10/28/22

**Unit testing**

unit testing is intended to:

* make sure a routine works as intended
* verify data combinatorial testing without using the UI
* ensure that a function works according to specification

I usually prefer to **start small, with one repo**, well structured. Creating many repositories from the beginning makes maintenance unnesessary complex. When the repository starts growing, **you will need to refactor it from time to time**, update the structure. If refactoring is not enough then you can think whether to move some tools or tests to separate repos.

**#** [Automated testing: focusing on testing with no dependency vs. workflow](https://sqa.stackexchange.com/questions/6363/automated-testing-focusing-on-testing-controls-vs-workflow)

I'm in charge of **regression automation** on my team, and I need to add some new tests into our regression. Before I get started, though, there's an issue I'm struggling with: where is the line between focusing on **automating user workflow** through a system, and focusing on only ensuring that **the controls are working** as they should be?

For example: imagine a system can set up new users, and later edit their information. On the screen that accesses users, there's a list of recently added users at the top, and there's a search function at the bottom. The **user** **workflow of a client using our system** would be to create a user, then access this screen to edit the user's information immediately afterward. I could write my automation that way, but I would like to **write short tests that only test one thing each**. I'm trying to stay away from behemoth tests that take a long time to run and test many things, and when one fails you need to wade through many many checkpoints to see exactly which one failed - this is inefficient. A test that follows this workflow would have to either test two things (creating the user, then editing the user's information), which would make it one of those behemoth tests I'm trying to stay away from, or I would have to write two tests that have a dependency between them (Test1 creates the user, Test2 depends on the user being on that screen to click).

It seems to me the better solution is to **write a test that focuses on creating a user** and **write a separate test that uses the search function I mentioned to look for an existing user** (**we always start our tests with a pre-set database of users**) to edit their information. The problem is that this tests purely the controls, not the workflow. I could always throw in a third test that tests the functionality of the recent users screen, so that's covered by the automation too. It seems to me that manual testers should focus more on workflow, whereas my automation should focus more on purely how the controls work. Is my reasoning correct?

Welcome to SQA, Ana. I see at least two questions:

* **Does it make sense to distinguish between focused tests and workflow tests?** Yes, it does. A focused test concentrates on a particular feature, including things like edge conditions. It attempts to answer the question, "Does this feature work in all circumstances?
* **Functional testing has overhead, which is setup and teardown for providing appropriate test environment**, **but not focusing on testing the setup env**. Ex:
* **To test login feature,**
* **the setup will be create a user with valid username/passwd;**
* **run login test** with all combination of usernames and passwds ;
* **Teardown will be removing the created user.**
* A workflow test verifies that the system as a whole is usable for its intended purpose. Both kinds of tests are necessary, but it is a better use of time to execute them separately.
* In my team we design our "**regression tests**" with a broad scope to cover as much of an **end-to-end workflow as possible**. **We run regression tests on every new build**. This means the tests are more complicated and there could be several points of failure which is why ensuring the machine is in the appropriate state during test execution and verbose logging is critical. Also, time is a factor, but our none of our tests take **NO more than** **1 minute** to complete, and **most e-2-e scenario tests execute in a few seconds**. (To be transparent...my team tests a set of APIs that transports and parses data between different clients and different services so our end-to-end workflows are essentially without a UI and mocking (or sometimes using real) web services.)
* **We also design a separate set of functional tests** (not regression tests) that focus on the functionality/capabilities of **individual APIs** or components. This can be used as during new feature verification in the work time, and will be integrated to regression test later. These tests are intended to hit edge cases, error conditions, use malformed data, use variable data, or inject systemic anomalies to expose weaknesses in the system. Our functional tests tend to be more discrete or smaller in scope because we are trying to find previously undiscovered bugs, and test the robustness of that particular component is a variety of conditions.

I am an Android developer but I am not satisfied with my testing and Continuous Integration environment and configuration.

Someone could recommend me how to configure the best setup and workflow for a good CI in Android?

