST user  
GET user/{userId}  
PUT user/{userId}  
DELETE user/{userId}
```

The first category of tests we will want to have are the simple happy path requests. For example:

- POST a new user and verify that we get a 200 response.
- GET the user and verify that we get a 200 response and that the correct user is returned.
- PUT an update to the user and verify that we get a 200 response.
- DELETE the user and verify that we get a 200 response.

The next category of tests we want to have are some simple negative requests. For example:

- POST a new user with a missing required field and verify that we get a 400 response.
- GET a user with an ID that doesn't exist and verify that we get a 404 response.
- PUT an update to the user with an invalid field and verify that we get a 400 response.
- DELETE a user with an ID that doesn't exist and verify that we get a 404 response.

You'll want to test 400 and 404 responses on every request that has them.

The third category of tests we want to have are more happy path requests, but with variations. For example:

- POST a new user with only the required fields rather than all fields and verify that we get a 200 response.
- GET a user with query parameters, such as user/{userId}?fields=firstName,lastName, and verify that we get a 200 response and the appropriate values in the response.
- PUT a user where one nonrequired field is replaced with null and one field that is currently null is replaced with a value and verify that we get a 200 response.

It's worth noting that we might not want to test every possible combination in this category. For example, if our GET request allows us to filter by five different values— firstName, lastName, username, email, and city—there are dozens of possibilities of what we could filter on. We don't want to automate every single combination; just enough to show that each filter is working correctly, and that some combinations are working as well.

Finally, we have the category of security tests. For example, if each request needs an authorization token to run, we can test the following:

- POST a new user without an authorization token and verify that we get a 401 response.
- GET a user with an invalid token and verify that we get a 403 response.
- PUT an update to the user with an invalid token and verify that we get a 403 response.
- DELETE a user without an authorization token and verify that we get a 401 response.

For each request, you'll want to test for both a 401 response (an unauthenticated user's request) and a 403 response (an unauthorized user's request).

Several additional tests may be appropriate for an API you are testing.

But these examples should get you thinking about the four different types of tests.

Now let's take a look at how we might use these four types in automation! First, we want to have some smoke tests that will run very quickly when code is deployed from one environment to another, up the chain to the production environment. We want these tests to simply verify that our endpoints can be reached. So all we need to do is run the first category of tests: the simple happy path requests. In our example API, we have four request types, so we need to run four tests. This will only take a matter of seconds.

We'd also like to have some tests that run whenever new code is checked in. We want to make sure the new code doesn't break any existing functionality. For this scenario, I recommend doing the first two categories of tests: the simple happy path requests and the simple negative requests. We could have one positive and one or two negative tests for each request, and this will probably be enough to provide accurate feedback to developers when they are checking in their code. In our example API, this amounts to no more than 12 tests, so our developers will be able to get feedback in about one minute.

Finally, it's also great to have a full regression suite that runs nightly. This suite can take a little longer because no one is waiting for it to run. I like to include tests of all four types in the suite, or sometimes I create two nightly regression suites: one that has the first three types of tests and one that has just the security tests. Even if you have 100 tests, you can probably run your full regression suite in just a few minutes because API tests run so quickly.

Once you have your smoke, build, and regression tests created and set to run automatically, you can relax in the knowledge that if something goes wrong with your API, you'll know it. This will free you up to do more exploratory testing!

# Chapter 84: Setting Up UI Tests

There are dozens of different ways to run automated UI tests, but this can make things more confusing because it's hard for someone new to automation to figure out what to do. And once you've chosen what UI testing tool to use, it can still be so frustrating to get your first end-to-end test to work.

That's why I'm so excited about Cypress! Cypress takes the most difficult part of UI testing—browser compatibility—and makes it easy. With Selenium Webdriver tests, you need to download a browser driver for the browser you'll be testing with, and you need to make sure the test knows how to find the driver. But with Cypress, the tests run directly in the browser, so there's no need to download a browser driver.

In this chapter, I'll walk you through getting started with Cypress for your UI automation project. I'll assume you already have Node.js installed; if you don't, install it and make sure Node has been added to your system path.

## Step #1: Create a Project

Open your command window, navigate to where you'd like to start your project, and type `mkdir MyFirstCypressProject`. This will create a new folder for the project. Navigate to that folder by typing `cd MyFirstCypressProject`, and then type `npm init --y`. This will initialize the project as a Node.js project.

## Step #2: Install Cypress

While still in the `MyFirstCypressProject` directory, type `npm i cypress --save-dev`. This will install Cypress in your project as a development dependency.

## Step #3: Open Cypress

While still in the `MyFirstCypressProject` directory, type `npx cypress open`. This will start Cypress, and you'll see a couple of things happen:

- Cypress will recognize that you are using it for the first time and will install some sample tests.
- Cypress will open the Cypress Test Runner.

## Step #4: Run the Sample Tests

To see how fast and versatile Cypress is, try running the sample tests. Open [MyFirstCypressProject](#) in your favorite code editor. I recommend Visual Studio Code. It is available for both Windows and Mac and works with a number of different languages, including Node.

In the code editor, open the [cypress](#) folder, then the [integration](#) folder, then the [1-getting-started](#) folder. You should see a file called *todo.spec.js* with some simple test examples. You can also look in the [2-advanced-examples](#) folder, which has many more example tests.

Go back to the Cypress Test Runner and click on the “Run 20 integration specs” link. You’ll see a new test window open, with a browser on the right and the test specs on the left, and you’ll see all the example tests fly by. You are now ready to begin writing your own tests.

The most important thing to remember about automated UI testing is that it should be done sparingly! Whatever you can test with unit and services tests should be tested that way instead. UI testing is best for validating that elements are on a web page and for running through simple user workflows.

# Chapter 85: Understanding the DOM

To find and use web elements by CSS selector or XPath for UI automation, it is very helpful to understand the DOM. The DOM (Document Object Model) is simply the interface that is used to interact with HTML and XML documents. When a JavaScript program manipulates elements on a page, it finds them through the DOM.

If you already understand HTML, it will be very easy to understand how the DOM is organized. The DOM is what is created when the HTML code has been parsed by the browser.

If you have little knowledge of HTML, here's an example of how web elements are organized on a page:

```
<html>
<head>
    <title>My Web Page</title>
</head>
<h1>My Web Page</h1>
<p>This is a paragraph</p>
<table border="2">
    <tr>
        <td>Row 1, Column 1</td>
        <td>Row 1, Column 2</td>
    </tr>
    <tr>
        <td>Row 2, Column 1</td>
        <td>Row 2, Column 2</td>
    </tr>
</table>
</html>
```

If you'd like to see what this web page would look like, you can simply copy and paste this HTML code into a Notepad file and save it with the *.html*

extension. Then find the file in the folder where you saved it and double-click on it. It should open as a web page.

Now let's take a look at the various elements on the page. Note that each element has a beginning and ending tag. For example, the line that says "This is a paragraph" has a `<p>` tag to indicate the beginning of the paragraph and a `</p>` tag to indicate the end of the paragraph. Similarly, the title of the page has a beginning tag, `<title>`, and an ending tag, `</title>`.

Notice that elements can be nested within each other. Look at the `<table>` tag, and then find the `</table>` tag several lines below it. In between the tags, you will see row tags (`<tr>`) and table data tags (`<td>`). Everything in between the `<table>` and `</table>` tags is part of the table.

Now look at the first `<tr>` tag and the first `</tr>` tag. Notice that there are two pairs of `<td></td>` tags in between. Everything between the first `<tr>` tag and the first `</tr>` tag is a row of the table. The `<td></td>` pairs in the row are elements of data in the row.

Now imagine that this data is organized into tree form:

| table |      |      |      |
|-------|------|------|------|
| row   |      | row  |      |
| data  | data | data | data |

If you were going to traverse the DOM to get at the data in Row 1, Column 1, you'd start by finding the `<table>` element, then the first `<row>` element, and then the first `<data>` element. In the next chapter, this is how we will use CSS selectors to find elements.

# Chapter 86: Locating Web Elements

When writing automated UI tests, you'll need to be able to locate elements. In Cypress, you can find an element by ID, text, class, name, data-cy, or CSS selector.

To decide which locator strategy to use, you'll need to know how to inspect an element. We'll go through examples of some of these locators, but before we do, let's create a new test file.

In the integration folder of your Cypress project that you created in Chapter 84, create a new folder called `contact-list-app`. Inside that folder, create a file called `contactList.spec.js`. Add the following code to that file:

```
describe('Element Locator Tests', () => {
  it('Can locate an element by id', () => {
    cy.visit('https://thinking-tester-contact-list.herokuapp.com')
  })
})
```

Now we're ready to practice locating items.

## Find by ID

Go to the Contact List app in a web browser, right-click in the Email field on the Login screen, and choose Inspect. You'll see the Developer Tools open, and the HTML code for the element will be highlighted: `<input id="email" placeholder="Email">`. In our test file, we'll locate the element using the id and add some text to the element so that we are sure we have located it correctly.

Add this line of code just under the `cy.visit` line:

```
cy.get('#email').type('I located the element!')
```

Save your code, and go to the Cypress test window. If the window isn't

open, navigate to your project in the command line and type `npx cypress open`.

You should see in the left section of the window that there is now a contact-list-app folder with your `contactList.spec.js` file in it. Click on that file to run that one test. You should see the browser test window open and your test should run. If you located the element correctly, you should see “**I located the element!**” in the email field:

# Contact List

Welcome! This application is for testing p

Log In:

I located the element!

Password

Submit

Not yet a user? Click here to sign up!

Sign up

Find by Text

In the Contact List app in your browser, right-click on the Submit button to inspect the element. You will see this element highlighted: <button

`id="submit">>Submit</button>`. The button element has Submit for its text. We'll use this text to interact with the element.

Add a new test to your test file, just below the previous test:

```
it('Can locate an element by text', () => {
  cy.visit('https://thinking-tester-contact-list.herokuapp.com')
  cy.contains('Submit').click()
})
```

When you save your new test, the test runner will automatically kick off the tests in the `contactList.spec.js` file. You should see both of your tests run and pass. Note that you'll see an error message in the browser for your second test. This is because you clicked the Submit button without adding an email or password. We'll log in as part of our next test.

## Find by Class

Add a new test to your spec file that begins by logging in to the application:

```
it('Can locate an element by class', () => {
  cy.visit('https://thinking-tester-contact-list.herokuapp.com')
  cy.get('#email').type('testuser@fake.com')
  cy.get('#password').type('mysecurepassword')
  cy.contains('Submit').click()
})
```

Log in to the browser with the credentials in the preceding code, or with your own credentials; then right-click on the Logout button and choose Inspect. You should see this element highlighted in DevTools:

```
Button           id="logout"           class="logout"
onclick="location.href='logout'">Logout</button>
```

We'll use the class this time to access the element. Add this line of code just below the step where you clicked on the Submit button:

```
cy.get('.logout').click()
```

Save your test, and watch it run and pass in the test runner.

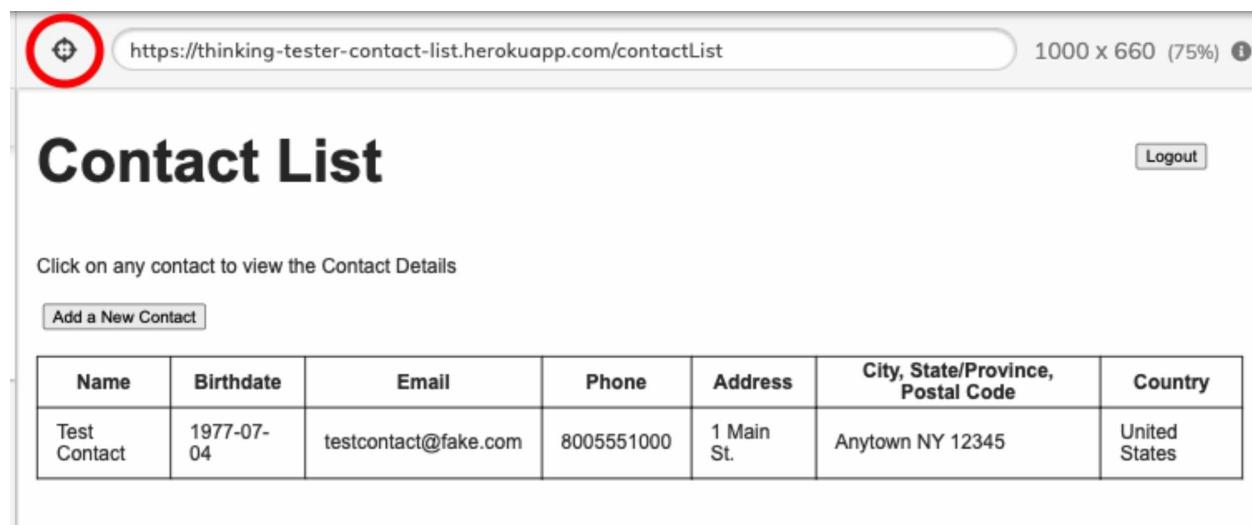
## Find by CSS Selector

Add another new test to your spec file that begins by logging in to the application:

```
it('Can locate an element by css', () => {
  cy.visit('https://thinking-tester-contact-list.herokuapp.com')
  cy.get('#email').type('testuser@fake.com')
  cy.get('#password').type('mysecurepassword')
  cy.contains('Submit').click()
})
```

Save and run this test, and then take a look at the test runner window. We're going to use the Cypress Selector Playground to get the CSS for the first name in the contact list.

Click on the selector icon in the browser window:



The screenshot shows a web browser window with the URL <https://thinking-tester-contact-list.herokuapp.com/contactList>. The window title bar indicates a resolution of 1000 x 660 (75%). In the top-left corner of the browser frame, there is a small circular icon containing a crosshair symbol, which is highlighted with a red circle. The main content area displays a "Contact List" page. At the top right, there is a "Logout" button. Below the header, a message says "Click on any contact to view the Contact Details". There is a "Add a New Contact" button. A table lists one contact entry:

| Name         | Birthdate  | Email                | Phone      | Address    | City, State/Province, Postal Code | Country       |
|--------------|------------|----------------------|------------|------------|-----------------------------------|---------------|
| Test Contact | 1977-07-04 | testcontact@fake.com | 8005551000 | 1 Main St. | Anytown NY 12345                  | United States |

This will open the selector playground. Now click on the element you want to select. In this case, we want to click on the Test Contact name. You'll see a CSS selector appear in the window:



Click on the copy icon to copy the entire cy.get command:



Now paste what you copied into your new test, just under the step where you click the Submit button. Then append .click() onto the end of that command so that your test looks like this:

```
it('Can locate an element by css', () => {
  cy.visit('https://thinking-tester-contact-list.herokuapp.com')
  cy.get('#email').type('testuser@fake.com')
  cy.get('#password').type('mysecurepassword')
  cy.contains('Submit').click()
  cy.get('.contactTableBodyRow > :nth-child(2)').click()
})
```

Save the test, and you should see it run and click through to the Contact Details for that contact.

You now have four tests using four different types of locators! We didn't add tests for name and data-cy selectors, simply because there aren't any selectors of those types in the Contact List app. But in case you encounter them in other applications, here is the syntax to use to select them:

- **Name:** cy.get('[name=submission]')
- **data-cy:** This is a special selector that you or your developer can add to an element to make it really easy to locate. It would look like this in your HTML: `data-cy="submit"`, where "submit" is replaced by whatever ID you'd like to give the element. To access it you would use `cy.get('[data-cy=submit]'`.

# Chapter 87: Automating UI CRUD Testing

There are a number of different patterns we can use to automate CRUD testing. At a minimum, we want to test one operation of each: Create, Read, Update, and Delete. For the purposes of this discussion, let's use the simple form mentioned in Chapter 6:

The diagram shows a rectangular form with a thin black border. Inside, at the top center, is the bold text "Add User". Below this, on the left, is the label "First Name" followed by a rectangular input field. Inside the input field is the text "Fred". At the bottom center of the form is a large rectangular button with the word "Submit" in its center.

This is the pattern I like to use when testing CRUD:

Scenario: Adding a user

Given I am adding a new user

When I add a first name and save

Then I navigate to the Users page

And I verify that the first and last names are present

Scenario: Updating a user

Given I am updating a user

When I change the first name of the user and save

Then I navigate to the Users page  
And I verify that the first name has been updated

Scenario: Deleting a user  
Given I am deleting a user  
When I delete the user and save  
Then I navigate to the Users page  
And I verify that the name is not present

These three tests have tested Create, Update, and Delete. The first two tests are also testing Read because we are retrieving the user for our assertions. Therefore, with these three tests I'm testing the basic functionality of CRUD.

It would also be a good idea to test some negative scenarios with our CRUD testing, such as creating a user with an invalid first name and updating a user with an invalid first name. These tests could look like this:

Scenario: Creating a user with an invalid first name  
Given I am adding a new user  
When I enter an invalid first name and save  
Then I verify that I receive the appropriate error message on the page  
And I navigate to the Users page  
And I verify that the user has not been added

Scenario: Updating a user with an invalid first name  
Given I am updating an existing user  
When I update the first name with an invalid value and save  
Then I verify that I receive the appropriate error message on the page  
And I navigate to the Users page  
And I verify that the existing first name has not been updated

These five scenarios—the three to test the happy path and the two negative tests—would be a great regression suite for our form. This is a very simple form, with just one field, which is not exactly a real-world scenario. But it is a good way to start thinking about automated UI test patterns.

# Chapter 88: Automated Form Testing

Now that we have looked at how we can automate CRUD testing, let's take a look at how we would do UI automation on a form.

\* First Name:  Last Name:

Date of Birth:

Email:  Phone:

Street Address 1:

Street Address 2:

City:  State or Province:

Postal Code:  Country:

We would want to ensure the following when we set up a regression test:

- Every field can be populated with data and saved.
- All the required fields are still required and we get an appropriate error message when we leave a required field blank.
- Validation rules are respected in each field and we get an appropriate error message when a value has failed validation.
- Both the Submit and Cancel buttons work correctly.

Since we know that automated UI tests can be slow and flaky, we'll want to limit the number of times we start up a browser for testing. But we'd also like our tests to only assert on one thing at a time. Here are my suggestions for a suite of tests that could be run on this form.

### Scenario: Happy path

Given I am adding a new user

When I fill out all the fields and click the Save button

And I navigate to the Contacts page

Then I verify that all the fields are displayed on the page

### Scenario: Required fields

Given I am adding a new user

When I leave all fields empty

And I click the Save button

Then I verify that an error message is present for each required field

### Scenario: Validation rules

Given I am adding a new user

When I give each field an invalid value

And I click the Save button

Then I verify that an error message is present for each invalid field

### Scenario: Cancel button

Given I am adding a new user

When I give each field a value

And I click the Cancel button

Then I verify that all fields are now empty

We can cover a lot of territory with just four automated tests! Our happy path scenario tests that values in all the fields will save correctly, and it also tests the Save button. If one of the values does not save, we will know we have an error. If all the values do not save, we will know there is a problem either with the Save button or with the underlying data store.

Our required fields test checks to make sure every field that is required displays an appropriate error message if it is not filled out. If even one required field does not display the message, we will know there is a problem.

Our validation rules test violates one rule for each text field. It would be a good idea to mix and match the different types of rule violations. For example:

- First Name: Send a value with numbers and nonletter characters, such as \$23.00.
- Last Name: Send a value with too many characters.
- Date of Birth: Send a value that is missing a year, such as 12-13.
- Email: Send a value that is not a valid email address, such as foo@bar.
- Phone: Send an invalid phone number, such as 1234.
- Street Address 1, Street Address 2, City, State or Province, Postal Code, Country: Send a value with too many characters.

While this does not test every single way in which the rules could be violated in every text field, it covers a wide variety of possibilities and verifies that for each error, you receive an appropriate error message. You could always add more tests here if you felt that more coverage was needed.

Finally, we verify that the Cancel button works correctly.

With just these four automated tests, we're able to make sure no functionality is broken. Running these regression tests will free you up to do more exploratory testing in your application, and test new and more interesting features.

# Chapter 89: Automated Visual Testing

Visual tests are more than just UI tests; they verify that what you are expecting to see in a browser window is actually rendered correctly. Traditional UI tests might verify that a page element exists or that it can be clicked on, but they don't validate what the element looks like. For example, without visual testing, you might have a button with text that spills over the edge, and your UI tests won't ever alert you. Fortunately, there are a variety of ways to do visual validation. My favorite way is through Applitools, which has a limited free version.

I've created an example project that will run some visual tests on the Contact List app using Cypress and Applitools. You can download or clone the project here: <https://github.com/kristinjackvony/visualTestExample>.

To run the tests, you'll need to sign up for a free Applitools account. This is easy to do, but you will need to use a work email to sign up since Applitools won't accept personal accounts such as Gmail accounts. Once you have signed up and logged in, you'll need to find your API key. You can do this by clicking on the user icon in the upper-right of the screen and choosing My API Key.

Next, go to the command line and navigate to the visualTestExample folder. Run `npm i` to install the modules you'll need to run the tests. Now type

`APPLITOOLS_API_KEY="" npx cypress run`, replacing <your api key> with your personal API key.

You should see two tests run and pass in the command line. If you want to watch them run in the browser, you can replace “`run`” with “`open`” in the preceding command.

Open the project in your favorite code editor and take a look at the `visualTests.spec` file. You'll see there are two tests: one that validates that the Login page renders correctly and one that checks that an error message

appears correctly. Applitools Eyes uses these commands to run visual checks:

- `cy.eyesOpen()` tells Eyes to begin running.
- `cy.eyesCheckWindow()` takes a screenshot of the current window and compares it to a saved screenshot.
- `cy.eyesClose()` tells Eyes to stop running.

Go to the Applitools page and you should see your test results:

| Test             | Browser      | OS    | Started         | Status |
|------------------|--------------|-------|-----------------|--------|
| Validation Error | Firefox 89.0 | Linux | 30 Jun 2021 ... | Passed |
| Validation Error | Chrome 91.0  | Linux | 30 Jun 2021 ... | Passed |
| Login Page       | Firefox 89.0 | Linux | 30 Jun 2021 ... | Passed |
| Login Page       | Chrome 91.0  | Linux | 30 Jun 2021 ... | Passed |

You should see that the two tests ran twice: once on Firefox and once on Chrome. I configured these browsers in the `applitools.config.js` file of my project. If you click through to one of the tests, you can see the screenshot that was taken.

Your tests will appear as Unresolved or Failed the first time you run them. This is because Applitools doesn't have any saved images on file. Once you have saved the images by clicking on the thumbs-up icon, and clicked the save icon in the upper right of the screen, you should be able to run the tests and see them pass.

Visual testing is a great way to validate that elements appear correctly on a page. Because the visual tests slow your test automation down a bit, you'll want to use the tests sparingly, only validating the most important screens.

# Chapter 90: Automated Security Testing

Often when people think of security testing, they think of complicated software scans, request intercepts, and IP address spoofing. But some of the most crucial application security testing can be done simply through API requests. In this chapter, we'll take a look at examples of authentication testing and authorization testing.

I created a suite of Postman tests to show examples of security testing, which you can download here: <https://github.com/kristinjackvony/securityTestExample>. You can import the JSON file into Postman using Postman's Import button. To run the tests, use the Environment file you created in Chapter 82.

## Authentication Testing

The first kind of security test to run is a simple login test. You'll want to make sure no one can log in with incorrect credentials or with an empty username or password field. You'll also want to check that the password respects casing; if a valid password is “Foobarfoo”, the API should not accept “foobarfoo” or “fooBarfoo”.

The login tests in my Postman collection begin by creating a user, then logging out as that user (because the user creation call automatically logs the user in). Then I go through a series of bad logins and verify that I get a 401 response rather than a 200 response. This indicates that I haven't been able to log in. Finally, I clean up the test suite by deleting the user account. I test all the possible combinations of having an empty username, an incorrect username, and a correct username with an empty password, an incorrect password, a password with the wrong casing, and a correct password.

The next way I test authentication is by making sure a user can't make an API request without a valid token. For simplicity, I'm going to focus on the contact operations in my Postman suite, but be aware that it's possible to create tests like these for the user operations as well.

For my token tests, I begin by creating a user and one contact for use in the tests. Then for each type of request—Add Contact, Get Contact List, Get Contact, Update Contact, Partial Update Contact, and Delete Contact—I have three tests: I try the request with no token, with an invalid token (“FOO”), and with a token that’s not found in the system. With each test I validate that I get a 401 response. Finally, I clean up my data by deleting the contact and the user I created in the setup.

## **Authorization Testing**

The final type of test I do makes sure a user can’t do something they are unauthorized to do. In the case of the Contact List app, this means one user should not be able to view, change, or delete another user’s contacts.

In my Authorization Tests folder, I create two users and add one contact to each user. I do a GET Contact List request and make sure only User 1’s contact is in User 1’s list and only User 2’s contact is in User 2’s list. Then I make sure User 1 can’t make any requests for User 2’s contact and User 2 can’t make any requests for User 1’s contact. With all of these requests, I’m expecting a 404 in response. A 403 would also be an appropriate response code, but I’m using a 404 so as not to give a malicious user any information about the fact that the contact exists. Finally, I clean up by deleting both contacts and both users.

These simple security tests are not particularly glamorous and will never make headlines. But it’s tests like these that can catch a security problem long before it becomes an issue. Once, I was alerted that some tests I had set up to run nightly in a QA environment were failing. Upon investigation, I discovered that my test user was now able to access information belonging to another user. If I hadn’t had authorization tests in place, that security hole might have been missed and might have made it to production.

# Chapter 91: Automating Load Tests

Load testing is a key part of checking the health of your application. Just because you get a timely response when you make an HTTP request in your test environment doesn't mean the application will respond appropriately when 100,000 users are making the same request in your production environment.

With load testing, you can simulate different scenarios as you make HTTP calls to determine how your application will behave under real-world conditions.

A wide variety of load testing tools are available, but many of them require a subscription. Both paid and free tools can be confusing to use or difficult to set up. For load testing that is free, easy to install, and fairly easy to set up, K6 is a good option.

To help demonstrate how K6 works, I created an automation script that you can download here: <https://github.com/kristinjackvony/loadTestExample>. This script is a simple load test for the Contact List app. To run the load test script, you'll need to install K6, which is easy to do with the instructions found here: <https://k6.io/docs/getting-started/installation>.

Once you have installed K6 and downloaded the test script, open a command window and navigate to the location where you downloaded the script. Then type `k6 run loadTestScript.js`. The test should run and display a number of metrics as the result.

Let's take a look at what this script is doing.

