System-Level Testing Principles

For Web-Based Software Applications

By Jenny Riecken

Table of Contents

Introduction

This manual will give you a good understanding of how to approach software testing at a system level. You will learn the basic principles of system-level testing, and the high-level test areas to consider. You will also learn what types of tests to run in each area.

"Quality is never an accident; it is always the result of intelligent effort"

– John Ruskin

Testing is a very important part of the software development process. Unfortunately, this essential job often falls to people who have not been trained to do software testing.

If you have suddenly been thrust into a testing role without training, you may feel overwhelmed. You may not know where to start, or how to make sure you're covering everything. Gaps in testing can lead to missed issues. In some cases, major issues might not be found until the software has already been released to customers.

Once you have a good understanding of system-level testing, you can be confident that you are not missing any major test areas. You will also understand exactly what you need to test in each area.

This manual will be useful to you if you need to learn the basics of system-level testing, or if you need to remind yourself of these before starting a new test plan.

Note: Once you are familiar with the basics of system-level testing, you should also look up details on:

- how to write a test plan
- how to write test cases
- how to run and track test cases
- how to report and track issues

These topics are not covered in this manual.

Although this manual focuses on testing a web-based software application, most of the principles apply to all types of testing in a product development environment.

What this Manual Contains

System-Level Testing Principles – gives a definition of system-level testing and explains why it is important

High-Level Test Areas to Consider – lists the basic areas to consider when testing a web-based software application.

- **Functional Testing –** describes what functional testing is, including success paths and failure paths. Gives examples of some success paths, and lists some general failure paths to consider testing.
- **User Interface Testing** – describes how user interface testing differs from functional testing, and lists some types of user interface tests that you should consider running.
- **Data Testing** – describes the two main types of data testing you may need to consider. Also discusses why you may need test data, and how to set it up.
- **Security Testing** – describes what type of security testing needs to be done for a web application, and gives examples of some security tests that you should consider running.
- **Performance Testing** – describes what performance testing is, and gives examples of some performance tests.
- **Browser/Operating System Testing** – describes what browser/operating system testing is and gives you a general test strategy for this area.
- **Regression Testing** – describes the three kind of regression testing, and discusses when and how you should use each one.

Glossary – gives definitions of the technical terms used in this manual. Some of these terms are also italicized and defined in the text.

System-Level Testing Principles

The main principle of system-level testing: *confirming that each part of a system does what it's supposed to is only the first step!*

System-level testing means testing a system as a whole, rather than focusing narrowly on individual parts. This is important because everything in a system is connected, and if you focus too narrowly on a certain area you might miss related issues that happen elsewhere in the system.

For example, if you are testing to make sure that you can enter data into a certain field, you might miss that entering a certain special character causes corruption in the database that makes another part of the program not work properly.

In system-level testing you not only test what the program is supposed to do, but also make sure that it behaves well when things don't go perfectly. Users do a lot of things that programmers don't intend or expect them to do, and you need to make sure the program can handle these actions.

When performing system-level testing, you should be methodical but also look at the big picture. As well, you can add a valuable perspective by not only testing what the programmers have produced, but also by approaching the system from the user's point of view.

High-Level Test Areas to Consider

When you are planning to test a web-based software application, you should consider the basic test areas shown in Figure 1 below.