That includes:

* **Automation build tool**: Maven, Ant, Graddle...
* **Testing tools and libraries**
* **Testing configuration**: Where to put unit tests, integration tests...
* **Test coverage**
* **Continuous Integration server**
* **Other QA tools and their configuration**: FindBugs, PMD...
* **General workflow**

 **Automation build tool**: I use both [Maven](http://code.google.com/p/maven-android-plugin/) and Ant. My first idea was using only Maven (I like how clean and structured it makes your projects), but seeing that Ant is the automation build tool used by the Android team and that I could not achieve some things with Maven, I also added Ant. More on that later.

 **Archetypes**: I used the [Maven Android Archetype](https://github.com/akquinet/android-archetypes/wiki) *android-release-archetype* to create my project, and modify the result to include the Android Library Project I am developing. So, I have a Maven parent project with three submodules: Android Library Project, Android standard application and Android Test Project (instrumenting the Android application).

 **Testing tools and libraries**: Standard JUnit, Android Instrumentation, Robolectric and Robotium.

 **Testing configuration**: I have the tests made with Instrumentation and Robotium in my Android Test Project and the JUnit and Robolectric tests in the same project I am testing the code. The reason for that is that I could not get Robolectric to test code in another project, but I would prefer to have all tests in the Android Test Project (I don't know if that is a good idea, but it seems to me). Also, I still don't know (I haven't had the time to look for it) how to tell Maven or Ant to execute a subset of the tests.

 **Test coverage**: I have not been able to generate a test coverage report with Maven and that is the reason I finally also added Ant (you can get test coverage with Ant following the next instructions: [Building an Android app and test project](https://wiki.jenkins-ci.org/display/JENKINS/Building+an+Android+app+and+test+project)).

 **Continuous Integration server**: Jenkins and the [Android Emulator Plugin](https://wiki.jenkins-ci.org/display/JENKINS/Android+Emulator+Plugin) (although I think it has some issues with last versions of the SDK).

 **Other QA tools and their configuration**: I have not looked at it yet.

* **General workflow**:

1. Jenkins polls for changes in the project every 5 minutes and builds everything if it finds any.
2. Also, I have nightly builds no matter what.
3. Regarding my development workflow, I just run all the tests from time to time during the day and always before committing.
4. I don't find that ideal because **the tests in the device (Instrumenation and Robotium) take a long time**, so I would prefer instead to run only the deviceless tests while developing, and all of them before committing.

The technical testing team has a formal process where they write the test objectives and test cases prior to developing the script in SOAPUI. Therefore, analyzing the specification and supporting requirements document are important for deriving the tests.

**The API is not an open API, only for customers, third party suppliers and internal web and app resources**.

In a TDD/BDD approach we should write the unit test case first and then write the development code which will pass the test. That's becoming a standard in the world of devops.

Now the important part here is the code coverage and the functionality developed. Now as a developer you should ensure that the unit tests give 100% code coverage.

The problem that I've seen here is developers do not write the whole spectrum of tests possible**. If there are 10 possible unit tests for a certain requirement and the developer has written only 5**. Then it's a miss from his side. For example, if the requirement is 'Given I'm an admin, I should be able to login to the application'. The tests could be:

* If correct username password the user logs in and displays a success message.
  + Incorrect password the user could not login and displays an error message.

Now if the developer writes the test for only successful login. Then the unit tests would pass. **Code coverage would be 100%**. **But there's a missing functionality**. What about the incorrect password scenario?

So a QA is definitely required along with unit tests, code coverage tools.

Another consideration is that if you just sit down and bang out unit tests based on the code, without the detailed study of the specification and requirements that you mentioned, **you risk testing what the code does, rather than what it's supposed to do**.

Yes. It is definitely a good idea to "keep all methods, verifiers of a particular page in a separate class." One way of doing this is by using the [**Page Object test design pattern**](https://www.selenium.dev/documentation/en/guidelines_and_recommendations/page_object_models/). It is a good way to make test code more robust, readable and reusable.