```
import http from "k6/http";
import { check } from k6";
```

In the first two lines of the script, I'm importing the modules needed for the script: the `http` module that allows me to make HTTP requests, and the `check` module that allows me to do assertions.

```
export let options = {  
  vus: 1,  
  duration: "5s"  
};
```

In this section, I'm setting the options for running my load tests. The word “vus” stands for “virtual users,” and “duration” describes how long in seconds the test should run.

```
export function setup() {  
  var url = "https://thinking-tester-contact-list.herokuapp.com/users/login"  
  var payload = JSON.stringify({ "email": "test@fake.com", "password": "foobarfoo" })  
  var params = { headers: { "Content-Type": "application/json" } }  
  let response = http.post(url, payload, params);  
  let JSONResponse = JSON.parse(response.body)  
  let token = JSONResponse.token  
  return token  
}
```

In this section, I'm setting up the tests by doing a login request and saving the authentication token, which I'll then be able to use in all my other requests. The return token command is how I pass the variable from the setup function to the default function.

```
export default function(data) {  
  var token = data  
  var url = "https://thinking-tester-contact-list.herokuapp.com/contacts"  
  var bearer = "Bearer " + token  
  var params = { headers: { "Authorization": bearer, "Content-Type": "application/json" } }
```

In these lines, I'm calling the default function, which is where the tests are run. I'm passing in a data variable. This data variable is actually the token that I returned from the previous function. Then I'm assigning the data variable as “token” because that will be easier to remember when I'm creating my requests. Next, I'm creating the url variable, the bearer variable for authentication, then the params (parameters) that will need to be passed in

with my requests. Now I'm ready to start making my requests.

```
let getResponse = http.get(url, params)
check(getResponse, {
  "get contact list response status is 200": (r) => r.status == 200,
  "response transaction time is OK": (r) => r.timings.duration < 1000
});
```

This is the request to get the contact list. I do the get request and pass in the url and the params. Then I check to see that I got a 200 response and that the response transaction time was less than one second.

I'll leave it up to the reader to look at and interpret the remaining five requests. Now let's take a look at the load test output:

- ✓ get contact list response status is 200
- ✓ response transaction time is OK
- ✓ add contact response status is 201
- ✓ get contact response status is 200
- ✓ update contact response status is 200
- ✓ partial update contact response status is 200
- ✓ delete contact response status is 200

You can see that all the requests were successful and that the response transaction time was OK for each transaction.

```
http_req_duration.....: avg=161.96ms min=83.91ms med=162.98ms
max=448.21ms
```

This section shows metrics about the duration of each request. The average request duration was 161.96 milliseconds, the minimum request time was 83.91 milliseconds, the median request time was 162.98 milliseconds, and the maximum request time was 448.21 milliseconds.

```
iterations..... : 6  0.948523/s
vus..... : 1  min=1 max=1
vus_max..... : 1  min=1 max=1
```

This section shows how many complete iterations were run during the test and what the frequency was; how many virtual users there were; and the maximum number of virtual users.

Obviously, this wasn't much of a load test, because we only used one user and it only ran for five seconds! Let's make a change to the script and see how our results change.

First we'll leave the number of virtual users at 1, but we'll set the test to run for a full minute. Change line 6 of the script to `duration: "1m"`, and run the test again with the `k6 run loadTestScript.js` command.

```
http_req_duration.....: avg=181.14ms min=71.1ms med=156.71ms  
max=748.72ms
```

The results look very similar to our first test, which isn't surprising, since we are still using just one virtual user.

Let's add some more virtual users now. Change line 5 of the script to `vus: 10` and run the script again.

```
http_req_duration.....: avg=983.7ms min=134.94ms med=981.45ms  
max=2.28s
```

Now we can see that the response times have slowed a bit, which makes sense since we are now under more load. Also, since we were validating that response times took less than one second, we can see that about half of our transaction time tests failed:

- ✓ get contact list response status is 200
- ✗ response transaction time is OK
  - ↳ 51% — ✓ 321 / ✗ 303
- ✓ add contact response status is 201
- ✓ get contact response status is 200
- ✓ update contact response status is 200
- ✓ partial update contact response status is 200
- ✓ delete contact response status is 200

This is by no means a complete load test; it's just an introduction to what can be done with the K6 tool. It's possible to set up the test to have realistic ramp-up and ramp-down times, where there's less load at the beginning and end of the test and more load in the middle. You can also create your own custom metrics to make it easier to analyze the results of each request type. And you can integrate your K6 tests with your continuous integration tool so that every time there's a change to your product's code, the K6 tests will run and let you know whether your response times have slowed.

# Chapter 92: Automated Accessibility Tests

Accessibility in the context of a software application means that as many people as possible can use the application easily. When making an application accessible, we should consider users with limited vision and/or hearing, limited cognitive ability, and limited dexterity. Fortunately, there are a wide variety of products that can help us automate accessibility testing by checking an application according to accepted accessibility standards such as the Web Content Accessibility Guidelines (WCAG). I created a couple of example tests that use cypress-axe, an npm package that runs axe (<https://www.deque.com/axe>) accessibility testing in Cypress. You can find my tests here: <https://github.com/kristinjackvony/accessibilityTestExample>.

Once you have downloaded the example folder, you can `cd` to the folder in the command line and run `npm i` to install all the packages you'll need to run the tests. To open Cypress, run `npx cypress open`. Then click the “Run 2 integration specs” link in the Cypress window to run the tests.

You'll see the tests run and have test failures. The descriptions of the failures in the Cypress Test Runner are not particularly helpful, but they can be a jumping-off point for further investigation. For example, if you hover over the “image-alt on 1 Node” error, you'll see the Thinking Tester logo highlighted. This is because I haven't put in an alt text for the logo. The “color-contrast on 3 Nodes” error is telling us that the light-blue text I've used on the screen is too light to provide a contrast with the white background.

Let's take a quick look at how to set up the cypress-axe tests. They require two npm packages: `axe-core` and `cypress-axe`. In the `support/commands/index.js` file, I've imported `cypress-axe` with the `import 'cypress-axe'` command.

In the `beforeEach` section of each test file, I've navigated to the page I want to test and then used the `cy.injectAxe()` command. This loads `axe-core` into the page under test. Then, in the test itself, I used the `cy.checkA11y()`

command. This runs the accessibility checks on the page under test.

Once you have located all the accessibility errors on the page and fixed them so that your test is passing, you can integrate your cypress-axe tests into your build system. This way, if any new accessibility issues are introduced, you'll be notified of them immediately by the failing test.

# Chapter 93: Automation Wheel Strategy: Moving from What to How to When to Where

In Chapter 79, I wrote about how I would decide what to test in a simple application in terms of testing every segment of the automation test wheel. I find it's very helpful to answer the question "What do I want to test?" before I think about how I'm going to test it.

In this chapter, we'll look at how to take the "what" of automated testing and continue on with *how* I want to test, *when* I want to test it, and *where* (what environment) I'm going to test.

## How I'm Going to Test

I'm going to run my unit and component tests directly in the code. Unit tests are designed to specifically run in the code because they test only the code and not any external interactions. My component tests are very simple—just one call to the database and one call for authentication—so I will run those directly from my code as well.

For my services tests, I'm going to use Postman. I'll run the Postman tests using Newman. I'll also include some security tests in Postman, validating that any requests without appropriate authentication return an appropriate error, and I will also do some performance checks here, verifying that the response times to my API requests are within acceptable levels.

For my UI tests, I'm going to use Cypress. I'll be adding a few security tests here, making sure pages do not load when the user doesn't have access to them. I'll also be integrating my visual tests into my Cypress tests using Applitools, and I'll be using both Cypress and Applitools to run my accessibility tests.

Finally, I will set up a performance testing tool such as Pingdom that will regularly monitor my page load times and alert me when load times have slowed, as well as include some assertions in my Postman tests that will let

me know when API response times are slower than expected.

## When I'm Going to Test

Now that I've figured out how I'm going to test, it's time to think about when I'm going to run my tests. I'm going to organize my tests into four events:

- **With every build:** Every time new code is pushed, I'm going to run my unit tests, component tests, and API tests. These tests will give me feedback very quickly. I'm not going to run any UI tests at this time, because I don't want to slow down my feedback time.
- **With every deploy:** Every time code is deployed to an environment, I'm going to run all my API tests and a small subset of my UI tests. My UI tests will include at least one visual check and one security check. This will ensure that the API is running exactly as it should and that there are no glaring errors in the UI.
- **Daily:** I'll want to run all my API tests and all my UI tests early in the morning, before I start my workday. When I begin my workday I'll have a clear indication of the health of my application.
- **Ongoing:** I'll have Pingdom monitoring my page load times throughout the day to alert me of any performance problems. I'll also set up a job to run a small set of API tests periodically throughout the day to alert me of any slow API responses.

## Where I'm Going to Test

Now that I've decided how and when to test, I need to think about where to test. Let's imagine my application has four different environments: Dev, QA, Stage, and Production. My Dev environment is solely for developers. My QA environment is where code will be deployed for manual and exploratory testing. My Stage environment is where a release candidate will be prepared for Production. Let's look at what I will test in each environment.

**Dev:** My unit and component tests will run here whenever a build is run,

and my Postman and Cypress tests will run here whenever a deploy is run.

**QA:** I'll run my full daily Postman and Cypress suites here, and I'll run my full Postman suite and a smaller Cypress suite with each deploy.

**Stage:** I'll run the full sets of Postman and Cypress tests when I deploy. This is because the Stage environment is the last stop before Production, and I'll want to make sure we haven't missed any bugs. I'll also run my Pingdom monitoring here, to catch any possible performance issues before we go to Production.

**Production:** I'll run a small set of daily Postman and Cypress tests here. I'll also point my Pingdom tests to this environment, and I'll have those tests and a set of Postman tests running periodically throughout the day.

## Putting It All Together

When viewed in prose form, this all looks very complicated. But we have actually managed to simplify things down to four major test modalities tested at four different times. Are we covering all the areas of the automation test wheel? Let's take a look:

| <b>Code</b> | <b>Postman</b> | <b>Cypress</b> | <b>Pingdom</b> |
|-------------|----------------|----------------|----------------|
| Unit        | Services       | UI             | Performance    |
| Component   | Security       | Security       |                |
|             | Performance    | Visual         |                |
|             |                | Accessibility  |                |

We are covering each area with one or more testing modalities. Now let's visualize our complete test plan:

|              | <b>Hourly</b> | <b>Daily</b> | <b>Build</b> | <b>Deploy</b> |
|--------------|---------------|--------------|--------------|---------------|
| <b>Dev</b>   |               |              | Unit         | Postman       |
|              |               |              | Component    | Cypress       |
|              |               |              | Postman      |               |
| <b>QA</b>    |               | Postman      |              | Postman       |
|              |               | Cypress      |              | Cypress       |
| <b>Stage</b> | Pingdom       |              |              | Postman       |
|              |               |              |              | Cypress       |
| <b>Prod</b>  | Pingdom       | Postman      |              | Postman       |
|              | Postman       | Cypress      |              | Cypress       |

Viewed in a grid like this, our plan looks quite simple! By considering each question in turn:

- What do we want to test?
- How are we going to test it?
- When will we run our tests?
- Where will we run them?

we've been able to come up with a comprehensive plan that covers all areas of the testing wheel and tests our application thoroughly and efficiently.

# Chapter 94: How Flaky Tests Destroy Trust

Anyone who has ever written an automated test has experienced test flakiness. There are many reasons for flaky tests, including:

- Environmental issues, such as the application being unavailable
- Test data issues, in which an expected value has been changed
- UI issues, such as a pop-up window taking too long to appear

All of these reasons are valid explanations for flaky tests. However, they are not excuses! It should be your mission to have all your automated tests pass every single day, except, of course, when an actual bug is present.

This is important, not just because you want your tests to be reliable, but because when you have flaky tests, trust in you and in your team is eroded. Here's why.

## **Flaky Tests Send a Message That You Don't Care**

Let's say you are the sole automation engineer on a team and you have a bunch of flaky tests. It's your job to write test automation that actually checks that your product is running correctly, and because your tests are flaky, your automation doesn't do that. Your team may assume this is because you don't care whether your job is done properly.

## **Flaky Tests Make Your Team Question Your Competence**

An even worse situation than the preceding example is one in which your team simply assumes you haven't fixed the flaky tests because you don't know how. This further erodes their trust in you, which may spill over into other testing. If you find a bug when you are doing exploratory testing, your colleagues might not believe you have a bug, because they think you are technically incompetent.

## **Flaky Tests Waste Everyone's Time**

If you are part of a large company where each team contributes one part of an application, other teams will rely on your automation to determine whether the code they committed works with your team's code. If your tests are failing for no reason, people on other teams will need to stop what they are doing and troubleshoot your tests. They won't be pleased if they discover there's nothing wrong with the app and your tests are just being flaky.

## **Flaky Tests Breed Distrust Among Teams**

If your team has a bunch of flaky tests that fail for no good reason, and you aren't actively taking steps to fix them, other teams will ignore your tests and may also doubt whether your team can be relied upon. In a situation like this, if Team B commits code and sees that Team A has failing tests, Team B may do nothing about it, and may not even ask Team A about the failures. If there are tests that fail because there are real issues, your teams might not discover them until days later.

## **Flaky Tests Send a Bad Message to Your Company's Leadership**

There's nothing worse for a test team than to have some automation tests fail on a daily basis. This sends a message to management that either test automation is unreliable or you are unreliable!

So, what can we do about flaky tests? I recommend the following:

- **Set up retries for your UI automated tests:** UI tests are, without a doubt, the flakiest of tests. Tools like Cypress allow you to set a number of retries for each test; this way, if something unexpected happens, like an element taking much longer than usual to load, the test will retry. This one step alone can make your test runs more likely to pass.
- **Make a commitment to having 100% of your tests pass every day:** The only time a test should fail is if a legitimate bug is present. Some might argue that this is an impossible dream, but it is one to strive for. There is no such thing as perfect software, or perfect tests, but we can work as hard as we can to get as close as we can to that perfection.

- **Set up alerts that notify you of test failures:** Having tests that detect problems in your software doesn't help if no one is alerted when test failures happen. Set up an alert system that will notify you via email or chat when a test is failing. Also, make sure you test your alert. Don't assume that because the alert is in place it is automatically working. Make a change that will cause a test to fail, and check to see whether you got the notification.
- **Investigate every test failure and find out why the test failed:** If the failure wasn't due to a legitimate bug, what caused it? Will the test pass if you run it again, or does it fail every time? Will the test pass if you run it manually? Is your test data correct? Are there problems with the test environment?
- **Remove the flaky tests:** Some might argue that this is a bad idea because you are losing test coverage and the test passes sometimes. But this doesn't matter, because when people see that the test is flaky, they won't trust the passing test anyway. It's better to remove the flaky tests altogether so that you demonstrate that you have a 100% passing rate, and then others will begin to trust your tests. An alternative would be to set the flaky tests to be skipped, but this might also erode trust. People might see all the skipped tests and think they are a sign that you don't write good test automation. Furthermore, you might forget to fix the skipped tests.
- **Fix all the flaky tests you can:** How you fix the flaky tests will depend on why they are flaky. If you have tests that are flaky because someone keeps changing your test data, change your tests so that the test data is set up in the test itself. If you have tests that are flaky because sometimes your test assets aren't deleted at the end of the test, do a data cleanup both before and after the test.
- **Ask for help:** If your tests are flaky because the environment where they are running is unreliable, talk to the team that's responsible for maintaining the environment. See whether there's something they can do to solve the problem. If they are unresponsive, find out whether other teams are experiencing the issue, and lobby together to make a change.

- **Test your functionality in a different way:** If your flaky test is failing because some element on the page isn't loading on time, don't try to solve the issue by making your "wait" commands longer. Try to come up with a different way to test the feature. For example, you might be able to switch a UI test to an API test. Or you might be able to verify that a record was added in the database instead of going through the UI. Or you might be able to verify the data on a different page, instead of the one with the slow element. Some might say that not testing the UI on that problematic page is dangerous. But having a flaky test on this page is even more dangerous because people will just ignore the test. It would be better to stick with an automated test that works and do an occasional manual test of that page.

## **Quality Automation Is Our Responsibility**

We've all been in situations where we have been dismissed as irrelevant or incompetent because of the reputation of a few bad testers. Let's create a culture of excellence for testers everywhere by making sure every test we run is reliable and provides value!

## Part XI: Testing Strategy

# Chapter 95: The Power of Not Knowing

There is a concept called “intentional ignorance” in which a tester intentionally doesn’t read some of the documentation or code for a new feature. This can prevent the tester from having biases when doing exploratory testing.

In healthy Agile software teams, the testers are invited to the feature grooming sessions, acceptance criteria are written for each story, and the developers do a feature handoff with the testers when each story is ready for testing. In these situations, there’s not much chance that a tester will be ignorant about a feature.

But when I worked on teams that were less healthy, I was often given a story to test with no feature handoff and no acceptance criteria. Sometimes the story wouldn’t even have a description, and would have a cryptic title like “Endpoint for search.” Before asking for clarification on what was expected in the story, I would use the opportunity to do some exploratory testing while I had no preconceived notions of what the feature could or couldn’t do. And while testing in this fashion, I would often find a bug and show it to the developer, and they would say, “Oh, it never even occurred to me to test the feature in that way.”

Of course, I don’t want to go back to the days of cryptic story titles and no descriptions! But testing without knowing what the feature does can have some benefits:

- You approach the application the same way a user would. When your users see your new feature for the first time, they don’t have the benefit of instructions. By trying out the feature without knowing how it works, you could discover that an action button is hard to find or that it’s difficult to know what to do first on a multipart form.
- You might enter data that no one was expecting. For example, a form field could require that dates be entered with the month and day

only, but you enter the month, day, and year, which breaks the form.

- Without any instructions from the developer, you might think of other features to test the new feature with, in addition to the combinations the developer thought of. Those feature combinations might yield new bugs.

So, how can we add these advantages back into our testing and still read the acceptance criteria and have feature handoffs? Here are a few ways:

- Pair-test with someone on another team. At my company we have many teams, and each team often has little idea what the other teams are building. We will frequently pair two testers, each from a different team, and they swap applications and start testing. This is a great way to find bugs and user experience issues!
- When you start testing, spend some time just playing around with the new feature before writing a test plan. By exploring in this way, you might come up with some unusual testing ideas.
- After you've tested the acceptance criteria, take some time to think about what features might be used with the new feature. What happens when you test them together? For example, if you were testing a new page of data, you could test it with the global sort feature that already exists in your application.
- Before you finish your testing, ask yourself, "What else could I test?"

Sometimes not knowing all the details about a feature is detrimental. Often in my testing career, I have tested a feature and missed something that the feature could do because no one told me about it. That's why I'm glad my company has acceptance criteria and conducts feature handoffs. Still, not knowing the details can reveal interesting bugs.

# Chapter 96: The Power of Pretesting

Having been in the software testing business for a few years now, I've become accustomed to various types of testing: acceptance testing, regression testing, exploratory testing, and smoke testing, among others. But recently I was introduced to a type of testing I hadn't thought of before: pretesting.

I was on a team that was working to switch some automatically delivered emails from an old system to a new system. When we first started testing, we were mainly focused on whether the emails in the new system were being delivered. It didn't occur to us to look at the email content until later, at which point we realized we had never really looked at the old emails. Moreover, because the emails contained a lot of detail, we found that we kept missing things: some extra text here, a missing date there. We discovered that the best way to prevent these mistakes was to start testing before the new feature was delivered, and thus, pretesting was born.

Once we adopted the pretesting process, whenever an email was about to be converted to the new system we first tested it in the old system. We took screenshots, and we documented any needed configuration or unusual behavior. Then when the email was ready for the new system, it was easy to go back and compare it with what we had before. This was a valuable tool in our testing arsenal, and we used it in a number of other areas of our application.

## **When Should You Use Pretesting?**

It's helpful to pretest in the following situations:

- When you are testing a feature you have never tested before
- When no one in your company seems to know how a feature works
- When you suspect that a feature is currently buggy
- When you are revamping an existing feature
- When you are testing a feature that has a lot of detail

## **Why Should You Pretest?**

Pretesting will save you the headache of trying to remember how something “used to work” or “used to look.” If customers will notice the change you are about to make, it’s good to note how extensive the change will be. If there are bugs in the existing feature, it would be helpful to know this before the development work starts, because the developer could make those bug fixes while in the code. Pretesting is also helpful for documenting how the feature works so that you can share those details with others who might be working on or testing the feature.

## How to Pretest

When conducting a pretest, make sure you do the following:

1. Conduct exploratory testing on the old feature. Figure out what the happy path is.
2. Document how the happy path works and include any necessary configuration steps.
3. Continue to test, exploring the feature’s boundaries and edge cases.
4. Document any “gotchas” you may find. These are not bugs, but rather, areas of the feature that might not work as someone would expect.
5. Log any bugs you find and discuss them with your team to determine whether they should be fixed with the new feature or left as is.
6. Take screenshots of any complicated screens, such as emails, messages, or screens with a lot of text, images, or buttons. Save these screenshots in an easily accessible place, such as a wiki page or shared folder.

Then, when the new feature is ready:

1. Run through the same happy path scenario with the new feature and verify that it behaves the same way as the old feature.
2. Test the feature’s boundaries and edge cases and verify that they behave the same way as the old feature.

3. Verify that any bugs the team has decided to fix have actually been fixed.
4. Compare the screenshots of the old feature with the screenshots of the new feature and verify that they are the same (with the exceptions of anything the team agreed to change).
5. Do any additional necessary testing, such as testing new functionality that the old feature didn't have or verifying that the new feature integrates with other parts of the application.

The power of pretesting is that it helps you notice details you might otherwise miss in a new feature and, as a bonus, find existing bugs in the old feature. Moreover, testing the new feature will be easy because you will have already created a test plan. Your work will help the developer do a better job, and your end users will appreciate it!

# Chapter 97: Your Future Self Will Thank You

A couple of years ago I learned a lesson about the importance of keeping good records. I've always kept records of what tests I ran and whether they passed, but I have learned that there's something else I should be recording: how a feature used to behave.

I was testing a file API that allowed users to upload and download files. The metadata used to access the files was stored in a nonrelational database. As discussed in Chapter 29, nonrelational databases store their data in document form rather than in the table form found in SQL databases.

My team had made a change to the metadata for our files. After deploying the change, we discovered that older files could not be downloaded. It turned out that the change to the metadata had resulted in older files not being recognized, because their metadata was different. The bug was fixed, so now the change was backward compatible with the older files.

I added a new test to our smoke test suite that would request a file with the old metadata. Now, I thought, if a change was ever made that would affect that area, the test would fail and the problem would be detected.

Then my team made another change to the metadata. The code was deployed to the test environment, and shortly afterward, someone discovered that some files could no longer be downloaded.

I was perplexed! Didn't we already have a test for this? When I met with the developer who investigated the bug, I found out there was an even older version of the metadata that we hadn't accounted for.

Talking this over with the developers on my team, I learned that a big difference between SQL databases and nonrelational databases is that when a schema change is made to a relational database, it goes through and updates all the records. For example, if you had a table with first names and last

names, and someone wanted to update the table to now contain middle names, every existing record would be modified to have a null value for the middle name:

| <b>FirstName</b> | <b>MiddleName</b> | <b>LastName</b> |
|------------------|-------------------|-----------------|
| Prunella         | NULL              | Prunewhip       |
| Joe              | Bob               | Schmoe          |

With nonrelational databases, this is different. Because each entry is its own document and there are no nulls, it's possible to create situations in which a name-value pair simply doesn't exist. To use the preceding example, in a nonrelational database Prunella wouldn't have a "MiddleName" name-value pair:

```
{
  "FirstName": "Prunella",
  "LastName": "Prunewhip"
},
{
  "FirstName": "Joe",
  "MiddleName": "Bob",
  "LastName": "Schmoe"
}
```

If the code relies on retrieving the value for MiddleName, that code would return an exception because there's nothing to retrieve.

The lesson I learned from this situation is that when we are using nonrelational databases, it's important to keep a record of what data structures are used over time. This way, whenever a change is made we can test with data that uses any old structures as well as the new structure.

And this lesson is applicable to situations other than nonrelational

databases! There may be other times when an expected result changes after the application changes.

Here are some examples:

- A customer listing for an e-commerce site used to display phone numbers; now it's been decided that phone numbers won't be displayed on the page.
- A patient portal for a doctor's office used to display Social Security numbers in plain text; now the digits are masked.
- A job application workflow used to take the applicant to a pop-up window to add a cover letter; now the cover letter is added directly on the page and the pop-up window has been eliminated.

In all these situations, it may be useful to remember how the application used to behave in case you have users who are using an old version, there's an unknown dependency on the old behavior that now results in a bug, or a new product owner asks why a feature is behaving in the new way. When something like this happens, your future self will be grateful that you took the time to document how the feature used to behave!

# Chapter 98: How to Design a Test Plan

Being a software tester means much more than just running through acceptance criteria on a story. We need to think critically about every new feature and come up with as many ways as we can to test it. When many permutations are possible in a feature, we need to be thorough with testing but do so in a reasonable amount of time. Automation can help us test many permutations quickly, but too many people jump to automation without really thinking about what should be tested.

For example, say I needed to design a test plan for the Super Ball Sorter feature that we first discussed in Chapter 20. Here is how the feature works.

Super Balls can be sorted among four children—Amy, Bob, Carol, and Doug:

- The balls come in two sizes: large and small.
- The balls come in six colors: red, orange, yellow, green, blue, and purple.
- The children can be assigned one or more rules for sorting: for example, Amy could have a rule that says she only accepts large balls, or Bob could have a rule that says he only accepts red or orange balls.
- Distribution of the balls begins with Amy and proceeds through the other children in alphabetical order, continuing in the same manner as though one were dealing a deck of cards.
- Each time a new ball is sorted, distribution continues with the next child in the list.
- The rules used must result in all the balls being sortable; if they do not, an error will be returned.

- Your friendly developer has created a ball distribution engine that will create balls of various sizes and colors for you to use in testing.

Here's a quick example: say Carol has a rule that she only accepts small balls.

The first ball presented for sorting is a large red ball. Amy is first in the list, and she doesn't have any rules, so the large red ball will go to her.

The next ball presented is a small blue ball. Bob is second on the list, and he doesn't have any rules, so the small blue ball will go to him.

The third ball is a large purple ball. Carol is next on the list, but she has a rule that says she only accepts small balls, so the ball will not go to her. Instead, the ball is presented to Doug, who doesn't have any rules, so the large purple ball will go to him.

Here's what we have after the first pass:

- Amy: large red ball
- Bob: small blue ball
- Carol: no ball
- Doug: large purple ball

Since Doug had the most recent turn, we'd continue the sorting by offering a ball to Amy.

How should we test this? Before I share my plan, you may want to take a moment and see what sort of test plan you would design. Then you can compare your plan with mine.

My test plan design philosophy always begins with testing the simplest possible option, and then gradually adding more complex scenarios. So, my test plan would begin as follows.

### **Part 1: No Children Have Any Rules**

If no children have any rules, we should see that the balls are always evenly distributed among Amy, Bob, Carol, and Doug, in that order. If we

send in 20 balls, for example, we should see that each child winds up with five.

Next, I would test just one type of rule. There are only two parameters for the size rule but six parameters for the color rule, so I would start with the size rule.

## **Part 2: Size Rules Only**

We could have anywhere from one child to four children with a rule. We'll start with one child and work up to four children. Also, one child could have two rules, although that would be a bit silly, since the two size rules would be accepting large balls only and accepting small balls only, which would be exactly the same as having no rules. So let's write up some test cases:

1. One child has a rule:

- Amy has a rule that she only accepts large balls. At the end of the test pass, she should only have large balls.
- Bob has a rule that he only accepts small balls. At the end of the test pass, he should only have small balls.

B. Two children have rules:

- Amy and Bob both have rules that they only accept large balls. At the end of the test pass, they should only have large balls.
- Carol has a rule that she only accepts large balls, and Doug has a rule that he only accepts small balls. At the end of the test pass, Carol should have only large balls, and Doug should have only small balls.

C. Three children have rules:

- Amy, Bob, and Carol have rules that they only accept small balls. At the end of the test pass, they should only have small balls.
- Amy and Bob have rules that they only accept small balls, and Carol has a rule that she only accepts large balls. At the end of the test

pass, Amy and Bob should have only small balls, and Carol should have only large balls.

- Amy has a rule that she accepts both large balls and small balls, and Bob and Carol have rules that they only accept large balls. At the end of the test pass, Amy should have both large and small balls, and Bob and Carol should have only large balls.

D. Four children have rules:

- Amy and Bob have rules that they only accept large balls, and Carol and Doug have rules that they only accept small balls. At the end of the test pass, Amy and Bob should have only large balls, and Carol and Doug should have only small balls.
- All four children have a rule that they only accept large balls. This rule should return an error.

Now that we have extensively tested the size rule, it's time to test the color rule in isolation.

### **Part 3: Color Rules Only**

As with the size rule, anywhere from one to four children could have a color rule. But this rule type is a bit more complex because each child could have from one to six color rules. Let's start simple with one child and one rule:

1. One child has one rule:

- Bob accepts only red balls.
- Bob accepts only orange balls.
- Bob accepts only yellow balls.
- Bob accepts only green balls.
- Bob accepts only blue balls.
- Bob accepts only purple balls.

This tests that each color rule will work correctly on its own.

B. One child has more than one rule:

- Carol accepts only red and orange balls.
- Carol accepts only red, orange, and yellow balls.
- Carol accepts only red, orange, yellow, and green balls.
- Carol accepts only red, orange, yellow, green, and blue balls.
- Carol accepts only red, orange, yellow, green, blue, and purple balls (which, again, is sort of silly because it's like having no rule at all).

C. Two children have color rules:

- Amy and Bob both accept only red balls.
- Amy accepts only red balls and Bob accepts only blue balls.
- Amy accepts only red and green balls and Bob accepts only blue and yellow balls.
- Carol accepts only red, orange, and yellow balls and Doug accepts only green balls.

Note that there are many more possibilities here than what we are actually testing. We are merely trying out a few different scenarios, such as one child has three rules and one child has one rule.

D. Three children have color rules:

- Amy, Bob, and Carol accept only red balls.
- Amy accepts only red balls, Bob accepts only orange balls, and Carol accepts only yellow balls.
- Amy accepts only red balls, Bob accepts only red and orange balls, and Carol accepts only red, orange, and yellow balls.

The preceding scenario exercises a path in which the children share one rule but not other rules.

E. Four children have color rules:

- Amy, Bob, Carol, and Doug only accept purple balls. This should return an error.
- Amy only accepts red and yellow balls, Bob only accepts orange

balls, Carol only accepts yellow and blue balls, and Doug only accepts green balls. This should also return an error because no one is accepting purple balls.

- Amy only accepts red balls, Bob only accepts red and orange balls, Carol only accepts yellow balls, and Doug only accepts yellow, green, blue, and purple balls.

Now that we've exercised both rule types separately, it's time to try testing them together! Here's where things get really complicated. Let's start with simple scenarios in which each child has either a color rule or a size rule, but not both, and move on to more complex scenarios from there:

#### **Part 4: Size and Color Rules**

1. Children have one size rule or one color rule:

- Doug only accepts large balls, and Bob only accepts red balls.
- Doug only accepts large balls, Bob only accepts red balls, and Carol only accepts small balls.
- Doug only accepts large balls, Bob only accepts red balls, Carol only accepts small balls, and Amy only accepts yellow balls.
- Amy and Doug only accept large balls, Bob only accepts small balls, and Carol only accepts purple balls.
- Amy and Doug only accept large balls, and Bob and Carol only accept purple balls. This should return an error because there's no one to accept any of the small balls except for the purple ones.

B. Children have both a size and a color rule:

- Amy only accepts large red balls.
- Amy only accepts large red balls, and Bob only accepts small blue balls.
- Amy only accepts large red balls, Bob only accepts small blue balls, and Carol only accepts large green balls.
- Amy only accepts large red balls, Bob only accepts small blue balls, Carol only accepts large green balls, and Doug only accepts small yellow balls. This should return an error because there are many

balls that cannot be accepted by anyone.

C. Children have a size rule and more than one color rule:

- Amy only accepts large red, orange, and yellow balls; Bob only accepts small red, orange, and yellow balls; Carol only accepts large green, blue, and purple balls; and Doug only accepts small green, blue, and purple balls.
- Try the preceding scenario, but remove the large yellow ball from Amy's list. This should return an error because there's no one to accept the large yellow balls.

D. Children have more than one size rule and more than one color rule:

- Amy only accepts large red balls, large blue balls, and small yellow balls; Bob only accepts large orange balls, large purple balls, and small green balls; Carol only accepts large yellow balls, small red balls, and small blue balls; and Doug only accepts large green balls, small orange balls, and small purple balls.
- Try the preceding scenario, but add a small purple ball to Amy's rules.
- Try the first scenario, but change Doug's small purple rule to a large purple rule. This should return an error because now there's no one to accept the small purple balls.

And there you have it! Forty-five tests that exercise a great many of the options and permutations that the Super Ball Sorter offers. If you have read this far, you must enjoy testing as much as I do!

How did my test plan compare to yours? Did you think of things I didn't?

You may have noticed that while this test plan is complete, it's not that easy to read and it doesn't provide any way to keep track of test results. In the next chapter I'll talk about how to organize the test plan so that you can run it quickly and record results easily.

# Chapter 99: Organizing a Test Plan

In the preceding chapter we looked at a hypothetical software feature that sorts Super Balls among four children according to a set of rules. I came up with 45 different test cases, from simple to complicated, that would test various combinations of the rules.

But reading a chapter is not a very easy way to execute on a test plan! So in this chapter, I'll explain how I organize a test plan.

First, though, I'd like to tell you what I find to be less effective.

## **What I Don't Do**

**I don't write step-by-step instructions**, such as:

1. Navigate to the login page.
2. Enter the username into the username field.
3. etc. etc.

Unless you are writing a test plan for someone you have never met, whom you will never talk to, and who has never seen the application, this is unnecessary. While some level of instruction is important when your test plan is meant for other people, it's safe to assume you can provide some documentation about the feature elsewhere.

**I don't add screenshots to the instructions.**

While having screenshots in documentation is helpful, when they are in a test plan it just makes the plan larger and harder to read.

**I don't use a complicated test tracking system.**

In my experience, test tracking systems require more time to maintain than the time needed to actually run the tests. If there are regression tests that

need to be run periodically, they should be automated. Anything that can't be automated can be put in a one-page test plan for anyone to use when the need arises.

## What I Do

### I use a spreadsheet to organize my tests.

Spreadsheets are so wonderful because the table cells are already built in. They are easy to use, edit, and share. For test plans I will be running myself, I use an Excel spreadsheet. For plans I will be sharing with a team, I use a table in a Confluence page which we can all edit. You can see the test spreadsheet I created for our hypothetical Super Ball sorter [here](#); I include some screenshots from the spreadsheet throughout this chapter.

### I keep the instructions simple.

This screenshot shows the first two sections of my test:

| Test Case and Rules                       | State at End of Test Pass |                  |                  |                  |
|---|---------------------------|------------------|------------------|------------------|
| No Children have Any Rules                | Amy                       | Bob              | Carol            | Doug             |
| No rules                                  | 1/4 of the balls          | 1/4 of the balls | 1/4 of the balls | 1/4 of the balls |
| Size Rules Only                           | Amy                       | Bob              | Carol            | Doug             |
| Amy- Large balls only                     | Large balls only          | any balls        | any balls        | any balls        |
| Bob- Small balls only                     | any balls                 | Small balls only | any balls        | any balls        |
| Amy and Bob- Large balls only             | Large balls only          | Large balls only | any balls        | any balls        |
| Carol- Large; Doug- Small                 | any balls                 | any balls        | Large balls only | Small balls only |
| Amy, Bob, and Carol- Large                | Large balls only          | Large balls only | Large balls only | any balls        |
| Amy and Bob- Small; Carol- Large          | Small balls only          | Small balls only | Large balls only | any balls        |
| Amy and Bob- Large; Carol and Doug- Small | Large balls only          | Large balls only | Small balls only | Small balls only |
| Amy, Bob, Carol, and Doug- Large          | Should return an error    |                  |                  |                  |

In the second test case, I called the test “Amy- Large balls only”. This is enough for me to know that I’m setting a rule for Amy that she should accept large balls only. I don’t need to write “Create a rule for Amy that says she should accept large balls only, and then run the ball distribution to pass out the balls.” All of that is assumed from the feature description.

Similarly, I created a grouping of four columns called “State at End of

Test Pass". There is a column for each child, and in each cell I included what the expected result should be for that particular test case. For example, in the third test case, I set Amy, Carol, and Doug to "any balls" and Bob to "Small balls only". This means Amy, Carol, and Doug can have any kind of ball at all at the end of the test pass, and Bob should have only small balls. I don't need to write "Verify that all of Bob's Super Balls are small." "Small balls only" is enough to convey this.

### I use headers to make the test readable.

Because this test plan has 45 test cases, I need to scroll through it to see all the tests. Therefore, I make sure every section has good headers so that I don't have to remember what the header values are at the top of the page.

|   |                        |                    |                        |                   |
|---|------------------------|--------------------|------------------------|-------------------|
| Amy- R; Bob- R and O; Carol- R, O, and Y      | Red balls only         | R and O balls only | R, O, and Y balls only | any balls         |
| Amy, Bob, Carol, and Doug- Purple balls only  | Should return an error |                    |                        |                   |
| Amy- R, Y; Bob- O; Carol- Y; Doug- Y, G, B, P | R and Y                | Orange balls only  | Yellow balls only      | Y, G, B, and P    |
| <b>Size and Color Rules</b>                   | <b>Amy</b>             | <b>Bob</b>         | <b>Carol</b>           | <b>Doug</b>       |
| Doug- Large; Bob- Red                         | any balls              | Red balls only     | any balls              | Large balls only  |
| Doug- Large; Bob- Red; Carol- Small           | any balls              | Red balls only     | Small balls only       | Large balls only  |
| Doug- L; Bob- R; Carol- S; Amy- Y             | Yellow balls only      | Red balls only     | Small balls only       | Large balls only  |
| Amy and Doug- L; Bob- S; Carol- Purple        | Large balls only       | Small balls only   | Purple balls only      | Large balls only  |
| Amy and Doug- L; Bob and Carol- Purple        | Large balls only       | Purple balls only  | Large balls only       | Purple balls only |

In the preceding figure, you can see the end of one test section and the beginning of the final test section, "Size and Color Rules". I put in the headers for Amy, Bob, Carol, and Doug so that I don't have to scroll back up to the top to see which column is which.

### I keep the chart cells small to keep the test readable.

As you can see in the following figure, as the test cases become more complex, I added abbreviations so that they don't take up too much space:

|  |                        |            |             |            |
|--|------------------------|------------|-------------|------------|
| Amy- Large Red; Bob- Small Blue                            | Large Red              | Small Blue | any balls   | any balls  |
| Amy- LR; Bob- SB; Carol- LG                                | Large Red              | Small Blue | Large Green | any balls  |
| Amy- LR; Bob- SB; Carol- LG; Doug- SY                      | Should return an error |            |             |            |
| A- LR, LO, LY; B- SR, SO, SY; C- LG, LB, LP; D- SG, SB, SP | LR, LO, LY             | SR, SO, SY | LG, LB, LP  | SG, SB, SP |

After running several tests, it's pretty easy to remember that "A" equals "Amy" and "LR" equals "Large Red".

## I use colors to indicate a passing or failing test.

The great thing about a simple test plan is that it's easy to use it as a report for others. A completed test plan like this will be easy for anyone to read. Here's an example of what the first section of the test might look like when it's completed, if all the tests pass:

| Test Case and Rules                       | State at End of Test Pass |                  |                        |                  |
|---|---------------------------|------------------|------------------------|------------------|
|   | Amy                       | Bob              | Carol                  | Doug             |
| No Children have Any Rules                |                           |                  |                        |                  |
| No rules                                  | 1/4 of the balls          | 1/4 of the balls | 1/4 of the balls       | 1/4 of the balls |
| Size Rules Only                           | Amy                       | Bob              | Carol                  | Doug             |
| Amy- Large balls only                     | Large balls only          | any balls        | any balls              | any balls        |
| Bob- Small balls only                     | any balls                 | Small balls only | any balls              | any balls        |
| Amy and Bob- Large balls only             | Large balls only          | Large balls only | any balls              | any balls        |
| Carol- Large; Doug- Small                 | any balls                 | any balls        | Large balls only       | Small balls only |
| Amy, Bob, and Carol- Large                | Large balls only          | Large balls only | Large balls only       | any balls        |
| Amy and Bob- Small; Carol- Large          | Small balls only          | Small balls only | Large balls only       | any balls        |
| Amy and Bob- Large; Carol and Doug- Small | Large balls only          | Large balls only | Small balls only       | Small balls only |
| Amy, Bob, Carol, and Doug- Large          |                           |                  | Should return an error |                  |

If a test fails, it's marked in red. If there are any extra details that need to be added, I don't put them in the cell, making the chart hard to read; instead, I add notes on the side that others can read if they want more detail.

## I use tabs for different environments or scenarios.

The great thing about spreadsheets is that they offer the use of tabs. If you are testing a feature in your QA, staging, and production environments, you can have a tab for each environment and copy and paste the test plan into each one. Or you can use the tabs for different scenarios. In the case of our Super Ball Sorter test plan, we might want to have a tab for testing with a test run of 20 Super Balls, one for 100 Super Balls, and one for 500 Super Balls.

Test plans should be easy to read, follow, and use when documenting results. You don't need fancy test tools to create them; all you need are a simple spreadsheet, an organized mindset, and an ability to simplify instructions.

# Chapter 100: The Positive Outcomes of Negative Testing

As software testers and automation engineers, we often think about the happy path: the path the user will most likely take when they are using our application. When we write our automated UI tests we want to make sure we are automating those happy paths, and when we write API automation we want to verify that every endpoint returns a “200 OK” or similar successful response.

But it’s important to think about negative testing in both our manual and automated tests. Here are a few reasons why.

## **Our Automated Tests Might Be Passing for the Wrong Reasons**

When I first started writing automated UI tests in JavaScript, I didn’t understand the concept of the promise. I just assumed that when I made a request to locate an element, it wouldn’t return that element until it was actually located. I was so excited when my tests started coming back with the green “Passed” result, until a co-worker suggested I try to make the test fail by asserting on a different value. It passed again, because it was actually validating against the promise that existed, which was always returning “True”. That taught me a valuable lesson: never assume that your automated tests are working correctly just because they are passing. Be sure to run some scenarios where your tests should fail, and make sure they do so. This way, you can be sure you are really testing what you think you are testing.

## **Negative Testing Can Expose Improperly Handled Errors That Could Impact a User**

In API testing, any client-related error should result in a 400-level response rather than a 500-level server error. If you are doing negative testing and you discover that a 403 response is now coming back as a 500, this could mean the code is no longer handling that use case properly. A 500 response from the server could keep the user from getting the appropriate information

they need for fixing their error, or worse, it could crash the application.

## **Negative Testing Can Find Security Holes**

Just as important as making sure a user can log in to an application is making sure a user can't log in to an application when they aren't supposed to. If you only run a login test with a valid username and password, you are missing this crucial area! I have seen a situation where a user could log in with anything as the password, a situation where a user could log in with a blank password, and a situation where a user could log in with an incorrect username and password.

It's also crucial to verify that certain users don't have access to parts of an application. Having a carefully tested and functional Admin page won't mean much if it turns out that any random user can get to it.

## **Negative Testing Keeps Your Database Clean**

As I mentioned in Chapter 12, having good, valid data in your database will help keep your application healthy. Data that doesn't conform to expectations can cause web pages to crash or fail to load, or cause information to be displayed incorrectly. The more negative testing you can do on your inputs, the more you can ensure that you will only have good data.

For every input field I am responsible for testing, I like to know exactly which characters are allowed. Then I can run a whole host of negative tests to make sure entries with the forbidden characters are refused.

## **Sometimes Users Take the Negative Path**

It is so easy, especially with a new feature that is being rushed to meet a deadline, to forget to test the user paths where they will click the Cancel or Delete button. But users do this all the time; just think about times when you have thought about making an online purchase and then changed your mind and removed an item from your cart. Imagine your frustration if you weren't able to remove something from your cart, or if a Cancel button didn't clear a form to allow you to start again. User experience in this area is just as crucial as the happy path.