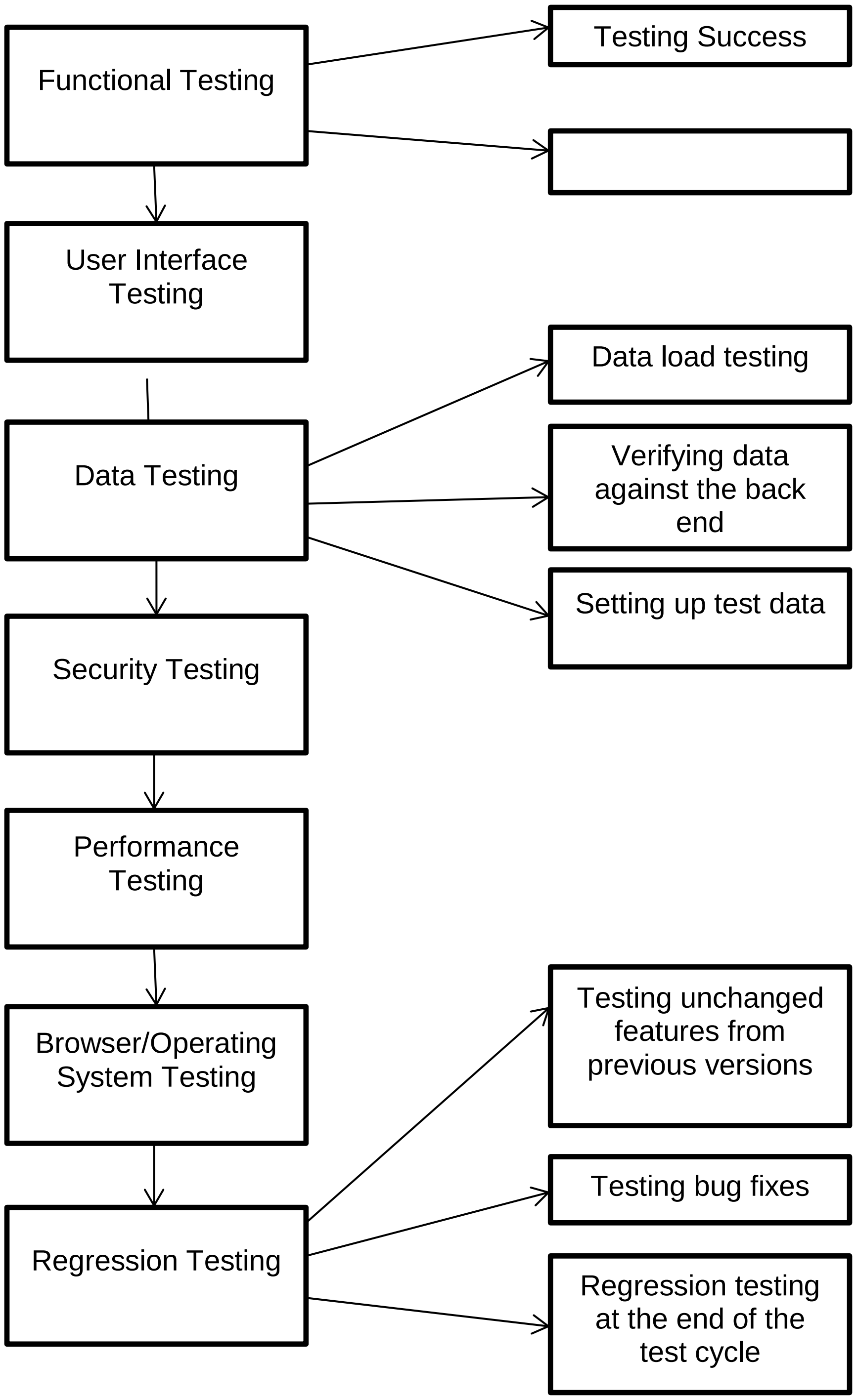

Figure 1: High-level test areas to consider.

Functional Testing

Functional testing is making sure the application does what it is supposed to do, as described in its specifications. This is very detailed testing in which you go through every possible action that a user might perform.

Testing Success Paths

"The happy path is the path through a system where everything works, the data is correct, the system stays up, and the users are well-behaved."

- Chuck Musciano

A success path (sometimes known as a "happy path") is a sequence of events in which the user successfully performs an action in the way the designer intended. For example, if Google was testing their search engine, a simple success path might be entering a search term into the search box, clicking on "Search", and getting a page with results.

When testing success paths, you should go through every action that the application designers intend to let a user perform. Table 1 gives a few examples.

Table 1: Examples of Success Paths

Action	Expected Result
Try to log in to the system with a valid user ID and password.	You should be successfully logged in.
Try to change your password.	Your password should be changed successfully.
Log in as a certain type of user.	You should have access to the expected set of actions for the type of user you are logged in as.
Click on the "FAQ" link.	You should be brought to the FAQ.
Click on the shopping cart icon.	You should be brought to your shopping cart page, containing all items you have previously added to your cart (and have not deleted).

Testing Failure Paths

A failure path is a sequence of events in which the user does something unexpected. The application should handle this well. It might prevent the user from performing the action or display a useful error message once the user has performed the action. In the worst case, the application should fail in a controlled way.

Use your imagination when testing failure paths. If you wonder what would happen if you did something, try it and see. A user is almost guaranteed to try it eventually.

Table 2 gives some examples of failure paths to test.

"Testers don't like to break things; they like to dispel the illusion that things work."

- Cem Kaner, James Bach, and Bret Pettichord.