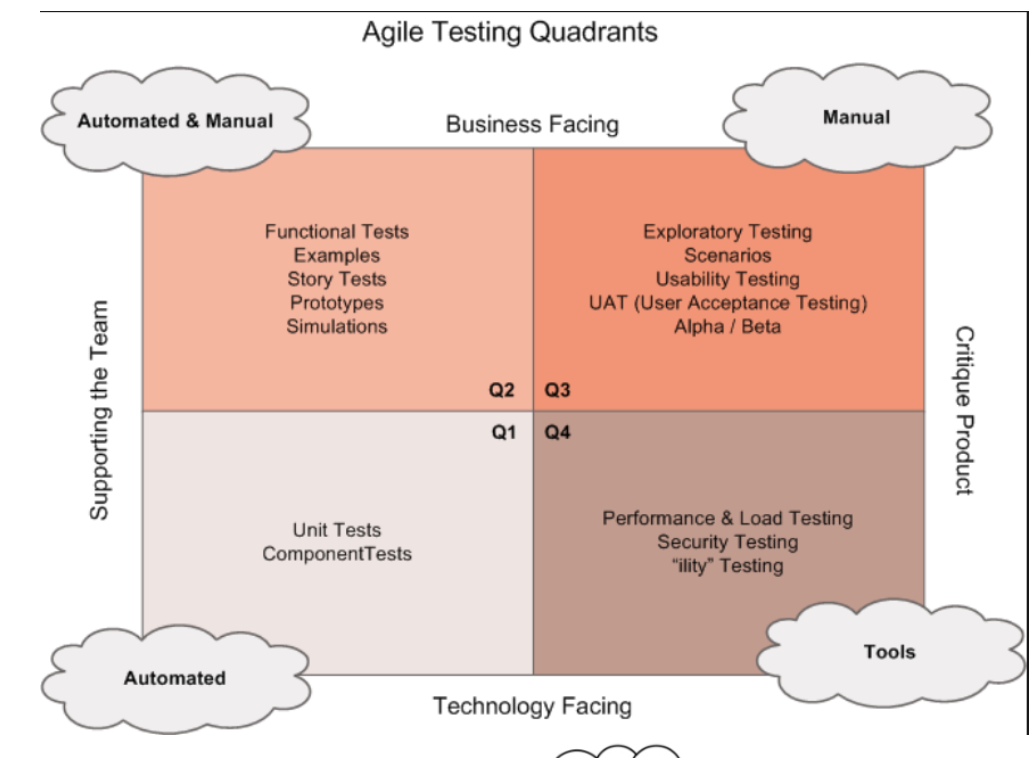
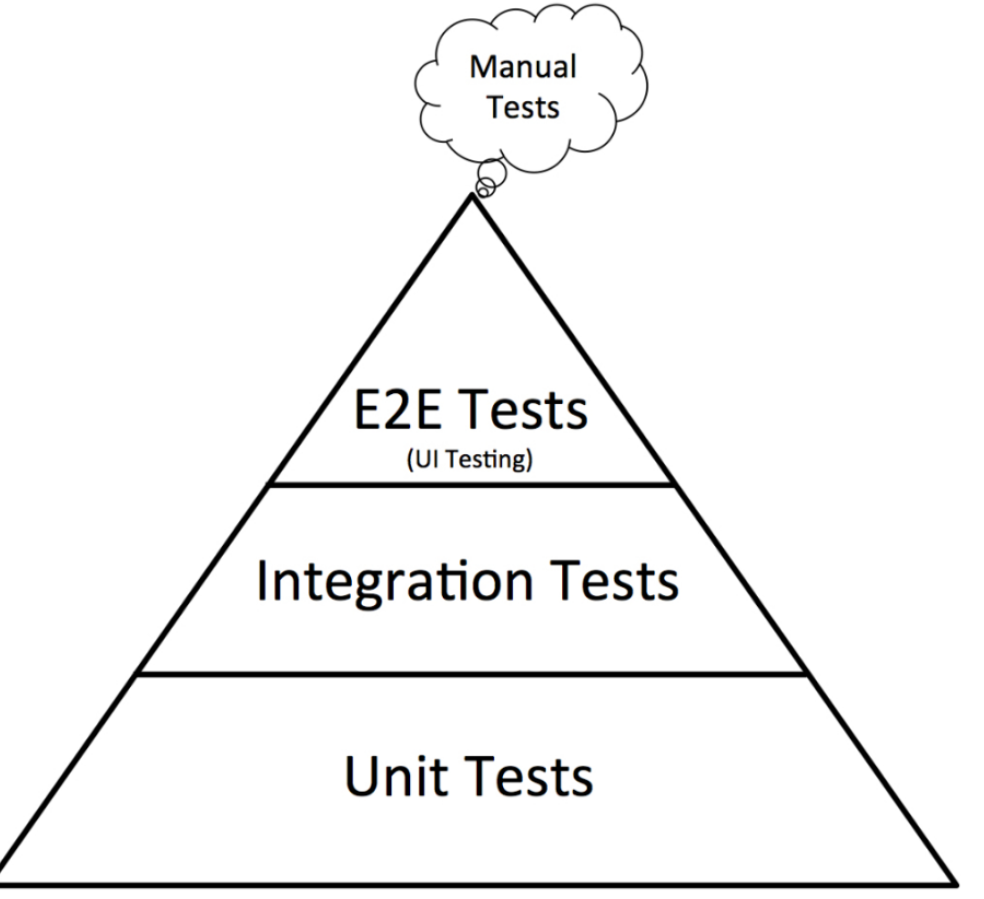
* Use **PageObjects pattern**
* Be fluent with
  + reuse your model.
* **Be robust and portable**
  + Preferred selector order : id > name > css > xpath
  + Avoid sleep prefer explicitly\_Wait or implicitly\_Wait
  + Use relative URLs
  + Don’t rely on specific Driver implementation
  + Create your dataset
* Know your new tool
  + Keep up to date (versions and usage pattern)
* I tend to spend my time working on tools to help the testing process and smoothing the edges of the tool.