Software testing is about looking for unexpected behaviors so that we find them before a user does. When negative testing is combined with happy path testing, we can ensure that our users will have no unpleasant surprises.

# Chapter 101: What to Put in a Smoke Test

The term “smoke test” is usually used to describe a suite of basic tests that verify that all the major features of an application are working. Some use the smoke test to determine whether a build is stable and ready for further testing. I usually use a smoke test as the final check in a deployment to production. In this chapter, I’ll share a cautionary tale about what can happen if you don’t have a smoke test. Then I’ll continue that tale and talk about how smoke tests can go wrong.

Early in my testing career, I worked for a company that had a large suite of manual regression tests, but no smoke test. Each software release was difficult because it was impossible to run all the regression tests in a timely fashion. With each release, we picked which tests we thought would be most relevant to the software changes and executed those tests.

One day, in between releases, we heard that there had been a customer complaint that our Global Search feature wasn’t working. We investigated and found that the customer was correct. We investigated further and discovered that the feature hadn’t worked in weeks, and none of us had noticed. This was quite embarrassing for our testing team!

To make sure this kind of embarrassment never happened again, one of our senior test engineers created a smoke test to run whenever there was a release to production. It included all the major features and could be run fairly quickly. We felt a lot better about our releases after that.

However, the tester who created the test kept adding test steps to the smoke test. Every time a new feature was created, a step was added to the test. If we found a new bug in a feature, even if it was a small one, a step checking for the bug was added to the test. As the months went by, the smoke test took longer and longer to execute and became increasingly complicated. Eventually the smoke test itself took so much time that we didn’t have time to run our other regression tests.

Clearly there needs to be a happy medium between having no smoke test at all and having one that takes so long to run that it's no longer a smoke test. To decide what goes into a smoke test, I suggest asking these three questions.

### **Question #1: What would absolutely embarrass us if it were broken in this application?**

Let's use an example of an e-commerce website to consider this question. For this type of website, it would be embarrassing or even catastrophic if a customer couldn't:

- Log in to their account
- Search for an item they were looking for
- Add an item to their cart
- Edit their contact information
- Purchase the item in their cart

So, at the very least, a smoke test for this site should include a test for each of these features.

### **Question #2. Is this a main feature of the application?**

Examples of features in an e-commerce website that would be main features, but less crucial ones, might be:

- Wish list functionality
- Product reviews
- Recommendations for the user

If these features were broken, it wouldn't be catastrophic, but they are features that customers expect. So a test for each one should be added.

### **Question #3. If there were a bug here, would it stop the application from functioning?**

No one wants to have bugs in their application! But some bugs are more important than others. If the e-commerce website had an issue where the Add to Cart button was off-center, it might look funny but it wouldn't stop

customers from shopping.

However, a bug that wouldn't allow a customer to remove an item from their cart might keep them from purchasing the items they do want, which would affect sales. So a test to check that items can be removed from a cart would be important in a smoke test.

With these questions in mind, here is an example of a smoke test that could be created for an e-commerce site:

1. Log in.
2. Verify that product recommendations are present.
3. Do a search for a product.
4. Read a review of a product.
5. Add an item to the cart.
6. Add a second item to the cart and then delete it.
7. Edit customer information.
8. Make a purchase.
9. Write a review.

A smoke test like this wouldn't take very long to execute manually, and it would be easy to automate.

Whenever new features are added to the application, you should ask yourself Questions #1 and #2 to determine whether a test for the feature should be added to the smoke test. And whenever a bug is found in the product, you should ask yourself Question #3 to determine whether a test for that issue should be added to the smoke test.

Because we want our applications to be of high quality, it's easy to fall into the trap of wanting to test everything all the time. But this can create a test burden that keeps us so busy that we don't have time for anything else. Creating a simple, reliable smoke test can free us up for other activities, such as doing exploratory testing on new features or creating nightly automated tests.

# Chapter 102: What to Test When There's Not Enough Time to Test

In today's Agile world, we often don't have as much time as we feel we need to fully test our software's features. Gone are the days when testers had weeks or months to test an upcoming release. Because software projects usually take longer than estimated, we may be asked to test things at the last minute, just a day or two before the release. In this chapter I'll discuss what to test when there's not enough time to test, and I'll suggest some tips to avoid this problem in the first place.

## **The Bare Minimum: What to Test When There's Almost No Time**

Let's use our hypothetical Super Ball Sorter as an example. As you recall from Chapter 97, the feature takes a number of Super Balls and sorts them among four children using a set of defined rules. What would I do if I were asked to test this feature for the first time and it was due to be released tomorrow?

### **Step #1: Test the most basic case.**

The first thing I would do is test the most basic use case of the feature. In this case, it would be running the Super Ball Sorter with no rules at all. I would test this first because it would give me a very clear indication whether the feature was working at all. If it wasn't, I could raise the alarm right away, giving the developer more time to fix it.

### **Step #2. Test the most typical customer scenario.**

In the case of the Super Ball Sorter, let's say the product owner has told us that in the most typical scenario, two of the children will be assigned a rule, and the rule will be by size rather than color. So the next test I would run would be to assign one child a rule that they only accept large balls, and assign another child a rule that they only accept small balls. I would run the sorter with these rules and make sure the rules were respected.

### **Step #3. Run a basic negative test.**

We all know how frustrating it can be to make a mistake when we try to do an activity online, such as filling out a form, and we have an error on the page but we haven't been given any clear message about what it is. So the next thing I would test would be to make a common mistake that a user would make and ensure that I got an appropriate error message. For the Super Ball Sorter, I would set four rules that resulted in some balls not being able to be sorted, and I would verify that I got an error message that told me this was the case.

### **Step #4. Test with different users or accounts.**

Just because something is working correctly for one user or account doesn't mean it's going to work correctly for everyone! Developers sometimes check their work with only one test user if they are in a big hurry to deliver their feature. So I would make sure to run the Super Ball Sorter with at least two users, and I would make sure those users were different from the one the developer used.

After running these four tests, I would be able to say with some certainty that:

- The feature works at its most basic level.
- A typical customer scenario will work correctly.
- The customer will be notified if there is an error.

### **Remember That It Will Never Be Perfect, and Things Will Never Be Completely Done**

When software is released monthly, weekly, or even daily, there's no way to test everything you want to test. Even if you could get to everything, there will always be some sneaky bug that slips through. This is just a fact of life in software development. The good news is that because software is released so frequently, a bug fix can be released very shortly after the bug is found. So relax, and don't expect things to be perfect.

### **Speak Up—in Person and in Writing—if Disaster Is About to Strike**

Early in my testing career, I was on a team that was asked to test a large number of new features for a release in a short amount of time. When we were asked whether we felt confident in the new release, every one of us said no. We each delineated the things we hadn't been able to test yet and why we were concerned about the risks in those areas. Unfortunately, management went ahead and released the software anyway because a key customer was waiting for one of the features. As a result, the release was a failure and had to be recalled after many customer complaints.

If you believe that your upcoming software release is a huge mistake, speak up! Outline the things you haven't tested and some of the worst-case scenarios you can envision. Document what wasn't tested so that the key decision makers in your company can see where the risks are. If something goes wrong after the release, your documentation can serve as evidence that you had concerns.

### **Enlist the Help of Developers and Others in Your Testing**

While testers possess a valuable set of skills that help them find bugs quickly, remember that all kinds of other people can run through simple test scenarios. If everyone on your team understands that you have not been given ample time in which to test, they will be happy to help you out. If I were asking my teammates to test the Super Ball Sorter, I might ask one person to test scenarios with just one rule, one person to test scenarios with three rules, and one person to test scenarios with four rules, while I continued to test scenarios with two rules. In this way, we could test four times as many scenarios as I could test by myself.

### **Talk with Your Team to Find Out How You Can Start Testing Earlier**

To prevent last-minute testing, try to get involved with feature development sooner in the process. Attend meetings about how the feature will work, and ask questions about integration with other features and possible feature limitations. Start putting together a test plan before the feature is ready. Work with your developer to write some automated tests that they can use while in development. Ask your developer to commit and push some of their code so that you can test basic scenarios, with the

understanding that the feature isn't completely done. In the case of the Super Ball Sorter, I could ask the developer to push some code once the sorter was capable of sorting without any rules, just to verify that the balls were being passed to each child evenly.

### **Automate as Much as Possible**

In sprint-based development, there's often a lull for testers at the beginning of a sprint while the developers are still working on their assigned features. This is the perfect time to automate features that you have already tested. When release day looms, much or all of your regression testing can run automatically, freeing you up to do more exploratory testing on the new features.

As testers, we want our users to have a completely bug-free experience. Because of that, we always want more time for testing than we are given. With the aforementioned strategies, we can ensure that the most important things are tested and that with each sprint we are automating more tests, freeing up our valuable time.

# Chapter 103: How to Keep Your Test Cases From Slowing You Down

One of the first testing jobs I had was at a company that made software which could be used to create mobile applications. It was a very complex application and had so many features that it was often hard to keep track of them all. Shortly before I started working there, the company adopted a test tracking system to keep track of all the possible manual tests the team might want to run. This amounted to thousands of test cases.

Many of the test cases weren't written well, leaving those of us who were new to the team confused about how to execute them. The solution to this problem was to assign everyone the task of revising the tests as they were run. This helped a bit, but it slowed us down tremendously. Adding to the slowdown was the fact that every time we had a software release, our manager had to comb through all the tests and decide which ones should be run. Then there was the added confusion of deciding which mobile devices should be used for each test.

We were trying to transition to an Agile development style, but the number of test cases and the amount of overhead needed to select, run, and update the tests meant that we just couldn't adapt to the pace of Agile testing.

You might be thinking at this point, "Why didn't they automate their testing?" Keep in mind that this was back when mobile test automation was in its infancy. One member of our team had developed a prototype for an automated test framework, but we didn't have the resources to implement it, because we were so busy trying to keep up with our gigantic manual test case library.

Even when you have a robust set of automated tests in place, you'll still want to do some manual testing. Having a pair of eyes and hands on an application is a great way to discover odd behavior that you'll want to investigate further. But trying to maintain a vast library of manual tests is so time consuming that you may find you don't have time to do anything else!

In my opinion, the easiest and most efficient way to keep a record of what manual tests should be executed is through the use of simple spreadsheets. If I were to go back in time to that mobile app company, I would toss out the test case management system and set up some spreadsheets. I would have one smoke test spreadsheet, and one regression test spreadsheet for each major feature of the application. Each time a new feature was added, I'd create a test plan on a spreadsheet, and once the feature was released, I'd either add a few test cases to a regression test spreadsheet (if the feature was minor), or adapt my test plan into a new regression test spreadsheet for that feature.

This is probably a bit hard to imagine, so I'll illustrate it with an example. Let's say we have a mobile application called OrganizeIt! Its major features are a To-Do List and a Calendar. Currently the smoke test for the app looks like this:

| <b>Test</b>                        | <b>iOS phone</b> | <b>iOS tablet</b> | <b>Android phone</b> | <b>Android tablet</b> |
|------------------------------------|------------------|-------------------|----------------------|-----------------------|
| Log in with incorrect credentials  |                  |                   |                      |                       |
| Log in with correct credentials    |                  |                   |                      |                       |
| Add an event                       |                  |                   |                      |                       |
| Edit an event                      |                  |                   |                      |                       |
| Delete an event                    |                  |                   |                      |                       |
| Add a To-Do item                   |                  |                   |                      |                       |
| Edit a To-Do item                  |                  |                   |                      |                       |
| Complete a To-Do item              |                  |                   |                      |                       |
| Mark a complete item as incomplete |                  |                   |                      |                       |
| Delete a To-Do item                |                  |                   |                      |                       |
| Log out                            |                  |                   |                      |                       |

We also have a regression test for the major features: Login, Calendar, and To-Do List. Here's an example of what the regression test for the To-Do List might look like:

| <b>Test</b>   | <b>Expected result</b>                        |
|---|---|
| Add an item to the list with too many characters                  | Error message                                 |
| Add an item to the list with invalid characters                   | Error message                                 |
| Add a blank item to the list                                      | Error message                                 |
| Add an item to the list with a correct number of valid characters | Item is added                                 |
| Close and reopen the application                                  | Item still exists                             |
| Edit the item with too many characters                            | Error message, and original item still exists |
| Edit the item with invalid characters                             | Error message, and original item still exists |
| Edit the item so that it is blank                                 | Error message, and original item still exists |
| Mark an item as completed   | Item appears checked off                      |
| Close and reopen the application                                  | Item still appears checked off                |
| Mark a completed item as completed again                          | No change                                     |
| Mark a completed item as incomplete                               | Item appears unchecked                        |
| Mark an incomplete item as incomplete again                       | No change                                     |
| Close and reopen the application                                  | Item still appears unchecked                  |
| Delete the item   | Item disappears                               |
| Close and reopen the application                                  | Item is still gone                            |

This test would also be run on a variety of devices, but I've left that off the chart to make it more readable on the page.

Now let's imagine that our developers have created a new feature for the To-Do List, which is that items on the list can now be marked as Important, and Important items will move to the top of the list. In the interest of simplicity, let's not worry about the order of the items other than the fact that the Important items will be on the top of the list. We'll want to create a test plan for that feature, and it might look like this:

| Test   | Expected result   |
|--|---|
| Item at the top of the list is marked Important              | Item is now in bold, and remains at the top of the list                         |
| Close and reopen the application                             | Item is still in bold and at the top of the list                                |
| Item at the middle of the list is marked Important           | Item is now in bold, and moves to the top of the list                           |
| Item at the bottom of the list is marked Important           | Item is now in bold, and moves to the top of the list                           |
| Close and reopen the application                             | All Important items are still in bold and at the top of the list                |
| Every item in the list is marked Important                   | All items are in bold   |
| Close and reopen the application                             | All items are still in bold   |
| Item at the top of the list is marked as normal              | Item returns to plain text, and moves below the Important items                 |
| Close and reopen the application                             | Item is still in plain text, and below the Important items                      |
| Item in the middle of the Important list is marked as normal | Item returns to plain text, and moves below the Important items                 |
| Item at the bottom of the Important list is marked as normal | Item returns to plain text, and is below the Important items                    |
| Close and reopen the application                             | All Important items are still in bold, and normal items are still in plain text |
| Delete an Important item                                     | Item is deleted   |
| Close and reopen the application                             | Item is still gone  |
| Add an item and mark it as Important                         | Item is added as Important, and is added to the top of the list                 |

We would again test this on a variety of devices, but I've left that off the chart to save space.

Once the feature is released, we won't need to test it as extensively, unless there's a change to the feature. So we can add a few test cases to our To-Do List regression test, like this:

| Test  | Expected result  |
|---|--|
| Add an item to the list with too many characters                  | Error message  |
| Add an item to the list with invalid characters                   | Error message  |
| Add a blank item to the list                                      | Error message  |
| Add an item to the list with a correct number of valid characters | Item is added  |
| Close and reopen the application                                  | Item still exists  |
| <b>Add an Important item to the list</b>                          | <b>Item is in bold, and is added to the top of the list</b>              |
| Edit the item with too many characters                            | Error message, and original item still exists                            |
| Edit the item with invalid characters                             | Error message, and original item still exists                            |
| Edit the item so that it is blank                                 | Error message, and original item still exists                            |
| <b>Mark an Important item as normal</b>                           | <b>Item returns to plain text and is moved to the bottom of the list</b> |
| Mark an item as completed   | Item appears checked off   |
| <b>Mark an Important item as completed</b>                        | <b>Item remains in bold and appears checked off</b>                      |
| Close and reopen the application                                  | Item still appears checked off   |
| Mark a completed item as completed again                          | No change  |
| Mark a completed item as incomplete                               | Item appears unchecked   |
| Mark an incomplete item as incomplete again                       | No change  |
| Close and reopen the application                                  | Item still appears unchecked   |
| Delete the item   | Item disappears  |
| Close and reopen the application                                  | Item is still gone   |
| <b>Delete an Important item</b>                                   | <b>Item disappears</b>   |

The new test cases are marked in blue, but they wouldn't be blue in the actual test plan.

Finally, we'd want to add one test to the smoke test to check for this new functionality:

| Test                                | iOS phone | iOS tablet | Android phone | Android tablet |
|-------------------------------------|-----------|------------|---------------|----------------|
| Log in with incorrect credentials   |           |            |               |                |
| Log in with correct credentials     |           |            |               |                |
| Add an event                        |           |            |               |                |
| Edit an event                       |           |            |               |                |
| Delete an event                     |           |            |               |                |
| Add a To-Do item                    |           |            |               |                |
| <b>Add an Important To-Do item</b>  |           |            |               |                |
| Edit a To-Do item                   |           |            |               |                |
| Complete a To-Do item               |           |            |               |                |
| Mark a completed item as incomplete |           |            |               |                |
| Delete a To-Do item                 |           |            |               |                |
| Log out                             |           |            |               |                |

With spreadsheets like these, you can see how it is easy to keep track of a huge number of tests in a small amount of space. Adding or removing tests is also easy because it's just a matter of adding or removing a line in the table.

Spreadsheets like these can be shared among a team. Each time a smoke or regression test needs to be run, the test can be copied and named with a new date or release number (e.g., “1.5 Release” or “September 2023”), and the individual tests can be divided among the test team. For example, each team member could do a complete test pass with a different mobile device. Passing tests can be marked with a check mark or filled in green, and failing tests can be marked with an X or filled in red.

And there you have it! An easy-to-read, easy-to-maintain manual test case management system. Instead of taking hours of time maintaining test cases, you can use your time to automate most of your tests, freeing up even more time for manual exploratory testing.

# Chapter 104: Confused? Simplify!

As testers, we are often asked to test complex systems. Gone are the days when testers were simply asked to fill out form fields and click the Submit button; now we are testing data stores, cloud servers, messaging services, and much more. When so many building blocks are used in our software, it can become easy to get overwhelmed and confused. When this happens, it's best to simplify what we are testing until our situation becomes clear.

Here's an example that happened at my company. We were testing that push notifications of a specific type were working on an iPhone. One of my teammates was triggering a push notification, but it wasn't appearing on the phone. What could be wrong? Maybe notifications were completely broken. Maybe they were broken on the iPhone. Maybe only this specific notification was broken. Maybe only notifications of this type were broken. We had a lot of notifications to test and we were working on a deadline, but we had to figure out why this particular notification wasn't working.