Table 2: Examples of Failure Paths

Action	Expected Result
Enter the special characters ~`!@#$%^&*()_+-={}|[]\:";'<>?,./ into all input fields	The characters will be accepted without causing a problem, or the software will prevent you from entering them, or the software will give you a useful error message after you enter them.
Click all buttons and links on each page, even if they are not part of the normal flow of events.	They should all work as expected and not cause any errors or crashes.
Test around the boundaries of input fields. For example, for a field which should accept any number from 5 to 15, try entering 4 and 5, 15 and 16, and something in the middle (such as 7).	The field should accept numbers in its range (5, 15, and 7) The field should not accept numbers outside its range (4, 16).

Action	**Expected Result**
Test character limits. If a field should only accept 20 characters, try 20 characters and 21 characters. If a field accepts an unlimited number of characters, paste in a very large number of characters.	A field with a character limit (say, 20) should accept a number of characters up to its limit (20) but should not accept more (21). If you enter too many characters, the field should either not allow you to enter the extra characters, or provide a useful error message. If you paste a large number of characters into an "unlimited" field, the program should not break or crash.
Test that each type of user only has access to the set of actions that they are supposed to have.	A user of a certain type should not have access to actions which should only be performed by a different type of user.

User Interface Testing

User interface testing is making sure that the user interface itself is working properly. This is different from testing the underlying program, which is done during functional testing. Table 3 shows some types of tests to run on your user interface:

"As far as the customer is concerned, the interface is the product."

- Jef Raskin

Table 3: Examples of User Interface Tests

Action	**Expected Result**
Check spelling and grammar, on all pages, all error messages, and all popup dialogues.	There should be no spelling or grammar issues. If exact text was included in the requirements, it should be there exactly as specified.
Click all buttons and links.	All buttons should do what they say they will. All links should go where they say they will go.

Action	Expected Result
Make sure that all expected buttons are present on each page, popup dialogue, and error message. This includes all buttons listed in the requirements, as well as buttons that are standard for user interfaces and that are expected by users.	All expected buttons should be there, and should work properly. A popup dialogue box which allows a user to perform an action should have a cancel button, in case the user does not want to perform the action after all. A popup error message, which is just informing the user about something, should not have a cancel button. A cancel button should close the dialogue box and actually cancel the action.
Check formatting and pictures.	Nothing should look strange or out of place. All pictures should load properly. No text should be cut off on a button. If it is, you should be able to hover over the button and see the full text in a tool tip.
Click on all drop-down lists.	All drop-down lists should have the correct options and no extras. You should be able to successfully select each option from a drop-down list. No text should be cut off inside a drop-down list. If it is, you should be able to hover over the list item and see the full text in a tool tip.
Read all labels and instructions.	Labels and instructions should be clear and understandable. Nothing should be confusing, misleading, or hard to read.

Action	Expected Result
Look at text size and contrast.	Text should be large enough for the average user to read comfortably, and should contrast against the background.
If the application is designed to be accessible to users who can't read the screen, load the page with a *screen reader* (a device that describes what is on the screen).	The screen reader should describe each graphic, button, and field so that a user can understand and use the application.
Use the keyboard only (no mouse) to navigate through the application.	You should be able to get to every field on every page in a logical order, using only the keyboard.

Data Testing

Data testing is making sure that data is moved around the system correctly. There are two main kinds of data testing:

1. **Data load testing** – testing to make sure that any data loaded into the system during *deployment* (when the application is installed or upgraded to a new version) is loaded correctly.

2. **Verifying data against the back end** – testing to make sure that data is moving correctly between the application and any *back-end databases* (data storage areas not visible to the user).

This section also discusses how to set up data for testing purposes.

Data load testing

If data will be loaded into the system when the application is deployed, you will need to make sure that it will load correctly. This involves making sure that:

- All necessary data is loaded.
- No extra data is loaded.
- The data that is loaded is correct.

There are two main ways of accomplishing this.

1. **Spot-checking.** Check some representative sample data. It is a good idea to also check the total number of records in the database, to make sure the correct amount of data has been loaded.

2. **Reviewing everything.** Retrieve all of the loaded data from the database and compare it with the data that was supposed to be loaded. This takes longer, but is more accurate.

Figure 2 below shows how to get both sets of data onto your computer so you can compare them.