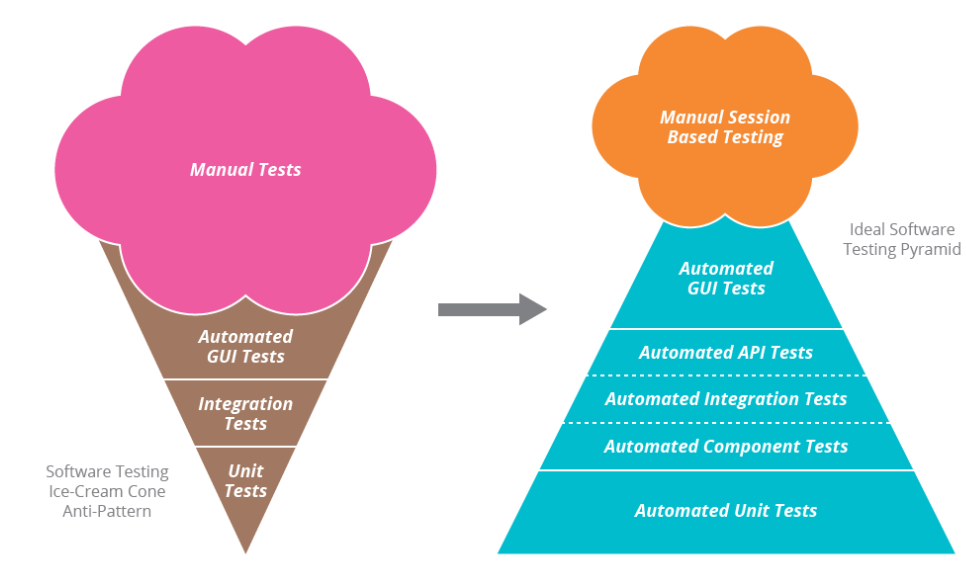
Here are some **cons** for shell scripting that would make me take my decision towards "compiled" case:

* Shell scripts **are more dependent on the execution context**. So this impacts the portability. It might happen that the required tool which is used in the script is not installed at your new place. When you write your script in programming language you can easily package all you need to a single executable file
* It is harder to write well-structured code in shell scrips which also means it is really hard to extend and maintain such code.
* It is harder to debug shell scripts
* Shell script logic cannot be well parallelized
* Shell scripting has poor IDE support
* Shell scripting has poor error handling mechanisms
* **Since there is no compilation phase** you will know about some bugs only when your script will crash.
* Not for test scripts but anyway (shell scripts are more sensitive to malicious impact)

Here are some **pros** for shell scripting:

* Shell script takes less resources (but it is not always true)
* Shell script **does not require compilation** phase so that you can just go to where your script is deployed and fix the issue by editing file
* Shell script does not require any additional software to interpret the commands

  **Background/Problem Statement**: Recently I came across a project team, which is having thousands of UI automated tests **covering all kind of manual tests** including positive, negative etc., and **obviously struggling to maintain them with every release with very high execution time**. Basically [ice-cream cone anti-pattern](https://watirmelon.blog/testing-pyramids/). There is a rest API based backend to support multiple application platforms.