So we simplified the process by asking a series of questions and running a test for each one.

We started with:

Is this push notification working on an Android phone?

We triggered the same notification to go to an Android phone, and the push was delivered. So we ruled out that the notification itself was broken.

Next, we asked:

Is this push notification working on any other iPhone?

We triggered the same notification to go to a different iPhone, and the push was delivered. So we ruled out that the notification was broken on iOS devices.

Then we asked:

Is any notification working on this specific iPhone?

We triggered some different notifications to go to the iPhone, and no pushes were delivered. So we concluded that the problem was not with the notification or the push service; the problem was with the phone. In taking a step back and asking three simple questions, we were able to quickly diagnose the problem.

Let's take a look at another example, using the Super Ball Sorter. In this testing scenario, we are sorting the balls by both size and color. We have the children set up with the following rules:

- Amy gets only large balls.
- Bob gets only small purple balls and large red balls.
- Carol gets only small balls.
- Doug gets only green balls.

When we run the sorter, a small purple ball is next in the sorting process, and it's Bob's turn to get a ball. We are expecting that Bob is going to get the small purple ball because his sorting rules allow it, but he doesn't get the ball. It goes to Carol instead.

What could be wrong here? Maybe Bob isn't getting any balls. Maybe the purple ball isn't being sorted at all. Maybe only the small balls aren't being sorted.

To figure out what is going on, we ask the following questions:

1. Can Bob get any sorted balls?

We set up the sorter so that Amy, Carol, and Doug only get large balls and Bob only gets small balls. We run the sorter and Bob gets all the small balls. So we know Bob can get balls.

2. Can anyone get the small purple ball?

We set up the sorter so that Amy only gets small purple balls, and Bob, Carol, and Doug get any ball at all. We set up our list of balls so that the small purple ball is first on the list. When we start our sorting process with Amy, she gets the small purple ball. So now we know the

small purple ball isn't the problem.

### 3. Can Bob get the small purple ball in some other scenario?

We saw in our initial test that Bob wasn't getting the small purple ball, but can he ever get that ball? We set up our rules so that Amy only gets large balls and Bob only gets small purple balls. We don't give Carol and Doug any rules. Then we set up our list of balls so that the small purple ball is first on the list. Amy doesn't get the small purple ball, because she only gets large balls, so the small purple ball is offered to Bob. He gets the ball, so now we know Bob can get the small purple ball in some scenarios.

At this point, we know the problem is not with the small purple ball. What is different between the original scenario and the one we just ran?

One difference is that in the original scenario, all four children had a rule. So we ask this question:

### 4. Can Bob get the small purple ball when it's his only rule, and the other children all have rules?

We set up the rules like this:

- Amy gets only large balls.
- Bob gets only small purple balls.
- Carol gets only small balls.
- Doug gets only green balls.

We again set up our list of balls so that the small purple ball is first on the list. The ball skips Amy because it doesn't meet her rule, and Bob gets the ball. So now we know the problem is not that all the children have rules.

The next logical question is:

### 5. What happens when Bob has two rules?

We set up the rules like this:

- Amy gets only large balls.
- Bob gets only small purple balls and small yellow balls.
- Carol gets only small balls.
- Doug gets only green balls.

Our list of balls is the same, in that the small purple ball is first. This time, the ball skips Amy and Bob, and Carol gets the small purple ball.

Aha! Now we have a good working theory: when Bob has two rules, the sorting is not working correctly. We can test out this theory by giving another child two rules, while giving everyone else one rule. Are the balls sorted correctly? What about when a child has two rules that specify color only and not size? Will the two rules work then? By continuing to ask questions, we can pinpoint precisely what the bug is.

By making your tests as simple as possible and proceeding with your tests in a methodical and logical manner, you can find bugs as quickly as possible, even in a very complex system.

# Chapter 105: Six Steps to Writing an Effective Bug Report

As testers, we know how important it is to test our software thoroughly and document our findings meticulously. But all of our talent will be useless if we can't effectively communicate our test results to others! If your test results are written in a giant, poorly organized spreadsheet with tiny text and lots of unnecessary details, even the most dedicated test manager will find their eyes glazing over with boredom when they look at it. In this chapter, I'll describe six steps to take to ensure that you can communicate your findings to others efficiently and effectively.

## **Step #1: Determine What Goal You Are Trying to Accomplish with Your Report**

Why are you creating your report? What do you want people to understand when they read it? You might be creating a report to demonstrate that you have tested all the acceptance criteria in the user story. You could be showing your manager exactly which areas of the application are not functioning properly. You could be demonstrating that after a recent code change, several bugs were fixed.

There are all kinds of reasons to create a report, but if you don't stop and think about *why* you are creating it, it's probably not going to be very clear to your readers. Simply sending over your test notes is not enough to communicate effectively.

## **Step #2: Focus on the Reader's Needs**

Who will be reading your report? Will it be a developer, your test manager, your product owner, your team lead, or the CTO of your company? You will want to tailor your test report for your audience. The CTO is probably very busy and will not care how many permutations of your form data you ran or how you developed the permutations. They will want to see that you ran 100 tests and that 99 passed. Your product owner will want to see that you have tested the user stories and that the outcomes were as

expected. Your test manager might be interested in how many different permutations of the test you ran, and what logic you used to create them. Your developer might want to know what test data you used and what your server response times were.

You can see how the interests of your reader will vary quite a bit depending on their role, so think about how you can best present the information that they want.

For Steps 3 through 6, we'll once again use the example of the Super Ball Sorter.

### **Step #3: Avoid Extraneous Information**

Make sure your report contains only the information your reader needs. Extraneous information means a reader has to sift through your report to find the important results. Consider this report, which shows the results of two tests in which two children have a sorting rule:

| Rules   | Amy   | Bob   | Carol   | Doug   |
|---|---|---|---|--|
| Amy: small blue balls;<br>Doug: large green balls | Small blue ball;<br>small blue ball; small blue ball; small blue ball     | Large red ball; small orange ball; large yellow ball; small purple ball | Large purple ball;<br>small green ball;<br>large yellow ball;<br>small red ball     | Large green ball; large green ball;<br>large green ball; large green ball  |
| Bob: large red balls; Carol: small yellow balls   | Large orange ball; small purple ball; large yellow ball; small green ball | Large red ball; large red ball; large red ball; large red ball          | Small yellow ball;<br>small yellow ball;<br>small yellow ball;<br>small yellow ball | Small blue ball; small green ball;<br>large purple ball; small orange ball |

Did the tests pass? How long did it take for you to determine that by looking at the test report? There is a lot of information here that is unnecessary. It doesn't matter that Bob got the large red ball first and the small orange ball second. What matters is that Amy only got small blue balls and Doug only got large green balls. Contrast that test report with this one:

| Rules   | Rules respected? |
|---|------------------|
| Amy: small blue balls; Doug: large green balls  | Yes              |
| Bob: large red balls; Carol: small yellow balls | Yes              |

Here you can very quickly see what rules were set and whether those rules were respected. If a reader of your report needs to know what balls Bob got, they can ask you for those details and you can look them up in your notes.

#### Step #4: Make the Report Visually Immediate

We are all busy people; developing and testing software is fast paced and time consuming! Your manager or CTO probably gets dozens of reports and emails a day, so make your report so easy to read that just a glance at it will give them the information they need. Take a look at this test report:

| Test Case                       | Result   |
|---------------------------------|--|
| None of the children have rules | The balls are sorted evenly among the children |
| One child has a rule            | The child's rule is respected                  |
| Two children have rules         | The two children's rules are respected         |
| Three children have rules       | The three children's rules are respected       |
| Four children have rules        | None of the balls are sorted                   |

How many seconds does it take you to see that a test failed? A reader needs to read the entire report to see that the fourth test failed.

Compare that with this test report, which shows the same tests and the same results:

| Number of Rules | Pass/Fail |
|-----------------|-----------|
| 0               | Green     |
| 1               | Green     |
| 2               | Green     |
| 3               | Green     |
| 4               | Red       |

This report is pretty immediate! It's also really easy to see how many rules were used when the test failed.

### Step #5: Make the Report Easy to Read

In addition to being visually immediate, the report needs to be easy to read. Take a look at this example, in which two tests are run and three children are given rules:

| Rules   | Rules respected?  |
|---|---|
| Amy: small blue balls; Bob: large blue balls; Carol: small purple balls   | Amy gets only small blue balls and Bob gets only large blue balls, but Carol gets balls other than the small purple balls |
| Amy: large blue balls; Bob: small purple balls; Carol: small yellow balls | Amy gets only large blue balls, Bob gets only small purple balls, and Carol gets only small yellow balls                  |

This report shows very quickly that one of the tests failed, but in order to see why the test failed, it's necessary to read the whole description to see that Carol's rule was not respected.

This report conveys exactly the same information:

| Rules  | Amy  | Bob  | Carol |
|--|------|------|-------|
| <b>Amy: small blue balls;<br/>Bob: large blue balls;<br/>Carol: small purple balls</b>   | PASS | PASS | FAIL  |
| <b>Amy: large blue balls;<br/>Bob: small purple balls;<br/>Carol: small yellow balls</b> | PASS | PASS | PASS  |

With this report, it is easy to see that Carol's rule was not respected and, by simply looking in the leftmost column, to see what that rule was.

#### **Step #6: Make the Report Readable Without Any Additional Explanation**

How long does it take for you to figure out what this report means?

| Rules                | Result             |
|----------------------|--------------------|
| A-SB; B-LO; C-L; D-S | A-Y; B-Y; C-Y; D-N |
| A-L; B-S; C-Y; D-P   | A-Y; B-N; C-Y; D-Y |
| A-LY; B-L; C-S; D-SG | A-Y; B-Y; C-N; D-Y |

It's fine to use all sorts of abbreviations when you are testing and taking notes for yourself, but your reader shouldn't need a key to interpret it. Who but the most interested testers are going to take the time to see where the bug is here?

This report conveys exactly the same information:

|                 |                         |                         |                     |                         |
|-----------------|-------------------------|-------------------------|---------------------|-------------------------|
| Test 1          | Amy: small blue balls   | Bob: large orange balls | Carol: large balls  | Doug: small balls       |
| Rule respected? | Yes                     | Yes                     | Yes                 | No                      |
| Test 2          | Amy: large balls        | Bob: small balls        | Carol: yellow balls | Doug: purple balls      |
| Rule respected? | Yes                     | No                      | Yes                 | Yes                     |
| Test 3          | Amy: large yellow balls | Bob: large balls        | Carol: small balls  | Doug: small green balls |
| Rule respected? | Yes                     | Yes                     | No                  | Yes                     |

It's easy to see exactly what rules each child was given for each test. Through the use of color, the report demonstrates very clearly where the bug is: whenever a child is given a rule that they should get only small balls, that rule is not respected.

In today's fast-paced world, we all have vast amounts of information coming to us every day. If we are going to make a difference with our testing and influence decision-making where we work, we need to be able to convey our test results in ways that clearly show what is going on with our software and what should be done to improve it.

# Chapter 106: Should You Hunt for That Bug?

Anyone who has ever done laundry has likely discovered while folding their clothes that they are missing a sock. Sometimes the sock is missing because it never made it into the laundry basket. Sometimes the sock was left in the washing machine. There are even jokes about how the clothes dryer sends socks into another dimension!

What's interesting is the reaction that people have to the missing sock. Some people shrug their shoulders and figure that the sock will turn up eventually. Others will spend most of their day looking for the sock: they'll search through the laundry room, all of the undone laundry, their closet, under the bed, and so on.

This is a great metaphor for what testers do when they encounter a strange and hard-to-reproduce bug! Some testers decide that since the bug is hard to reproduce, they should move on and test something else. Other testers decide to devote every moment to finding the cause of the odd behavior, to the exclusion of all other testing. Which is the correct approach? It depends. In this chapter, I'll list three reasons why you might want to hunt for the elusive bug and three reasons why you might want to put off the hunt for later.

## Reasons to Hunt for the Bug

- **When the bug happens, it's a big deal.** You might be testing a system in which everything works just fine most of the time. But when the bug occurs the system crashes, or data is lost, or a customer can't submit an order. This is a serious problem. Even if the bug happens just 1% of the time, it's important to figure out what's going on because you will lose users as a result of this issue.
- **The problem is intermittent, but it happens frequently.** I was once plagued by a bug in some software I was using. When I was logged

in to the software, about 50% of the time the application would forget who I was. This was really annoying. If someone were to ask me about this software, do you think I would have recommended it? You don't want your users to give you poor reviews or stop using your application!

- **The problem hints at an important performance issue.** Perhaps your software works just great when you test it with one or two users in your test environment, but you're seeing strange behavior in your production environment. Don't just shrug this off with a "Works for me" statement! This bug could indicate that there is a problem with your application when it's under load. Perhaps there's a memory leak that gets worse the longer the application is used. Or maybe the calls to the server are taking too long, and the problem is compounded as more and more calls are made, locking the database. Whatever the reason, it's important to find out the root cause of the problem and fix it before your customers see it.

## Reasons to Save the Hunt for Later

- **You've been testing for weeks and you only saw the problem once.** We all know that software and hardware aren't perfect. Strange glitches can happen, including service interruptions in hosting environments, power surges on equipment, and the loss of electricity or internet connections. The bug you saw just once could have been caused by any one of these things. This is the kind of bug to watch for, but not to chase after. If it happens again, then you can start looking into it.
- **You're pretty sure you know the cause already.** If you think you saw a bug because a team member forgot to deploy one part of the application, someone forgot to turn on a toggle, or your test user was deleted, and the bug went away as soon as the problem was fixed, then there's probably no reason to hunt any further.
- **You are in a time-sensitive situation and you think the issue doesn't pose a risk.** It's very aggravating to have a giant bug show up in production and then hear from a tester, "Oh, I saw that bug, but

we were in a hurry and didn't have time to investigate." That said, if the bug is something obscure that you think a user will never do and you need to meet an important deadline, it might be OK to wait until after the release to dig in further.

Here's an example. A team is trying to release some new search functionality to their application. The team has been testing for weeks and it appears that things are working well. The day of the release, a tester discovers that if she enters a search term with 1,000 characters, there is an intermittent bug. Rather than calling off the release and spending hours looking for the bug, the team should probably go forward with the release and investigate later, because it's very unlikely that end users will be doing searches on 1,000-character terms.

### **Do You Have a "Missing Sock"?**

The next time you encounter a strange bug, ask yourself: is this something that should be investigated now, or can it wait until later? The size of the bug, the frequency of the bug, and the likelihood that it will be seen by users can help you decide how to proceed.

# Chapter 107: Why You Should Be Testing in Production

This is a true story; I'm keeping the details vague to protect those involved.

Once there was a software team that was implementing new functionality. They tested the new functionality in their QA environment and it worked just fine. So they scheduled a deployment: first to the staging environment, then to production. They didn't have any automated tests for the new feature, because it was tricky to automate. And they didn't bother to do any manual tests in staging or production, reasoning that if it worked in the QA environment, it must work everywhere.

You can probably guess what happened next: they started getting calls from customers that the new feature didn't work. They investigated and found that this was true. Then they tried the feature in the staging environment and found that it didn't work there either. As it turned out, the team had used hard-coded configuration strings that were only valid in the QA environment. If they had simply done one test in the staging or production environment, they would have noticed that something was wrong. Instead, it was left to the customers to notice the problem.

There are two main reasons why things that work in a test environment don't work in a production environment.

## Configuration Problems

This is what happened with the team I just described. Software is complicated, and there are often multiple servers and databases that need to talk to one another for the software to work properly. Keeping software secure means each part of the application must be protected through the use of passwords or other configuration strings. If any one of these strings is incorrect, the software won't work completely.

## Deployment Problems

In this age of microservices, deploying software usually means deploying several different APIs. In a large organization, different teams may be responsible for different APIs. Let's say that Team A is responsible for API A and Team B is responsible for API B. When a new feature in API A needs the new code in API B to work properly, API B will need to be deployed first. It's possible that Team B will forget to deploy API B or not even realize that it needs to be deployed. In cases like this, Team A might assume that API B had been deployed, and they will go ahead and deploy API A. Without testing, Team A will have no way of knowing that the new feature isn't working.

By running tests in every environment, you can quickly discover whether you have configuration or deployment problems. It's often not necessary to go through extensive testing of a new feature in production if you've already tested it in QA, but it is vital that you do at least some testing to verify that it's working! You never want to have your customers find problems before you do.

# Chapter 108: What to Do When There's a Bug in Production

A software tester can find it bone-chilling to realize a bug has been found in production! In this chapter, I'll walk you through a series of steps testers can take to handle production bugs and prevent them in the future.

## **Step #1: Remain Calm**

Because we are the ones who are testing the product and signing off on the release, it's easy to panic when a bug is found in production. We ask ourselves, "How did this happen?" and we can be tempted to thrash around looking for answers. But this is not productive. Our top priority should be to make sure the bug is fixed, and if we don't stay calm, we may not investigate the issue properly or test the fix properly.

## **Step #2: Reproduce the Issue**

If the issue came from a customer or from another person in your company, the first thing to do is to see whether you can reproduce it. It's possible that the issue is simply user error or a configuration problem. But don't jump to these conclusions too quickly! Make sure to follow any steps described by the user as carefully as you can, and wherever possible, make sure you are using the same software and hardware as the user: for example, use the same type of mobile device and the same build; or the same type of operating system and the same browser. If you are unable to reproduce the issue, ask the user for more information; then keep trying until you can reproduce it.

## **Step #3: Gather More Information**

Now that you have reproduced the issue, gather more information about it. Is the issue happening in your test environment as well? Is it present in the previous build, and if so, can you find a build where the issue is not present? Does it only happen with one specific operating system or browser? Do

certain configuration settings need to be in place to see the issue? The more information you can gather, the faster your developer will be able to fix the problem.

## **Step #4: Understand the Root Cause**

At this point, your developer will be working to figure out what is causing the bug. When they figure it out, make sure they tell you what the problem was and that you understand it. This will help you figure out how to test the fix and determine what regression tests should be done.

## **Step #5: Decide When to Fix the Issue**

When there's a bug in production, you will want to fix it immediately, but that is not always the best course of action. I'm sure you've encountered situations in which fixing one bug created new bugs. You will want to take some time to test any areas that might have been impacted by the fix.

When trying to decide when to fix a bug, think about these two things:

- How many users are affected by the issue?
- How severe is the issue?

You may have an issue that affects less than 1% of your users. But if the bug is so severe that these users can't use the application, you may want to fix it right away.

Alternatively, you may have an issue that affects all of your users, but it is so minor that their experience won't be impacted. For example, a misaligned button might not look nice, but it's not stopping your users from using the application. In this case, you might want to wait until your next scheduled release to fix the issue.

## **Step #6: Test the Fix**

When you test the bug fix, don't check it just once. Be sure to check the fix on all supported browsers and devices. Then run regression tests in any areas affected by the code change. If you followed Step #4, you'll know

which areas to test. Finally, do a quick smoke test to make sure no important functionality is broken.

## **Step #7: Analyze What Went Wrong**

It's tempting to breathe a big sigh of relief and move on to other things when a production bug is fixed. But it's very important to take the time to figure out exactly how the bug got into production in the first place. This is not about finger-pointing and blame; everybody makes mistakes, whether they are developers, testers, product owners, managers, or release engineers. This is about finding out what happened so that you can make changes to avoid the problem in the future.

Perhaps your developers made a code change and forgot to tell you about it, so it wasn't tested. Perhaps you tested a feature but forgot to test it in all browsers, and one browser had an issue. Maybe your product owner forgot to tell you about an important use case, so you left it out of your test plan.

Whatever the issue, be sure to communicate it clearly to your team. Don't be afraid to take responsibility for your role in causing the issue.

## **Step #8: Brainstorm Ways to Prevent Similar Issues in the Future**

Now that you know what went wrong, how can you prevent it from happening again? Discuss the issue with your team and see whether you can come up with some strategies.

You may need to change a process: for example, having your product owner sign off on any new features to make sure nothing is missing. Or you could make sure your developers let you know about any code refactoring so that you can run a regression test, even if they are sure they haven't changed anything.

You may need to change your strategy: you could have two testers on your team look at each feature so that it's less likely that something will be overlooked. Or you could create test plans which automatically include a step to test in every browser.