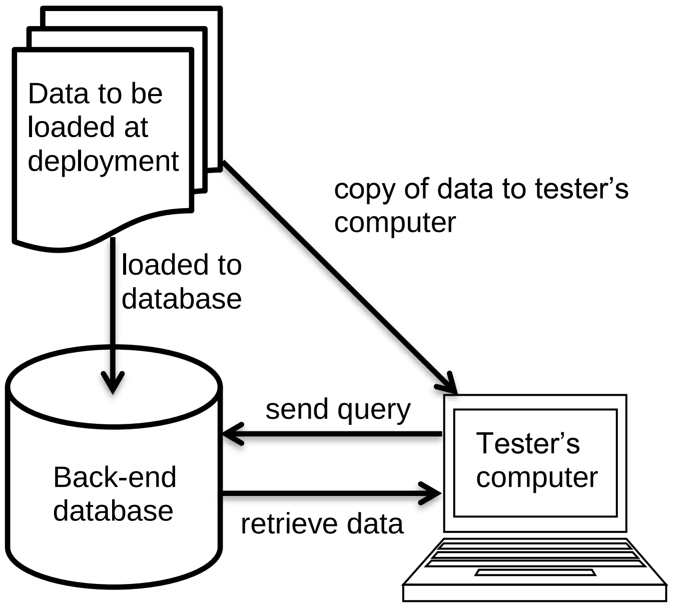

Figure 2: Comparing the planned data load with the actual loaded data.

Verifying data against the back end

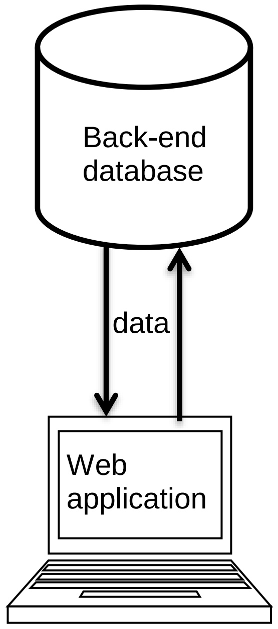

Figure 3: Normal flow of data.

You need to make sure that data is moving correctly between the application and the back-end database, as shown in Figure 3. To do this, change some of the data in the database and make sure you see the change in the web application. You should also test that if you change the data in the web application, you will see the change in the back-end database.

You should do this for each type of data in the database, and for each method of changing the data in the web application.

Setting up test data

When testing a web application, you will often have to set up test data that you will need while running your test cases. For example:

- You may need to set up some test users that have features you want to test. For example, you may want to run tests while logged in as a user with a certain account type, or of a certain user type.

- You may need to create some records with certain features. For example, if your application classifies records into different categories, you may need several records from each category.

- You may also need to change test your test data while you are testing. For example, you may need to set a field in the database to a certain value to lock out a test user’s account. If you later need the account to be unlocked, you will have to change the value in the database again.

You can sometimes create and modify your test data using the application itself. Other times, you may have to add test data directly into the back-end database in the test environment.

Security Testing

Security testing is making sure that only authorized users can access the application, or particular parts of the application. You will have to do this kind of testing if users have to log in to the application to access all or some of its features. Table 4 gives some examples of security testing:

Table 4: Examples of Security Tests

Action	Expected Result
Try to log in with the wrong password.	You should not get into the application.
Try to log in with a non-existent user name.	You should not get into the application.
Log out of the application and then use the “back” button on your browser.	You should not be taken back to the page you were on and re-logged in. Instead, you should get a message telling you that you will have to log in to access the page.
Delete or freeze an account and then try to log in to that account.	You should not get in to the application.
Try the maximum number of incorrect login attempts.	The user’s account should be frozen for a specified length of time before the application allows the user to try again.
While logged out, try to go to a page that you have to be logged in to access. Paste the URL of the page directly into your browser.	You should get an error message telling you that you will have to log in to access the page.
While logged in, try to go to a page that you do not have permission to access. Paste the URL directly into your browser.	You should get an error message telling you that you do not have permission to access the page.

Performance Testing

Performance testing is testing the system under heavy loads to make sure that it doesn't break, or if it does break, that it fails gracefully with a warning to the user.

You will probably have to use simulation software to do some of your performance testing.

Table 5 gives some examples of performance tests:

Table 5: Examples of Performance Tests

Action	Expected Result
Simulate a large number of users accessing the system all at once.	The application should respond normally to users without slowing down. If the application does slow down, it should display a message telling users that the system is experiencing delays. No user requests should be lost, even if the application has slowed down dramatically.
Upload a large amount of data while logged in as one user. On another computer, simultaneously log in as another user.	The other user should not notice the application slowing down during the first user's data upload.