1. Estimate working load, how many test cases are there to be migrated from UI tests?
2. Estimate how much time you can spare in each sprint and discuss with your team leader and your fellow members regarding you want to spare some time in each sprint to work on test case migration.
3. Prioritize automated UI tests, there is a chance a portion of them are no longer needed. Migrate highest-prioritized tests first.
4. Based on your discussion from step 2, allocate work load to yourself during each sprint.
5. Keep old UI tests running even after they have been replaced, phasing them out slowly. You may not be the only person who is running them, keeping every stakeholder updated.
6. Document your approach as there are thousands of UI tests, it is unlikely someone will come in to help you or you will leave before you finish; good documentation will help others continue working on test case migration.
7. Introduce some **proxy** between your front-end and back-end that would log the requests to your Back-End
8. Run all your tests for UI
9. **Parse the proxy logs so that I have all the endpoint calls grouped by the endpoint addresses** (and possible the parameters - depends on what cost of the migration we can afford)
10. Having all the things grouped I would spend significantly less time to implement the API tests. Probably I would even introduce the converter that would build the tests automatically basing on the grouping obtained on the previous step.
11. Since front-end usually extends the functionality, when we convert from UI to back-end the result would likely be more compact. If we're talking about REST API then **there is just a limited number of endpoints** which are invoked by your front-end. Anyway whatever API it is, we can capture the calls from "UI" and parse in order to build API auto-tests
12. if the api was internal/http, you could run tcpdump on the proxy, load it in wireshark, extract HTTP requests+headers, and either replay them or port them in postman.
13. If you are looking for a quick win, I have come up with an idea for [converting selenium tests to API tests](https://blog.loadmill.com/convert-flaky-ui-tests-to-blazing-fast-api-tests-d71fa4702161). This will only work for cases in which you have "UI" tests that have nothing to do with UI logic and are only used to test backend logic.
14. The idea is to [capture the network traffic when running a selenium test](https://www.npmjs.com/package/har-recorder) in a HAR file, and covert this HAR file to a replayable API test - which involves identifying the request-response correlations and adding assertions.

I have to integrate API test-automation on a project that has a microservice architecture. Each back-end service has a different repository and is deployed separately. I haven't had the opportunity of implementing test-automation on a microservice architecture before, therefore I am not sure how to approach setting up the test data.

Let me introduce you to my dilemma: let's say we have a service called Categories and a service called Subjects, both services have create and delete operations, each with its own API endpoint. But here is the pickle, in order to create a *Subject* using the Subjects service, a *Category* already has to exist, in order to be selected. But since a microservice architecture is in place, I don't have access to the Categories API endpoints in the Subjects service, so I cannot create a *Category* while the tests for the Subjects service run in the Subjects repo, which leads to my question: **what approach should I opt for in order to provide the required-dependent test data?**

In this case, your options are limited to:

1. **Getting access to the Categories service and creating a test entity using it**, or
2. **Writing to Database directly**

It is hard to tell which option is preferred because the choice would depend on many factors.

I have been in similar situation in my qa services project. API are being developed, (**also getting updated frequently**) and **UI does not exist** and we are doing a minor release based on API testing.

1. We need to draft **E2E flow cases**. (high level not very formal)
2. Running **the API end to end flow test is always useful. Ultimately we want the E2E flow to behave correctly**. It also covers the unit tests
3. If there are any issues in E2E flow, we can identify the failing API and update the unit test for it. This approach worked for us.

I test the database using a disposable Oracle Docker container. I need to figure out how to test the JSON message that goes from the browser to the server.

Project has **one public api** that makes some data available. So I will **automate API tests**. **Tests will live in one repository**. This part is clear to me, I did this before in other monolith projects. I had **one repo for API tests**. And possibly **another repo for selenium tests**. Should be in the same depo, just different folder.

But then here the whole poduct consists of many microservices that communicate via restful apis and/or rabbit queues. How would I go about automating tests for each of these individual servicess? Would tests for each individual service be in a separate repo?