You may need to change both your process and your strategy! Whatever the situation, you can now view the bug that was found in production as a blessing, because it has resulted in you being a wiser tester and your team being stronger.

# Chapter 109: Fix All the Things

It's very tempting when you are rushing to complete features to let some bugs slide. This chapter will explain why, in most cases, it's better to fix all the bugs as soon as they are found rather than later. The following scenario is hypothetical, but it is based on my experience as a tester.

NewTech Inc. is very excited about its new email editor, which will enable customers to compose emails to their clients and schedule when they should be sent. NewTech's service reps will also be able to add or change their customers' company logos on the emails.

Because they are on a tight deadline, the developers are rushing to complete the work. The tester finds an issue with the logo-changing feature: the logo doesn't change unless the user logs out and back in again. The team discuss the issue and decide that because it's a feature that only NewTech employees can use, it's safe to put it on the backlog to be fixed at another time.

The app is released and customers begin using it. All of the customers would like to add their company logos to their emails, so they begin calling the NewTech service reps and asking for this service. The service reps add and save the logos, but they don't see the logos appear on the email config page. The dev team forgot to let them know about the bug, and the workaround of logging out and back in again.

Let's assume that each time a service rep encounters the issue and emails someone on the dev team about it, five minutes are wasted. If there are 10 service reps on the team, that's 50 wasted minutes.

**Total time wasted to date:** 50 minutes

But now everyone knows about the issue, so it won't be a problem anymore, right? Wrong! NewTech has hired two new testers, and neither one knows about the issue. They encounter it in their testing, and ask the original tester about it. "Oh, that's a known issue," he replies. "It's on the backlog."

Time wasted: 10 minutes for each new tester to investigate the problem, and 10 minutes for each new engineer to ask the first engineer about it.

**Total time wasted to date:** One hour and 10 minutes

Next, a couple of new service reps are hired. At some point, they each get a request from a company to change the company's logo. When they go to make the change, the logo doesn't update. They don't know what's going on, so they ask their fellow service reps. "Oh yeah, that's a bug," say the senior service reps. "You just need to log out and back in again." Time wasted: 10 minutes for each new service rep to wonder what's going on, and 10 more minutes in conversation with the senior service reps.

**Total time wasted to date:** One hour and 30 minutes

It's time to add new features to the application. NewTech decides to give its customers the option of adding a profile picture to their account. A new dev is tasked with adding this functionality. He sees that there's an existing method to add an image to the application, so he chooses to call that method to add the profile picture. He doesn't know about the logo bug. When one of the testers tests the new feature, she finds that profile picture images don't refresh unless she logs out and back in again. So she logs a new bug for the issue. Investigating the problem and reporting it takes 20 minutes.

**Total time wasted to date:** One hour and 50 minutes

The dev team meets and decides that since customers will see the issue, it's worth fixing. The dev who is assigned to fix the issue is a different developer from the one who wrote the image-adding method (who has since moved on to another company), so it takes her a while to become familiar with the code. Time spent fixing the issue: two hours.

**Total time wasted to date:** Three hours and 50 minutes

The dev who fixed the issue didn't realize the bug existed for the company logo as well, so she didn't mention it to the tester assigned to test her bug fix. The tester tests the bug fix and finds that it works correctly, so she closes the issue. Time spent testing the fix: 30 minutes.

**Total time wasted to date:** Four hours and 20 minutes

When it's time for the new feature to be released, the testing team does regression testing. They discover there is now a new issue on the email page: because of the fix for the profile images, now the email page refreshes when customers make edits, and the company logo disappears from the page. One of the testers logs a separate bug for this issue. Time spent investigating the problem and logging the issue: 30 minutes.

**Total time wasted to date:** Four hours and 50 minutes

The dev team realizes this issue will have a significant impact on customers, so the developer quickly starts working on a fix. Now she realizes the code she is working on affects both the profile page and the email page, so she spends time checking her fix on both pages. She advises the testing team to be sure to test both pages as well. Time spent fixing the problem: one hour. Time spent testing the fixes: one hour.

**Total time wasted to date:** Six hours and 50 minutes

How much time would it have taken for the original developer to fix the original issue? Let's say 30 minutes, because he was already working with the code.

How much time would it have taken to test the fix? Probably 30 minutes, because the tester was already testing that page and the code was not yet used elsewhere.

So, by fixing the original issue when it was found, NewTech would have saved nearly six hours in work that could have been spent on other things. This doesn't seem like a lot, but when considering the number of features in an application, it really adds up.

And this scenario doesn't account for lost productivity from interruptions. If a developer is fielding questions from the service reps all day about known issues that weren't fixed because they weren't customer-facing bugs, it's hard for her to stay focused on the coding she's doing.

The moral of the story is that unless you think that no user, internal or external, will ever encounter the issue, fix things when you find them!

# Chapter 110: The Hierarchy of Quality

When thinking about the different facets of software quality, I found myself wondering what I would do if I were brought on to a project that had never had any testing and I needed to start from scratch. Where would I begin my testing? I thought about what the most important needs are for quality, and I was reminded of Maslow's Hierarchy of Needs.

For those who are unfamiliar with this concept, this is a theory in psychology that all human beings must have certain basic needs met before they can grow as people. The needs are as follows:

1. Physiological: food, water, shelter
2. Safety: security, property, employment
3. Love and belonging: friendship, family
4. Esteem: respect, self-esteem
5. Self-actualization: becoming the best person one can be

Looking at this list, it's clear that physiological needs are the most important. After all, it doesn't matter whether you have high self-esteem if you have no water to drink. Each successive need builds on the more important one before it.

With this in mind, I realized there is a Hierarchy of Quality: certain conditions of quality that must be met before a team can move on to the next area of quality. The various facets of quality I mention here and in the next chapter were inspired by this blog post by Federico Toledo: <https://abstracta.us/blog/software-testing/software-testing-wheel>

Here is my perception of where the different areas of quality fall in the hierarchy, starting with the most important.

## **1. Functionality and Reliability**

These two areas share the most important spot. Functionality means the software does what it's supposed to do. This is critical, because without this, the application might as well not exist. Imagine a clock app that didn't tell time or a calculator that didn't add numbers.

Reliability means the software is available when it's needed. It doesn't really matter whether the app works if a user can't get to it when they need it.

Once these quality needs have been met, we can move on to the next level.

## **2. Security and Performance**

Security is important because users need to feel that their data is being protected. Even applications that don't have login information or don't save sensitive data still need to be protected from things like cross-site scripting, which might allow a malicious user to gain control of someone else's device.

Performance is also important, because no one wants to wait 60 seconds for a web page to load. If an application isn't responsive enough, users will go elsewhere.

Now that the application is secure and performant, we can go to the third level.

## **3. Usability**

This is the level where we make sure that as many users as possible have a good experience with the application. Usability means an application's workflows are intuitive so that users don't get confused. It also means the application is internationalized so that users around the world can use it, and accessible so that users with visual, auditory, or physical differences can use it as well.

Now that we've made our application accessible to as many users as possible, it's time to go on to the next level.

## **4. Maintainability**

This is a level of quality that benefits the software team. Maintainability refers to how easily an application can be updated. Is it possible to add new APIs or update existing ones? How easy is it to test the system? Is it easy to deploy new code? Is it easy for other teams to use the code? Is the code clear and easy to understand?

When software is accessible and easy to use for all end users, and is easy to work with and maintain for the development team, truly high quality has been achieved.

# Chapter 111: Measuring Quality

The concept of measuring quality can be a hot-button topic for many software testers. This is because metrics can be used poorly; we've all heard stories about testers who were evaluated based on how many bugs they found or how many automated tests they wrote. These measures have absolutely no bearing on software quality. A person who finds a bug in three different browsers can either write up the bug once or write up a bug for each browser; having three JIRA tickets instead of one makes no difference in terms of what the bug is! Similarly, writing 100 automated tests when only 30 are needed for adequate test coverage doesn't ensure quality and may actually slow down development time.

But measuring quality is important, and here's why: software testers are to software what the immune system is to the human body. When a person's immune system is working well, they don't think about it at all. They get exposed to all kinds of viruses and bacteria on a daily basis, and their immune system quietly neutralizes the threats. It's only when a threat gets past the immune system that a person's health breaks down, and then they pay attention to the system. Software testers have the same problem: when they are doing their job really well, there is no visible impact in the software. Key decision makers in the company may see the software and praise the developers who created it without thinking about all the testing that helped ensure that the software was of high quality.

Measuring quality is a key way that we can demonstrate the value of our contributions. But it's important to measure well; a metric such as "There were 100 customer support calls this month" means nothing because we don't have a baseline to compare it to. If the number of customer support calls went from 300 in the first month to 200 in the second month to 100 in the third month, and daily usage statistics stayed the same, then it's logical to conclude that customers are having fewer problems with the software.

Let's take a look at some ways we can measure various facets of quality.

## **Functionality**

*How many bugs are found in production by customers?*

A declining number could indicate that bugs are being caught by testers before going to production.

*How many daily active users do we have?*

A rising number probably indicates that customers are happy with the software, and that new customers have joined the ranks of users.

## **Reliability**

*What is our percentage of uptime?*

A rising number could show that the application has become more stable.

*How many errors do we see in our logs?*

A declining number might show that the software operations are generally completing successfully.

## **Security**

*How many issues were found by penetration tests and security scans?*

A declining number could show that the application is becoming more secure.

## **Performance**

*What is our average response time?*

A stable or declining number will show that the application is operating within accepted parameters.

## **Usability**

*What are our customers saying about our product?*

Metrics like survey responses or app store ratings can indicate how happy customers are with an application.

*How many customer support calls are we getting?*

A rising number of support calls from customers could indicate that it's

unclear how to operate the software.

## Maintainability

*How long does it take to deploy our software to production?*

If it is taking longer to deploy software than it did during the last few releases, then the process needs to be evaluated.

*How frequently can we deploy?*

If it is possible to deploy more frequently than was possible six months ago, then the process is becoming more streamlined.

There's no one way to measure quality, and not every facet of quality can be measured with a metric. But it's important for software testers to be able to use metrics to demonstrate how their work contributes to the health of their company's software, and the preceding examples are some ways to get started. Just remember to think critically about what you are measuring, and establish good baselines before drawing any conclusions.

# Chapter 112: Managing Test Data

It's never fun to start your workday and discover that some or all of your nightly automated tests failed. It's especially frustrating when you discover that the reason why your tests failed was because someone changed your test data.

Test data issues are a very common source of aggravation for software testers. Whether you are testing manually or running automation, if you think your data is set the way you want it and it has been changed, you will waste time trying to figure out why your test results aren't right.

Here are some of the common problems with test data:

- **Users overwrite one another's data:** I was on a team that had an API I'll call API 1. I wrote several automated tests for this API using a test user. One of the tests in API 1 set the user's email address. API 1 was moved to another team, and my team started working on API 2. I wrote several automated tests for API 2 as well. Unfortunately, I inadvertently used the same test user for API 2, and this test user needed to have a different email address for API 2 than it did for API 1. This meant that whenever automated tests were run on API 1, they changed the email address of the test user, and then my API 2 tests would fail.
- **The configuration is changed by another team:** When teams need to share a test environment, changes to the environment configuration made by one team can impact another team. This is especially common when using feature toggles. One team might have test automation set up with the assumption that a feature toggle will be on, but another team might have automation set up with the expectation that the feature toggle is off.
- **Data is deleted or changed by a database refresh:** Companies that use sensitive data often need to periodically scramble or overwrite

that data to make sure no one is testing with real customer information. When this happens, test users that have been set up for automation or manual testing can be renamed, changed, or deleted, causing tests to fail.

- **Data becomes stale:** Sometimes data that is valid at one point in time becomes invalid as time passes. A great example of this is a calendar date. If an automated test needs a date in the future, the test writer might choose a date a year or two from now. Unfortunately, in a year or two, that future date will become a past date, and then the test will fail.

What can we do about these problems? Here are some suggestions:

- **Use containers:** Using containers like those created with Docker means you have complete control over your test environment, including your application configuration and your database. To run your tests, you spin up a container, run the tests, and destroy the container when the tests have completed.
- **Create a fresh database for testing:** It's possible to create a brand-new database solely to run your test automation. You can set your database schema, add only the data you need for testing, create the database, point your tests to that database, and destroy the database when you are finished.
- **Give each team their own test space:** Even if teams have to share the same test environment, they might be able to divide their testing by account. For example, if your application has several test companies, each team can get a different test company to use for testing. This is especially helpful when dealing with toggles; one team's test company can have a feature toggled on while another team's test company has that feature toggled off.
- **Give each team their own users:** If you have a situation where all teams have to use the same test environment and the same test account, you can still assign each team a different set of test users. This way, teams won't accidentally overwrite one another's data.

You can give your users names specific to your team, such as “Sue GreenTeamUser.”

- **Create new data each time you test:** One easy way to manage test data is to create the data you need at the beginning of the test. For example, if you need a customer for your test, you create the new customer at the beginning of your test suite, use that customer for your tests, and then delete the customer at the end of your tests. This ensures that your test data is always exactly the way you want it, and it doesn’t add bloat to the existing database.
- **Use “today+1” for dates in the future:** Rather than choosing an arbitrary date in the future, which will eventually become a date in the past, you can use an add date operation to add some interval, such as a day, month, or year, to today’s date. This way, your test date will always be in the future.

Working with test data can be very frustrating. But with some planning and strategizing, you can ensure that your data will be correct whenever you run a test.

# Chapter 113: A Question of Time

Time is the one thing of which everyone gets the same amount. Whether we are the CEO of a company or we are an intern, we all have 1,440 minutes in a day. I've often heard testers talk about how they don't have enough time to test, and this can certainly happen when deadlines are imposed without input from everyone on the team. In this chapter, I tackle the question:

*Is it worth my time to automate this task?*

Sometimes we are tempted to create a little tool for everything, just because we can. I usually see this happen with developers more than testers, but I do see it with some testers who love to code. However, writing code does not always save us time. When considering whether to do a task manually or to write automation for it, ask yourself the following four questions.

## **Question #1: Will I need to do this task again?**

I was once on a team that was migrating files from one system to another system. I ran the migration tool manually and manually checked that the files had migrated properly. I didn't write any automation for this, because I knew I was never going to need to test it again.

Contrast this with a tester from another team who was continually asked to check the UI on a page when his team makes updates. He got really tired of doing this again and again, so he created a script that will take screenshots and compare the old and new versions of the page. Now he can run the check with the push of a button.

## **Question #2: How much time does this task take me, and how much time will it take me to write the code?**

Sometimes test data gets refreshed, which means the information we have for our test users gets changed. Whenever this happened to me, it would take about eight hours to manually update all my users. It took me a few hours to

create a SQL script that would update the users automatically, but it was totally worth my time, because after that, I saved eight hours of work whenever the data was refreshed.

But there have been other times when I've needed to set up some data for testing and a developer has offered to write a little script to do it for me. Since I can usually set up the data faster than they can create the script, I decline the offer.

### **Question #3: How much time will it take to maintain the automation I'm writing?**

Once, I was testing email delivery and I wanted to write an automated test that would show that the email had actually arrived in the email test account. The trouble was that there could be up to a 10-minute delay for the email to appear. I spent a lot of time adjusting the automated test to wait longer, to have retries, and so on, until finally I realized it was just faster for me to take that assertion out of the test and manually check the email account from time to time.

However, the automated API smoke tests I wrote took very little time to maintain because the API endpoints changed so infrequently that the tests rarely needed to change. The first API smoke test I set up took a few days; but once we had a working model, it became very easy to set up tests for the other APIs.

### **Question #4: Does the tool I'm creating already exist?**

At one company where I worked, the web team was porting over many customers' websites from one provider to another. I was asked to create a tool that would crawl through the sites and locate all the pages, and then crawl through the migrated site to make sure all the pages had been ported over. I created the tool, and shortly thereafter I discovered that web-crawling software already exists!

Nonetheless, I didn't feel like I'd wasted my time creating the tool. I had the time to do it, it was really fun to do it, and I learned a lot about coding that helped me with my other test automation. So, sometimes it may be worth

“reinventing the wheel” if it will help you or your team.

### **The Bottom Line: Are You Saving or Wasting Time?**

All of these questions boil down to one major consideration: whether your task is saving or wasting time. If you enjoy coding, you may be tempted to write a fun new script for every task you need to do; but this might not always save you time. Similarly, if you don’t enjoy coding, you might insist on doing repetitive tasks manually; but using a simple tool could save you a ton of time. Always consider the time-saving result of your activities!

# Chapter 114: Why the Manual Versus Automation Debate Is Wrong

I'm tired of the whole "manual versus automated testing" discussion. Some people describe automated testing as the cure for bad code everywhere and they pity the poor manual tester who has no technical skills. Meanwhile, there are advocates who say that automation is merely a panacea and that automation code should be used only for simple tools that will aid the manual tester, who is the one who really knows the product.

In my opinion, this debate is unnecessary for two reasons:

1. "Manual" and "automated" are arbitrary designations that don't really mean anything. If I write a Python script that will generate some test data for me, am I now an automation engineer? If I log in to an application and click around for a while before I write an automated test, am I now a manual tester?
2. The whole point of software testing, to put it bluntly, is to do as much as we can to ensure that our software doesn't suck. We often have limited time in which to do this. So we should use whatever strategies we have available to test as thoroughly as we can, as quickly as possible.

Let's take a look at three software testers: Marcia, Cindy, and Jan. Each of them is asked to test the Super Ball Sorter.

Marcia is very proud of her role as a "software developer in test." When she's asked to test the Super Ball Sorter, she thinks it would be really great to create a tool that will randomly generate sorting rules for each child. She spends a couple of days working on this, and writes an automated test that will set those generated rules, run the sorter, and verify that the balls were

sorted as expected. Then she sets her test to run nightly and with every build.

Unfortunately, Marcia didn't take much time to read the acceptance criteria, and she didn't do any exploratory testing. She completely missed the fact that it's possible to have an invalid set of rules, so sometimes her randomly generated rules are invalid. When this happens, the sorter returns an error, and because she didn't account for this, her automated test fails. Moreover, it takes a long time for the test to run because the rules need to be set with each test and she needs to build in many explicit waits for the browser to respond to her requests.

Cindy is often referred to as a "manual tester.". She doesn't have any interest in learning to code, but she's careful to read the acceptance criteria for the Super Ball Sorter feature and she asks good questions of the developers. She creates a huge test plan that accounts for many different variations of the sorting rules and she comes up with a number of edge cases to test. As a result, she finds a couple of bugs, which the developers then fix.

After she does her initial testing, she creates a regression test plan, which she faithfully executes at every software release. Unfortunately, the test plan takes an hour to run, and combined with the other features that she is manually testing, it now takes her three hours to run a full regression suite. When the team releases software, they are often held up by the time it takes for her to run these tests. Moreover, there's not enough time for her to run these tests whenever the developers do a build, so they are often introducing bugs that don't get caught until a few days later.

Jan is a software tester who doesn't concern herself with what label she has been given. She pays attention during feature meetings to understand how the Super Ball Sorter will work long before it's ready for testing. Like Cindy, she creates a huge test plan with lots of permutations of sorting rules. But she also familiarizes herself with the API call that's used to set the sorting rules, and she starts setting up a collection of requests that will allow her to create rules quickly. With this collection, she's able to run through all her manual test cases in record time, and she finds a couple of bugs along the way.