Browser/Operating System Testing

When testing a web application, you should make sure that it works properly using the major web browsers and operating systems that users are likely to be using. Don't forget to include mobile browsers if your users are likely to be using smartphones or tablets.

Do a little research to find out which browser/operating system combinations are used by most of the intended users of the application. You don't have to test every possible browser and operating system—just those which a significant number of your users will be using.

You also do not have to re-test the entire application with every browser/operating system combination you are testing. Pick one main browser/operating system combination and use it to run most of your test cases.

Then go back with the other browser/operating system combinations you are testing, and do the following:

- Test a selection of success paths from your functional test cases.
- Run a selection of test cases from your user interface testing section.
- Test that all buttons and links work.
- Test user actions that might differ between browsers. For example, loading pages, selecting options from drop-down lists, viewing pictures, viewing text formatting, and sending system-generated emails.

Regression Testing

Regression testing is re-testing areas that have already been tested. This is done to make sure that nothing has been broken since the area was tested.

Testing unchanged features from previous versions

This type of regression testing is done when the software you are testing is an upgrade to an existing application, rather than an entirely new application.

Often when changes are made to one area of an application, other areas are accidentally broken. Regression testing is used to make sure these issues are found before the software is released.

This can be done using high-level spot-checking of the basic functions in the area you are testing. It helps to have a copy of the test plan from when the area was first tested.

Sometimes this is done using an automated test suite—a set of test cases that were previously programmed into an automated testing program. At the end of your test cycle you might add some of your new test cases into the automated regression suite for future regression testing.

Testing bug fixes

After you have reported an issue that you found while testing and it has been fixed, you will need to retest it. There are two parts to retesting a bug fix:

1. **Make sure the issue you reported is actually fixed**. Go through the steps you described in the *bug report* when you reported the issue, and make sure that the issue is no longer there.

2. **Make sure that the fix has not broken anything else.** Do a quick check of surrounding and related features, to make sure that nothing obvious has been broken. You will be doing a more thorough check at the end of the test cycle.

Regression testing at the end of the test cycle

Once you have finished all of your testing, including retesting all bug fixes, you will need to run a set of regression test cases.

You should select these test cases from the set of test cases you have just finished running. Choose a set of test cases that cover as much of the application as you can in as few test cases as possible. Pay extra attention to any areas of the application that had a lot of bug fixes and updates during the test cycle.

This set of regression test cases is run to double check that nothing has been broken by updates and bug fixes during the test cycle.

Glossary

application – a computer program designed to perform a particular task.

automated – done by a machine rather than a person.

back end – part of the application not visible to the user.

break – to get into a state where something that was working before is no longer working properly, but the entire program has not crashed.

browser – a program used to access web pages.

bug – a problem found in a computer program.

bug fix – a small software update that fixes an issue that was found and reported.

bug report – a report written by a tester who has discovered an issue in the software they are testing. It describes the problem in detail, including a step-by-step recipe to reproduce it.

crash – to get into a state where the entire program freezes or shuts down.

database – a structured collection of data.

deployment - when an application is installed or upgraded to a new version.

drop-down list – in a graphical user interface, a list of choices that drops down if you select or hover over a menu item. You can move the mouse or keyboard down the list and select one of the items in the list.

field – the smallest unit of storage in a database. A single record in a database can have multiple fields.

operating system – a program that manages all of the other programs on a computer.

query – to send a command to a database, such as a request to retrieve data.

record – a piece of information stored in a database, made up of the information stored in one or more fields.

release – to make the application available to customers.

representative sample – a smaller number of items that accurately reflect the whole population.

screen reader - a device that describes what is on the screen. This kind of device is used by people who cannot read the screen themselves.

simulation – a software model that behaves as if it is the actual process it is simulating.

specifications – detailed requirements and descriptions of how something will be made and what it will look like.

test cycle – the period of time when you are actually testing an application. If the software is still being developed as you are testing, you may have multiple test cycles before it is ready to release to customers.

test environment – a server where you can test a web application without it being visible to anyone else. Includes test versions of the web pages and any back end databases.

test case – a step-by-step checklist describing how you plan to test one thing. Each test plan has many test cases associated with it.

test plan – a document which describes what you are planning to test, and gives a high-level description of how you plan to do it.

tool tip – a box with text in it that appears when you hover the mouse over an element in a graphical user interface.

URL – Uniform Resource Locator. A web address (for example, http://www.google.ca)