* **services are written in Java or PHP**. I will **automate tests with Python**.
* Product is being developed by one team of 6-7 devs and 1 qa (me).
* We are planning to implement CI and CD also. On **pull request** components should be deployed to a **test env/ test server** and my tests executed there. Test can be run on ci/cd server, while regression runs on a separate test machine.
* Then if successful components will be **deployed to staging env/server** and some smaller set of [smoke] tests might be executed.
* In a case of a successful deployment to stag env and successful smoke test execution components might be **deployed to production**. All this done by CI/CD tools and tests that Im about to write.
* Hopefully the Developer teams know their responsibilities i.e. having high test coverage at Unit and Integration level (Test pyramid). Assuming that, you can focus on high level testing i.e. **end2end testing and contract testing**.
* Contract Testing: A very good practice in micro-service architect.There are some solution for it. One of the most suitable one is a consumer-driven contract testing called "[pact](https://github.com/realestate-com-au/pact)".
* If possible, **keep your Test Automation in the same repo** since you may need Mocking/Stubing plus versioning can be tricky. Also staying in the same repo **could** be a help for you CI/CD pipeline.

# [Test automation for microservices architecture](https://sqa.stackexchange.com/questions/17487/test-automation-for-microservices-architecture)

I am in charge of implementing QA processes and test automation for a project using microservices architecture.

**What to test and how**

* You definitely need to test full flows of the system, to see whether services understand each other, in particular, when one of service responds with error messages. This is a place where I usually find problems. **Mocks will definitely help here**.
* Also, if you're working with queues, then many there might **asynchronous events and time-outs** that may come into play. You can try simulating each such scenario individually but performance tests when many concurrent threads are loading one of queues with unexpected number of messages can reveal interesting bugs.
* Many scenarios are easier to automate at unit testing level, where you do not need to start an HTTP server to host a service or service mock. This is particularly the case for negative scenarios. I usually work closely with developers of such services and every time I find automating a scenario is to expensive at system level**, I ask developers to add a test to their unit testing suite**.
* Martin Fowler identifies [multiply levels of testing for microservices](http://martinfowler.com/articles/microservice-testing/): unit tests, integration tests, **contract tests, and end-to-end tests**. However, make sure you're really having microservice architecture there. I've seen many microservices ending as **distributed monolith, sharing code or even sharing databases**.

# [How to test a dropdown that depends on the value in another dropdown](https://sqa.stackexchange.com/questions/25770/how-to-test-a-dropdown-that-depends-on-the-value-in-another-dropdown)

I have two drop downs. One of the **drop down** list is dependent on the Selected item from from other drop down.

I need to verify data in the dependent list is correctly listed based on the selected value. (note:- Both lists are dynamic and gets updated every week)

Eg: IF I select Truck, dependent drop down should have all small parts of truck.

The data coming in the dependent list is not always updated in the application I am working on.

Also the application has different filters, so need to always verify returned data is accurate.

Assuming the app exposes an API, there should be API tests. You need to test the API to have confidence that that layer works as expected before moving up a layer and testing the UI. What you have pointed out is that there is a problem, what you don't know (yet) is whether the problem is API related or between the interaction between the two components in the UI.

For the API tests, I'd suggest using SoapUI or Postman. With these tools, you can call the API, store the response, then call the db and verify that the response contains the expected db items. Take a look at Asserts. If the API response doesn't match the expected items from the db, then the API test is a fail. You can also trace how long these API calls take, which might help when testing the UI....

In terms of the UI test. Is the speed of the test too fast for the interaction between the two components? Have you tried manually testing with the console window open to see if there are any logging messages? This might give you a clue as to any issues between the two components. In the UI test, you can pause whilst you wait for the parts dropdown to load after the vehicle dropdown has been clicked.

1. **Data validity**: Testing the question "Do I get back correct list of small truck parts when input is **'Truck'**?" **Here you focus on the output correctness**: does it contain all parts? Does it contain some parts that should not be there? Is some answer provided for all possible inputs? **Ideally, you should not test it through UI using Selenium, but via API**. Even better if even filtering is provided by the data source. Of course, this depends on how the data gets into the dropdown.
2. When the bulk of the data correctness is tested directly, you can then just check, using Selenium**, if the data is correctly presented in UI**: if the second dropdown eventually fills, if it's filled with expected content, etc. Because you have confidence in the underlying data source, you need fewer tests over the UI (they are notoriously flakey and slower).