She also learns about the API call that triggers the sorting process and the call that returns data about what balls each child has after sorting. With these

three API calls and the use of environment variables, she's able to set up a collection of requests that set the rules, trigger the sorting, and verify that the children receive the correct balls.

She now combines features from her two collections to create test suites for build testing, nightly regression testing, and deployment testing. She sets up scripts that will trigger the tests through her company's CI tool. Finally, she writes a couple of automated UI tests that will verify that the sorter's page elements appear in the browser correctly, and she sets them to run nightly and with every deployment.

With Jan's work, the developers are able to quickly discover whether they've made any changes in logic that cause the Super Ball Sorter to behave differently. With each deployment, Jan can rest assured that the feature is working correctly as long as her API and UI tests are passing. This frees her to do exploratory testing on the next feature.

Which of these testers came up with a process that more efficiently tested the quality of the software? Which one is more likely to catch any bugs that come up in the future? I am fairly certain it will be Jan! Jan isn't simply a "manual tester," but she isn't a "software developer in test" either. Jan spends time learning about the features her team is writing and the best tools for testing them. She doesn't code for coding's sake, but she doesn't shy away from code either. The tools and skills she utilizes are a means to ensure the highest-quality product for her team.

# Chapter 115: Tear Down Your Automation Silos

On many software teams, developers are responsible for writing unit and component tests and software testers are responsible for writing API and UI tests. It's great that teams have so much test coverage, but problems can arise when test automation is siloed in this way. For one thing, developers and software testers often don't know how one another's tests work, which means if a developer makes a change that breaks a test, they don't know how to fix it. And if only one person on the team knows how the deployment smoke tests work, then that person will need to be on call for every single deployment.

I recommend that every developer and software tester on the team knows how to write and maintain every type of test automation for their product. Here are three good reasons to break down automation silos:

- **No more test overlap:** If automated tests are siloed between developers and testers, it's possible that work may be duplicated. Why have several UI tests that exercise business logic when there are already integration tests that do this?
- **No more bottlenecks:** Testers are often required to create and maintain all the UI automation while they're doing all the testing. If a developer pushes a change that breaks a UI test, it's often up to the tester to figure out what's wrong. If developers know how the UI automation works, they can fix any tests they break, and even add new tests when needed, allowing testers to finish testing new features.
- **Knowledge sharing:** Software testers have a very special skill set—they can look at application features and think of ways to test the limits of those features. By learning from testers, developers will become better at testing their own code.

Developers have a very special skill set as well: they are very familiar with good coding patterns. Many software testers came to their vocation from diverse backgrounds and don't always have formal training in coding. They can benefit from learning clean coding skills from developers.

By breaking down automation silos and taking responsibility for test automation together, software developers and software testers can benefit from and help one another, speeding up development and improving the quality of the application.

# Chapter 116: Stop Writing So Many UI Tests

If you were to guess the importance of various types of automated tests by looking at the number of tutorials and articles about them on the Web, you'd think that UI tests were the most important. But this is not the case. So much of an application can be tested through other means, especially API tests. API tests are faster and much less flaky than UI tests, and they're easier to write as well! Following are four types of tests that are better suited for API testing.

## **Login Tests**

It's easy to cycle through all kinds of username and password combinations with API tests. The response time from a POST to a login endpoint is lightning fast, as opposed to a UI test which has to wait for the username and password fields to be populated and wait for the login response to reach the browser. To prove this, I created a Postman collection that had 16 login tests with various username and password combinations. The 16 tests ran in less than three seconds! Imagine how long the equivalent UI tests would take.

That said, you should have two automated UI tests for login: one that validates that the login page looks correct and the user is able to log in, and one that validates that an appropriate error message is displayed when the user attempts to log in with bad credentials.

## **CRUD Tests**

When you're testing CRUD functionality, you're testing how your application interacts with the underlying data store. This is best done at the API level. It's easy to create GET, POST, PUT, and DELETE tests using a tool like Postman. You can assert on both the response codes and the body of the response (if any), and you can also do GETs to assert that your POSTs, PUTs, and DELETEs have been saved correctly to the database.

The only UI tests you need in this area are one that demonstrates that form fields can be filled out and submitted, and one that shows that data is displayed correctly on the page.

## Negative Tests

API testing is great for negative tests because not only can you run through all kinds of negative scenarios very quickly, you also can run tests that aren't possible in the UI. For example, let's say your form has a required field. In the UI, you can't test whether you can submit a new record without that required field, because the UI simply won't let you. But in the API, you can do a POST without the required field and verify that you are getting a 400-level response. API testing is also great for checking application security because malicious users are likely to try to attack the application at this level.

Here is just a sampling of the types of negative tests you can run with API testing:

- Sending in an inaccurate URL
- Trying a request without appropriate authentication
- Testing for IDOR (Insecure Direct Object Reference)
- Sending in incorrect headers
- Sending in a request without a required field
- Trying a request with a field value that violates type or length constraints
- Verifying that the correct 400-level error is displayed when a request is invalid

For UI negative testing, you'll simply want to verify that appropriate errors are displayed on the page when you leave a required field blank or violate a field constraint. Everything else can be covered by API tests.

## Tests of Complicated Business Logic

If an area of your application requires extensive data setup and complicated business logic, it's much easier to test with an API than with the UI. Let's examine this with the Super Ball Sorter. Setting up the rules through the UI in an automated test would be tedious; assuming each child

had a dropdown picker for size and color, you'd need to do a lot of element selection. But if the Super Ball Sorter had an API that could set all the rules for the children in a single POST, it would take milliseconds to prepare the test.

Similarly, after the sorting has been run, a UI test would need to grab all the responses on the page to validate that the balls have been sorted correctly; instead, an API could do a GET request for each child and validate that the appropriate balls are returned. Four GET requests will most likely be returned and validated before a UI test could validate a single child's values.

Now that you have seen the many ways that API tests can be used, I hope you will take the time to look at your current UI test suite to see which tests could be shifted to API testing. Your automation suite will be faster, more reliable, and easier to maintain as a result!

# Chapter 117: Five Reasons You're Not Ready for Continuous Deployment

Continuous deployment (CD) is often seen as the holy grail of software development. A developer checks in code, and it is miraculously deployed and tested in the QA, staging, and production environments, without needing any human intervention at all. This sounds great—and it is, but only if you are ready for it! Here are five reasons your team might not be ready for continuous deployment.

## **Reason #1: You Don't Have Enough Test Coverage**

Sometimes teams can be so excited to set up continuous deployment that they don't pay attention to what they are testing. It's great to have tests pass and deployments automatically go all the way to production, but if you are missing tests for important functionality, you're going to need to remember to do manual testing with every deployment. Otherwise, something could break and the automated tests won't pick up on the problem.

The remedy: make sure you have all the tests you need before you set up CD.

## **Reason #2: Your Tests Are Flaky**

If your tests aren't reliable, you are going to get all sorts of false failures. With CD set up, this means deployments will fail. If your developers are trying to deploy to the QA environment but they can't get their code there because of your flaky tests, they will be annoyed. And no one wants to have to stop what they are doing to investigate why your automation failed in production.

The remedy: make sure your tests are reliable. If there are flaky tests, pull them out of the test suite until you can fix them, and make sure you are manually testing anything that's no longer covered by automation.

### **Reason #3: Your Tests Take Too Long**

UI tests can take a very long time. If you really want to set up CD, you'll have to consider how much time the tests are taking. If Developer A checks in code that kicks off the tests and then has to wait an hour to find out whether the tests passed, and meanwhile Developer B checks in code that now has to wait until the first deployment has completed, soon you will have a mess on your hands.

The remedy: make sure your tests are fast. See which tests you can shorten through strategies such as switching to API tests for testing backend logic, setting up your test data ahead of time, using API and other services calls to set up conditions for tests, running tests in parallel, and eliminating redundant tests.

### **Reason #4: You Don't Understand the Deploy Process**

Having CD set up won't be helpful at all if you and your team don't understand how it works. When things go wrong with a critical deployment, you don't want to have to find someone in DevOps to help you diagnose the issue. That will waste the DevOps engineer's time and cause stress for everyone on the team.

The remedy: make sure everyone on the team understands the deployment process. Learn how to configure the deployments, what common errors mean, how to fix a hung deploy, and so on. Take turns monitoring the deployments and solving problems so that you aren't dependent on one team member who can then never take a vacation.

### **Reason #5: You Don't Have Alerting Set Up**

Just because your deployments are now automatic doesn't mean you can sit back, relax, and never think of them again! Sometimes your tests will fail, sometimes your connections to dependencies won't get configured properly, and sometimes an unusual situation will happen that will fail the deployment. You don't want to find this out from your CEO, or someone in DevOps, or your customers!

The remedy: make sure you have alerting and paging set up when deployments fail. You could have the person who made the code change get paged when there's a failure, or you could have everyone on the team take turns being the one on duty for that week. Make sure everyone takes their paging seriously; if they're on call for a week where they're going to be on vacation, they should find someone to substitute for them.

Continuous deployment, when done correctly, is a valuable tool that makes it easier for teams to quickly produce quality software. But be sure you are completely ready for this step by taking an honest, objective look at these five reasons with your team.

## Part XII: Soft Skills for Testers

# Chapter 118: Ask Your Way to Success

Twelve years ago I didn't know how to use a Windows computer. I didn't know how the filesystem worked. I didn't know what right-clicking on a mouse did. Today I am a principal engineer, managing the quality efforts of the company I work for. How did I get here from there?

I asked a lot of stupid questions.

Most people are reluctant to ask questions because they are afraid to look ignorant. But I maintain that the best way to learn anything quickly is to ask questions when you don't understand what's going on.

Here are six ways that asking questions improves your knowledge and the health of your company:

**1. Questions give others an opportunity to help you, which helps them get to know you better and establishes rapport.**

At my first official testing job, I was working with hotshot developers, all of whom were at least a decade and a half younger than I was. It was embarrassing having to admit that I didn't know how to reset a frozen iPhone or find the shared drive in File Explorer, but I asked those questions anyway, I remembered the answers, and I showed my co-workers that I was a fast learner.

**2. Questions help developers discover things they may have missed.**

On countless occasions when a developer has been demonstrating a feature to me, I'll ask a question like "But what if there are no records for that user?" or "What if GPS isn't on?" and they will suddenly realize there is a use case they haven't handled.

**3. Questions keep everyone honest.**

I have worked with test engineers who toss around terms and phrases

like “backend call” and “a different code path” without actually knowing what they are talking about. Asking an engineer to clarify what they mean ensures that they do the work to find the answer. And when they get their answer, I get my answer as well.

#### **4. Questions give you an opportunity to clear things up in your head.**

You may have heard the expression “rubber duck debugging,” which occurs when a developer is advised to explain their coding problem to an inanimate object, and in the process of explaining they come up with their answer. I think this method works well when you’re asking questions. I have found that sometimes just formulating the question out loud while I’m asking it clears things up for me.

#### **5. Questions clarify expectations.**

Sometimes I have felt silly asking things like “You want me to test this on the latest build, right?” But every now and then I discover that there’s been a miscommunication, and I’d much rather find out about it before I start testing than after I’ve been testing the wrong thing for an hour.

#### **6. Questions clarify priorities.**

Many times I’ve asked, “Why are we adding this feature?” There is almost always a good reason, but the discussion helps the team understand what the business use case is, which helps the developers decide how to design their solution.

While it’s good to ask questions, don’t ask questions that you can find the answers to by using a search engine (e.g., “How do I find the UDID of a device using iTunes?”) or by going back and reading your email (e.g., “What day did we decide on for the code freeze?”). Asking these types of questions results in wasted time for everyone!

In summary, asking might make you feel silly in the short run, but it will make you and your team much smarter in the long run. And hopefully it will create an atmosphere in which others feel comfortable asking questions as

well, improving teamwork for everyone!

# Chapter 119: Seven Excuses Software Testers Need to Stop Making

Recently I read an interesting book titled *Extreme Ownership*. Written by two Navy SEAL officers, it describes the concept of taking responsibility for every facet of your job, even the things you feel you have no control over. If one of their soldiers made a mistake, the officers would take responsibility because they could have trained the soldier better. If their commander made a poor decision, the officers would take responsibility for that as well because they could have “managed up” and provided information that would have led to a better decision. When everyone exercises extreme ownership, a culture of excellence and achievement is the result.

Extreme ownership can be applied to any career, including software testing! Yet I often hear software testers give a number of excuses for not applying this concept to their work. Excuses keep us from taking full ownership of our work and from being taken seriously. Following are seven excuses that software testers need to stop making.

## **Excuse #1: I don't know how the feature works.**

All too often, testers simply follow the meager directions left for them by the developer in the software story, without having any idea what they are doing. For example: “Run this SQL query and verify the result is 1.” Why? What information is this query obtaining? How do you know this answer is the right one? If it turns out there is a bug related to this feature, how can you possibly say you’ve tested it?

When you are presented with a story to test that you don’t understand, start asking questions. If the developer can’t explain the feature to you, find someone who can. Restate the information you are given to make absolutely sure you understand it correctly. Ask for the information to be presented in a way that makes sense to you.

Many times I have uncovered bugs in a feature before I started testing,

simply by asking questions about how the feature works!

### **Excuse #2: There's no way to test the feature.**

Really? There's no way to test the feature? Then how does the developer know the feature is working? Are they just sending it to you and hoping for the best? There must be some way for your developer to know their code is working. What is that way? Can they show it to you?

Sometimes I am unable to test some features myself because I don't have access to the backend system that is being used in the feature. When this is the case, I make sure to work with the developer and have them show me that the feature is working. Then I can ask them to try various test cases while I watch so that we are pair-testing the feature. In this way, we can uncover any bugs that may exist.

### **Excuse #3: The developer coded it incorrectly.**

At times I have seen instances when a developer misunderstood the requirements of the feature to be built and created it incorrectly. This is why it's important for everyone on the team to understand the requirements and ensure that acceptance criteria are included in the story. If you test the story based solely on what the developer tells you and don't verify exactly what was supposed to be built, then the fault lies with you. You are the tester—usually the last line of defense before the product goes to the customer. Make sure the customer is getting the right thing!

### **Excuse #4: The other tester on my team missed the bug.**

Even the best software testers miss a bug now and then. That's why it's important to have at least two sets of eyes on every feature. I once set a policy on my team that when one tester was finished testing a feature, a different tester would be asked to test the feature in the next environment. The week after I instituted this policy, one of my co-workers found two bugs that I missed!

If you are the only tester in your company, set up a “bug hunt” where everyone in the company looks for bugs. Don't be embarrassed if someone

finds something you missed; when we test the same thing over and over again, we can sometimes develop inattentional blindness.

### **Excuse #5: There wasn't enough time to test.**

Let's face it: there will never be enough time to test everything you want to test. Software developers have time constraints too; they would probably really like to refactor their code a few more times before they hand it over to you, but they are working with a deadline just as you are. So, instead of making excuses, test the most important things and manage your time wisely.

### **Excuse #6: If I log the bug I found in production, I'll be asked why I didn't find it sooner.**

Some managers blame testers for finding bugs in production, but these managers are misguided. It's up to us to educate our team about what testers do. We simply can't find every bug; there are too many ways that software can go wrong. What we can do is report what we find as soon as we find it, and keep an eye out for similar bugs next time. If you don't report that bug in production, it will go unfixed, and the next person to find it will be a customer or the CEO of your company!

### **Excuse #7: I don't know how to code.**

Software development has changed significantly over the past two decades. Companies used to release software every six months, so testers had tons of time to do regression testing. Now software is released every week or two. It's simply not possible to manually test an entire application during that time frame. This is why automation is necessary, and why you need to learn how to automate!

You don't have to take a college course in Java to learn how to code (although that is a fine idea if you have the time). All coding languages run on some very simple logical principles that are easy to understand. The only tricky thing is the syntax of whatever language is being used, and the more you expose yourself to the code, the more you will understand.

If there's no test automation at your company, see whether you can get

one of your developers to write some tests. If you have software testers who are already writing automation at your company, ask them to walk you through their tests. Learn how to make a simple change to an automated test, such as changing an assertion that says “true” to one that says “false”. Copy a test that verifies the value of a text field, and see whether you can change it so that it verifies the value of a different text field. Learn how your company’s version control system works, and see whether you can submit a code change for your team’s approval.

Take small steps! You don’t have to learn it all at once. Think of learning code as learning a new language. When you learn a new language, no one expects you to be fluent right away. You learn a few phrases and keep using them, and you gradually add more.

Software testing is such a valuable profession, but too often companies take testers for granted. By applying the principles of extreme ownership and eliminating excuses from your vocabulary, you will come to be seen as an indispensable asset to your company.

# Chapter 120: Six Testing Personas to Avoid

If you are working for a company that makes software for end users, you have probably heard of user personas. A user persona is a representation of one segment of your application's end users. For example, if you worked for a company that made a website for home improvement supplies, one of your user personas might be New Homeowner Nick, who has just purchased his first home and might not have much experience fixing small things in his house. Another persona might be Do-It-Yourself Dora, who has lots of experience fixing everything in her home herself.

It occurred to me recently that there are also testing personas. But unlike our user personas, these personas are ones we want to avoid! Read on to see whether one of these personas applies to you.

## **Persona #1: Test Script Ted**

Ted loves running manual test scripts and checking them off when they're completed. It gives him a feeling of satisfaction to see tests pass. He doesn't particularly care whether he doesn't understand how his application works; he's just satisfied to do what he's told. But because he doesn't understand how the application works, he sometimes misses important bugs. If he sees something strange but it's not addressed in the test plan, he just lets it slide. His job is to test, not figure things out!

## **Persona #2: Automation Annie**

Annie considers herself an automation engineer. She thinks manual testing is a colossal waste of her time. She'd rather get into the hard stuff: creating and maintaining automated tests! When a new feature is created, she doesn't bother to do any exploratory testing; she'll just start coding, and she figures her great automation will uncover any issues.

### **What Ted and Annie have in common:**

Ted and Annie are making the same mistake for different reasons; they

are not taking the time to really learn how their application works. They're both missing bugs because of a lack of understanding; Ted doesn't understand the code that makes the features work and Annie doesn't understand the application's use cases.

### **How not to be Ted or Annie:**

To be a thorough tester, it's important to take the time to understand how your features work. Try them out manually; explore their limits. Look at the code to see whether there are other ways you might test the features. Ask questions when you see things that don't make sense.

### **Persona #3: Process Patty**

Patty is passionate about quality. She likes things to work correctly. But she likes having processes and standards even more! She's got test plans and matrices she's expecting her team to follow precisely. Regression testing must be completed before any exploratory testing is done, and there are hundreds of regression tests to be run. The trouble is, with releases happening every two weeks there's no time to do any exploratory testing. There's no time to stop and think about new ways to test the product, or what might be missing. The team needs to get all those regression tests completed!

### **Persona #4: Rabbit Hole Ray**

Ray is passionate about quality too; he doesn't want any bug to go unnoticed. So when he sees something strange in the application when it runs on IE10, he's determined to find out what's wrong! He will take days to investigate, looking at logs and trying different configuration scenarios to reproduce it. He doesn't want to be bothered with the standard regression tests that he's leaving undone as the feature is being released. And he doesn't care that only .05% of their customers are using IE10. He's going to solve the mystery!

### **What Patty and Ray have in common:**

Patty and Ray are both wasting time. They are focused on something other than the primary objective: releasing good software on time with a