# [Is it necessary to verify databases for functional automation testing?](https://sqa.stackexchange.com/questions/6757/is-it-necessary-to-verify-databases-for-functional-automation-testing)

Currently, after every action that adds or changes data, **we *verify* that that data was successfully added/changed in the database by querying the tables directly**. I feel this is redundant because that data is already being pulled by the application and is displayed correctly (or incorrectly) in the application. Wouldn't any data inconsistencies expose themselves in the application itself?

This is the list of the issue, but I will provide a simplified test case to expand on what I'm talking about:

1. Go to user "John Smith" profile and update his phone number from "(999) 999-9999" to "(666) 666-6666" and click the save button in the application
2. Verify application displays "Successfully changed user profile"
3. Go to preview profile and verify John Smith's profile reflects the changes
4. Verify table USER\_PROFILE in DB users database updated with changes

Is it necessary to check the table in the database if we've already verified the application is displaying the data correctly? The application is ALREADY pulling the data from the database table, so obviously if the application is displaying the data correctly then the data in the database is correct. Am I missing something?

**Reasons to check DB, besides UI testing**:

* You have a **bulk update/insert function** where **it's impractical to validate the results via the front end**, such as importing new user records from a CSV file. While you can go in through the front end and check that each record you imported is present and correct in the GUI, it's **a LOT faster to do a bulk database read and compare against the expected results**.

**Not all fields in the database are exposed to the front end**. Calculated fields, timestamps and the like all need to be checked

* **If the UI doesn’t display all the relevant data that DB holds**, **there could be incorrect data stored in DB**. For example, inventory of a book is 10. UI requests to buy 1, results show UI purchased one and confirmed the transaction goes successfully. But UI was not required to show the inventory again. And there is a possibility that the inventory was updated incorrectly, and UI didn’t verified for the updated inventory after the purchase.
* Your database storage uses dependent records such as storing all addresses in a separate table. Here you would need to check that for a delete, the dependent records were also removed, **something that is not going to show up in a front end check**.
* Where the application uses the database to **store transitional records**, you may need to check the relevant tables after each step to ensure that the correct data is stored and cleared. The example I know best is purchasing items with a limited quantity. If the quantity available is 30, and someone adds 5 to their cart, that quantity 5 must be held and quantity available decreased by 5 until either purchased (in which case the 5 held is cleared and the quantity sold increases by 5) or the purchase is cancelled (in which case the 5 held is cleared, and the quantity available is increased by 5). **Without those checks, problems with clearing available quantities may not become evident until a later purchase** - particularly if the transitional records include dependent records.

# [Controlling database state in Selenium UI Testing](https://sqa.stackexchange.com/questions/12218/controlling-database-state-in-selenium-ui-testing)

**We have a big test suite with around 300+ tests. Nowadays we are manually restoring database and creating few Data Objects and hard coding some data to feed into the tests**. However, there are a lot of tests that need more data and we are trying to **generate Data AT run time** with tests. Problem is when database changes or the test suite run on a different machine with a different version of the database they are failing.

1) We **reload (through sql script) a skeleton database at the start of regression test run**. This has a minimal set of data, as little as possible - a user, minimal config. At one point we had an automated suite to generate this database regularly to ensure it stayed in sync.

* We **don't reload the DB for each test**, because we don't want to slow down our test run.
* At the time we started the framework, the underlying database was changing quite rapidly, and writing SQL scripts to modify the db directly would have required a lot of maintenance. It is still changing, just not as rapidly.

2) **The general approach is that tests create the data they need via the API** (e.g. customers, orders, etc), and if they make breaking changes (e.g. to config), then they need to undo them at the end of the test. **Being able to use the API to set up data rather than the UI** makes this much faster to do at run time. We also know the data will be representative, as that's how the UI creates the data (for the most part) so we don't have to worry about our test data gradually diverging from real live data.

**Requirements**:

* Minimal dataset needed for all the tests (Generated from **SQL scripts**)
* **Functions to generate users and test-data from code**, preferable re-using the actual internal classes of the application. (To make sure when the application changes, the generation of data also changes)
* **Test classes has setup code that describes, sets up and tears down the actual test-data**
* Testsets are generated into a backup file and the code checks if the backup is still valid based on a delta number which indicates the db version (updated by the developers for each database change).