minimum of defects. Patty is so caught up in the process that she doesn't see the importance of exploratory testing, which could find new bugs. And Ray is so obsessed with that elusive bug he's exploring that he's ignoring important testing that would impact many more users.

### **How not to be Patty or Ray:**

When testing a new feature or regression testing existing ones, it's important to think about which tests will have the biggest impact and plan your testing accordingly. Be careful not to get too caught up in processes, and if that elusive bug you're searching for won't be that impactful to end users, let it go.

### **Persona #5: Job Security Jim**

Jim's been working at his current position for years. He knows the application like the back of his hand. He's the go-to guy for all those questions about how the most ancient features behave. He knows there's no way the company will let him go; that would mean they'd lose all his knowledge! So he doesn't feel like there's any reason to learn new skills. What he knows has served him just fine so far. Who needs to waste time after work learning the latest programming language or the newest testing tool?

### **Persona #6: Conference Connie**

Connie is so excited about tech! She loves to hear about the latest testing techniques and development trends. She signs up for webinars, goes to conferences, reads blog posts, and takes courses online. She knows a little about almost everything! But she has never actually implemented any of the new things she learns. She's so busy going to conferences and webinars that she barely has time to do her regular testing tasks. And besides, trying things out is a lot of work. It's easier to just see how other people have done it.

### **What Jim and Connie have in common:**

Jim and Connie seem like total opposites at first: Jim doesn't want to learn anything new and Connie wants to learn everything new. But they actually have the same problem: they are not growing as testers. Jim is

content to do everything he has already learned and doesn't see any reason to learn anything more. But he could be in for a shock one day if his company decides to rewrite the software and he suddenly needs a new skill. And Connie has lots of great ideas, but great ideas don't mean anything unless you actually try them out. Her company isn't benefiting from her knowledge, because she's not putting it to use.

### **How not to be Jim or Connie:**

It's important to keep your testing skills fresh by learning new languages, tools, and techniques. You don't have to learn everything under the sun; just pick the things that you think would be most beneficial to your current company, learn them, and then try to implement them in one or two areas. Your teammates will be thankful for the new solutions you introduce, and you'll be developing marketable skills for your next position.

### **Be a Great Tester, Not a Persona!**

We all become some of these personas now and then. But if we can be aware of them, we can catch ourselves if we start to slip into Automation Annie or Rabbit Hole Ray, or any of the others. Great testers learn their application better than anyone else, they make good choices about what to test and when, and they keep their skills updated so that their testing keeps improving.

# Chapter 121: How to Train Your Dev

Training your dev is really about training yourself. A more accurate (but much less catchy) title for this chapter would be “How to Work and Communicate Effectively in Order to Facilitate a Productive Relationship with Developers.”

There are two steps to having a good working relationship with your developer: 1) developing good work habits, and 2) communicating clearly. We’ll take a look at these two steps in detail.

## Good Work Habits

- Make sure you have completely read a feature’s acceptance criteria and all available notes and documents. This can help prevent unnecessary and time-consuming misunderstandings.
- Ask questions if there is anything in the feature that you don’t understand. Don’t make potentially incorrect assumptions.
- Document your work. This is especially helpful when you have found an issue and the developer needs to know what browser you were using or what server you are pointing to.
- Check twice to make sure you really have a bug. Perhaps what you are seeing is a configuration problem, a connection problem, or simply user error.

## Clear Communication

- Learn your dev’s preferred communication style and use it. For example, some developers like to hear about issues immediately, and testing and bug fixing become a collaboration. Other developers prefer to hear right away only if the issues are big ones, and would rather have you document the smaller issues for a later conversation.

- Ask your dev to walk you through any confusing features. They will be happy to explain things to you because they know that any information they give you at the outset of testing will save misunderstandings later.
- Be kind when reporting issues. Your dev has worked hard on the feature they delivered, and we all know it's no fun to have our work criticized.
- Give feedback in the form of a question. This can soften the blow of finding a bug. For example: "I noticed that when I clicked the Save button, I wasn't taken to the next page. Is this as designed?"
- Let your dev know what they can do to help you do your job more efficiently. A good example of this is asking them not to assign an issue to you until it is actually in the test environment so that you won't inadvertently start testing it before the code is there.

A good working relationship with your dev is all about trust! You trust that your dev has completed the work they've assigned to you, they've done some of their own testing before the handoff, the work is in the test environment and ready for testing, and they've let you know about any potential areas of regression to test.

In turn, your developer trusts that you have tested everything in the acceptance criteria, you've done regression testing, you've tested with various security levels and on various browsers, the issues you've found are legitimate, and you will clearly communicate what you tested and what issues you found.

Train yourself to work effectively and communicate clearly, and you will create this level of trust in your relationship with all the developers you work with!

# Chapter 122: Get Organized for Testing Success

Before I discovered the joy of software testing, I had a brief career as a professional organizer. I organized homes, small businesses, and nonprofit organizations. I've always loved getting organized because it helped me accomplish my goals more quickly. The same is true with software testing! Being organized as a tester means you have easy access to your tools, test plans, and resources, which frees you up to do more creative thinking and exploratory testing. In this chapter, I'll outline four of my strategies for organizing.

## **Strategy #1: Avoid Reinventing the Wheel**

At various times in my testing career, I've needed to test a file upload feature. I made sure to test with different file types: PDF, JPEG, PNG, and so on. Sometimes it was hard to find the file type I was looking for; for instance, it took me a long time to locate a TIFF file.

After I had tested file uploading a couple of times, I realized it would be a good idea to save all the files I'd found in a folder called File Types for Testing. This way, the next time I needed to test file uploads I would have all my files ready to go.

Recently I expanded my File Types for Testing folder to include some very large files. Now when I need to test file size limits I don't have to waste a second looking for files to use.

Similarly, I have a folder of bookmarked web pages which contains all the tools I use regularly, such as a character count tool and a GUID generator. This way, I don't need to spend valuable time conducting a search for the tool or asking a co-worker to remind me where the tool is.

## **Strategy #2: Be Consistent with Naming and Filing**

Every now and then someone will ask me how I tested a feature, or I'll ask myself the question because I need to do some regression testing. If I don't remember what I named my test plan when I saved it or what folder I saved it to, I'll waste a lot of time looking for it. For this reason, I name all my test plans consistently: the name begins with the JIRA ticket number, and then I include a brief description of the feature—for example, "W-246- File resizing".

When I first started consistently naming my test plans, I just named them with the description, but that made them difficult to find because I could never remember what verbiage I used: was it "Resizing files" or "File resizing"? Then I named them with just the JIRA ticket number, but locating them required two steps: first I needed to remember the ticket number by searching through JIRA, and then I needed to look up the test plan. Naming the test plan with both the number and the description gives me two ways to find the plan, which speeds up the process.

I also organize my test plans by feature. For example, all my test plans associated with messaging go in a Messaging folder. And all my test plans associated with file uploads go in a File Upload folder.

### **Strategy #3: Have a Place for Shared Tests**

As much as I love not reinventing the wheel myself, I also enjoy helping others avoid doing so. When I was a tester on a development team, I created a shared workspace where I put all our saved collections. The collections were organized by API, so they were easy to find. Really long collections were organized in subfolders by endpoint or by topic. This was helpful, not just for the other testers on my team, but also for the developers; they mentioned to me that it was much faster for them to reproduce and fix an issue when they could use our saved requests, instead of setting up their own.

On that same team, we saved all our regression test plans in Confluence. They were organized by version number for major releases, and by API and date for smaller releases. We used Confluence because it was easy to collaborate on a test plan; we added our name to the tests we ran so that we could see who was working on each section and which tests had been completed. Saving the test plans this way made it easy to go back and see

what we tested, as well as easy to duplicate and edit a plan for the next release.

### **Strategy #4: Leave Yourself Notes**

Whenever I get a new piece of information, such as a test user's credentials or a URL for a test environment, I say to myself, "Am I likely to need this information again?" If so, I make sure I add it to my notes. I used to use a notebook for notes like this, but now I use Notepad++. Keeping this information in saved files makes it easier to locate, instead of searching back through pages of a notebook. I keep all my Notepad++ files in the same folder, and I give them recognizable names, such as "Test Users" or "Email Addresses for Testing".

As in any company with more than one employee, we share files, and sometimes other people don't file things in the places where I would expect them. After getting really frustrated trying to find the same information over and over again, I created a spreadsheet for myself called "File Locations". This spreadsheet has a column for what I would have named the file, and then a column with a link to get to the file. This has saved me valuable time searching for files, and freed me from frustration.

When I have a piece of information that I need to save, but I know I will only need it temporarily, I save it in a Notepad++ file called "Random Notes". I periodically delete information that is no longer needed to keep the file from getting too long and hard to read.

Organizing files, test plans, and information takes a little bit of time at first, but with practice it becomes second nature. And it saves you the time and frustration of constantly searching for the information you need. With the time you save, you can do more exploratory testing, which will help find new bugs; and you can write more test automation, which will free you up to do even more exploratory testing!

# Chapter 123: Time Management for Testers

It's a perennial problem: there's so much testing to be done and not enough time in which to do it. I already wrote about this in Chapter 102, which talks about how to prioritize your testing and how to work with your team to avoid getting into situations where there's not enough testing time. But in this chapter I'm taking a more general view of time management: how can we structure our days so that we don't feel continually stressed by the many projects we work on? Here are eight time management strategies that work for me.

## **Strategy #1: Know Your Priorities**

I had a biweekly one-on-one meeting with a previous manager, and in each meeting he asked me, "What's the most important priority for you right now?" I love this question because it helps me focus on what's most important. You may have 10 different things on your to-do list, but if you don't decide which things are the most important, you will always feel like you should be working on something else, which keeps you from focusing on the task at hand. I like to think about my first, second, and third priorities when I am planning what to work on next.

How do you decide what's most important? One good way is to think about impacts and deadlines. If there is a release to production that is going out tonight and it requires some manual testing, preparing for that release is going to be your top priority because customers will be impacted by the quality of the release. If you are presenting a workshop to other testers in your company and that presentation is tomorrow, you're going to want to make that preparation a priority. If you evaluate each task in terms of its impact and due date, it will become clearer how your priorities should be ordered.

## **Strategy #2: Keep a To-Do List**

Keeping a to-do list means you won't forget about any of your tasks. It

does not mean that all your tasks will get done. When I finally realized that my to-do list would never be completed, I was able to stop worrying about how many items were on it. I have found that the less I worry about how many things are on my to-do list, the more items I can cross off from it.

### **Strategy #3: Use Quiet Times for Your Deepest Work**

One of the best things about working for a remote-friendly company is that our employees are spread over four time zones. Since I am in Eastern Standard Time, our daily meetings don't start until late morning in my time zone. This means the first couple of hours of my workday are free of interruptions. So I use those hours to work on projects that require concentration and focus. My teammates on the West Coast do the opposite, using the end of their workday for their deepest work because the rest of us have already stopped for the day.

Even if you are in the same time zone as your co-workers, you can still carve out some quiet time to get your most challenging work done. Maybe your co-workers take a long lunch while you choose to eat at your desk. Or maybe you are a morning person and get into the office before they do. Take advantage of those quiet times.

### **Strategy #4: Minimize Interruptions**

It should come as no surprise to anyone that we are interrupted with notifications on our phones and laptops several times every hour. Every time one of those notifications comes through, your concentration is broken as you take the time to look at the notification to see whether it's important. But how many of those notifications do you really need? I've turned off all notifications except text messages and work-related messages on my phone. Any other notifications, such as LinkedIn, Facebook, and email, are not important enough to cause me to break my concentration.

On my laptop, I've silenced all my notification sounds except one: the notification that I'm about to have a meeting. That way, I have fewer sounds disrupting my concentration.

Another helpful hint is to train your team to send you an entire message

all at once, instead of sending messages like this:

9:30 Fred: Hi  
9:31 Fred: Good morning  
9:31 Fred: I have a quick question  
9:33 Fred: Do you know what day our new feature is going to production?

If you were to receive the messages in the preceding example, your work would be interrupted four times in just four minutes. Instead, ask your co-workers to do this:

9:33 Fred: Good morning! I have a quick question for you: do you know what day our new feature is going to production?

In this way, you are only interrupted once. You can answer the question quickly and then get back to work.

### **Strategy #5: Set Aside Time for the Big Things**

Sometimes projects are so big that they feel daunting. You may have wanted to learn a new test automation platform for a long time, but you never seem to find the time to work on it. While you know that learning the new platform will save you and your colleagues time in the long run, it's not urgent, and the course you'd like to take will take you 10 hours to complete.

Rather than trying to find a day or two to take the course, why not set aside a small amount of time every day to work on it? When I have a course to take, I usually set aside 15 or 20 minutes at the beginning of my workday to work on it. Each day I chip away at the coursework to be done, and if I keep at it consistently, I can finish a 10-hour course in six to eight weeks. That may seem like a long time, but it's much better than never starting the course at all!

### **Strategy #6: Ask for Help**

We testers have a sense of personal pride when it comes to the projects we work on. We want to make sure we are seen as technologically savvy and

not as “just a tester.” We take pride in the automation we write. But the fact is that developers usually have more experience working with code than we do, and they might have ideas for better or more efficient ways of doing things.

Recently, I was preparing an example project in C# to teach some new employees how to write unit tests. It was just a simple app that compared integers. I knew exactly how to write the logic, but when I went to compile my program, I ran into a “cannot instantiate class” error. I knew the cause of the error was probably something tiny, but since I don’t often write apps on my own, I couldn’t remember what the issue was. I had a choice to make at this point: I could save my pride and spend the next two hours figuring out the problem by myself, or I could ask one of my developers to look at it and have him tell me what the problem was in less than 10 seconds. The choice was obvious: I asked my developer, and he instantly solved the problem.

However, there is one caveat to this strategy: sometimes we can get into the habit of asking for information that we could easily find ourselves. Before you interrupt a co-worker and ask them for information, ask yourself whether you could find it through a simple browser search or by looking through your company’s wiki. If you can find it yourself faster than it would take to ask your co-worker and wait for their response, you’ve just saved yourself and your co-worker some time!

### **Strategy #7: Take Advantage of Your Energy Levels**

I am a morning person; I am the most energetic at the start of my workday. As the day moves along, my energy level drops. By the end of the workday, it’s hard for me to focus on difficult tasks. Because of this, I organize my work so that I do my more difficult tasks in the morning, and save the afternoon for more repetitive tasks.

Your energy level might be different; if you think about what times of the day you do your best and worst work, you will be able to figure out when you have the most energy. Plan your most challenging and creative work for those times.

### **Strategy #8: Adjust Your Environment**

I am very fortunate that I am able to work remotely. This means I have complete control over the cleanliness, temperature, and sound in my office. You may not be so lucky, but you can still find ways to adjust your environment so that you work more efficiently. If your office is so warm that you find yourself falling asleep at your desk, you can bring in a small fan to cool the air around you. If you are distracted by back and shoulder pain that comes from slumping in your chair, you can install a standing desk and stand for part of your workday. If your co-workers are distracting you with their constant chitchat, you can buy a pair of noise-canceling headphones.

Experiment with what works best for you. What works for some people might not be right for you. You might work most efficiently with total silence, white noise, ambient music, classical music, or heavy metal playing in your ears. You might find that placing your desk so that you can look out the window helps you relax your mind and solve problems, or you might find it so distracting that you are better off facing empty, white walls. Whatever your formula, once you've found it, make it work for you!

The eight strategies I listed here can make it easier for testers to manage their time and work more efficiently. You may find that these strategies help in other areas of your life as well: paying bills and doing housework, home improvement projects, and so on. If these tips work for you, consider passing them on to others in your life to help them work more efficiently too!

# Chapter 124: How to Be Seen

One of the difficulties of being a software tester is that when you're doing your job really well, it's unnoticeable! Unlike software developers, who are creating a product that will then be seen by management, software testers create tests that will help validate that the product is working correctly. When we do a great job, it's not clear what the difference is between the product that would have gone out to production if we hadn't found all the bugs, and the product that actually did go out to production.

The problem with not being seen by management is that it becomes difficult to advance in one's career. Fortunately there are ways that we can make sure our managers and others see the impact we are making.

## **Tell Them**

Make sure you are letting your manager know all the ways you are helping to make a great product. For example, in your daily stand-up meeting, you could say, "I found an important bug yesterday in the chat feature that would have kept users from accessing their chat window if it went to production."

Also be sure to mention whenever your test automation catches a bug: "Our automated regression suite caught a critical bug on the User Info page shortly after the change was deployed to the test environment."

## **Show Them**

Managers love dashboards and metrics. Remember that managers often have their own managers to whom they need to report. If you can make your manager's job easier by providing them clear data about the quality of your application, they will be very grateful.

For example, you could create a dashboard that shows the pass/fail rates of your nightly regression tests. This dashboard could show the different environments you are testing in, and ideally it should show that the passing

rates in your production environment are close to 100% because the automation found the bugs well before the new code made it to production.

Or you could start keeping metrics of escaped defects: these are bugs that made it to production without being noticed. Ideally the number of escaped defects will be zero, but even if it isn't, your metrics for each release can demonstrate that your team is getting better at releasing bug-free code.

## **Teach Them**

Having lots of great software testing skills is awesome, but even more awesome is teaching those skills to others in your company so that they will be as effective as you are.

Your company probably has a number of different ways that you can teach others testing-related skills. For example:

- Talking about an automated test framework you are using in a departmental meeting
- Leading a workshop for other testers on security or performance testing
- Mentoring a new or struggling tester

## **Lead Them**

Your manager can't see you as a leader if you don't speak up! One great way to lead is to suggest process improvements for your team during your sprint retrospective meetings. When your team adopts those ideas and sees the quality of your product improve as a result, you'll be viewed as a positive change agent.

Setting up a Community of Practice (CoP) meeting is also a great way to stand out as a leader. In this meeting, all the testers at your company can gather together and share ideas and solve problems. If your company already has a CoP meeting, volunteer to lead a discussion or talk about an innovation that your team recently adopted.

## **Common Objections to Being Seen**

Software testers are often introverts. Many of them enjoy working quietly, testing features and writing automation without much interruption from others. Some prefer not to be the center of attention. It's OK to feel this way, but it will not get you promoted!

If you are shy or fear public speaking, start out by making small improvements. You could begin by writing a blog post or creating a test dashboard. Then you could try adding one comment in each sprint retrospective meeting. Next, you could volunteer to demo a new feature at a meeting. You can continue to add small steps in this way until you feel comfortable enough to run a workshop.

## **Final Thoughts**

Software development processes have come a long way in the past two decades; very few managers today don't understand the importance of software testing. But they may not notice your contributions to your product unless you make sure to be seen. I hope these suggestions will help you show your manager what a great asset you are to your team.

## Acknowledgements

This book would not have been possible without the help of many individuals:

Thanks to my editor, Audrey Doyle, who rewrote my awkward sentences and made them professional.

Thanks to my cover designer, Vanessa Mendoza, for designing a cover that fit my personality perfectly.

For help with the translations in Chapter 57, thanks to Damien Gonot, Lisi Hocke, and Homero Barbosa.

Thank you to my loyal blog readers Alena Dubeshko, Srinivas Kadiyala, and Sunny Sachdeva for reading through early chapters of this book and providing feedback.

A big thank you to my husband, Kevin Jackvony, for putting up with hours of neglect while I worked on the book and the app that accompanies it, and for testing out the instructions in Chapters 32 through 34.

Finally, thank you to all of the developers and testers I've worked with over the last twelve years who took the time to explain software concepts to me. You have helped shape me into the tester I am today.

## About The Author

### **Kristin Jackvony**

Kristin Jackvony discovered her love of software testing after a career in music education. She has been a QA Engineer, QA Lead, QA Manager, and SDET, and is currently the Principal Engineer for Quality at Paylocity. She writes regularly in her blog, "Think Like a Tester", which can be found at <https://thinkingtester.com>.