**Each test run does the following**:

* Test initialize
  + Check if the backup datasets are still valid and generate new data if needed
  + Generate a unique id for the test
  + Restore a testset into a db with the unique id in the name
  + Configure application/test to use the correct DB
* Run the test
  + Arrange (login, prepare extra application state, etc)
  + Act (Actions to test)
  + Assert (Verify actions are successful)
* Test clean up
  + Remove DB
  + Check logs for errors

Restoring the database takes less than a second on our SSD's, not too much overhead here. For **datasets that takes long to generate from scratch**, we save the appended dataset to a separate backup file and restore it for each test run that needs it.

**The main concept** to get used to is having **an empty database or with minimal essential datasets before EACH individual test**. Only create the reference data you need for *that* test and delete it after that test. For examples for zip code lookup we have 20,000 records but for testing we need ONE - 90210 - so we just create that one record for the test and delete it afterwards. This approach with no seed data is a big change from the past. Some folks need to see it to really believe it. It came about 'cos we learned that all the other approaches don't scale or maintain well.

**Think about the three steps for every test**

* **setup**
* **execute**
* **teardown**

Ideally this is done for every test and the database strategy used between tests is truncation.

Frequently this is deemed not possible and seed data is used which would slow down the tests too much if deleted each time. **In those cases each test should use the database strategy of rolling back the transactions it create**.

It's hard to provide input on your specific processes, but here's how I generally deal with Selenium automation. Maybe it can help you out.

1. One separate database which contains all the data for the tests: data-driven testing. This means you'll likely have to compose Views (combining tables as necessary) which serve as the data source for test methods.
2. Insert/Update/Delete scripts as a stored procedure on the application's database. Have your test class execute these scripts before running any tests. This way you force the 'variable' data in the application to a specific state which your automated tests expect.

Explanatory example:

1. For a test method 'Customer login', the Selenium database provides customer@login.com and password 1234. So the test method expects a customer profile screen after logging in with this data (if not it will fail).
2. The insert/update/delete scripts are executed before running the case. It will (1) delete the existing customer@login.com profile in the application database, then (2) insert the profile with the correct password and settings. This way **we know for sure** our test data is correct. If we don't execute this script, the profile could've been changed by manual tests, the password can be obsolete, ...

* Thanks for your valuable answer. I sort of do this now. The problem with this though ours is a payroll application and a lot of data we need in order to replicate user behavior. If I want to write my insert/update scripts, seems like it would be lot more work than just the actual writing the test since the sql queries sometime will be massive. So I wanted to know a database known state as baseline and take it from there. – [saifur](https://sqa.stackexchange.com/users/8398/saifur" \o "325 reputation) [Feb 21 '15 at 15:01](https://sqa.stackexchange.com/questions/12218/controlling-database-state-in-selenium-ui-testing?rq=1#comment13445_12223)
* 1

@saifur in that case I would recommend taking a database snapshot which you can easily (manually) restore before starting the test suite. – [FDM](https://sqa.stackexchange.com/users/8869/fdm) [Feb 21 '15 at 16:34](https://sqa.stackexchange.com/questions/12218/controlling-database-state-in-selenium-ui-testing?rq=1#comment13446_12223)

In what circumstances we should use selenium?

I am new to selenium and I am using selenium to do the testing but I am confused right now.

For example, the website I am working on contains lots of data, however all of them are from database. Users need to click some buttons or select values from drop down list, then the front-end with send request with these parameters. Database will return different values due to different parameters. After front-end receives these values, front-end will process them to graphs or show them directly.

I used selenium to complete the click or select parts, but how can I verify the data come from the database? use SQL with selenium code? But I heard that selenium should not do this. I thought about data-driven, but it seems the only part i can use is login part. Can someone tell my what kinds of things that selenium should test? Or in my case, selenium is not a good choice at all.

Selenium is for browser automation, Not a test framework for webpage testing; although it is used for testing purpose a lot.

* **You can combine Selenium framework with other frameworks, in your case, DB testing frameworks.**

**But I heard that selenium should not do this.**

* You can carefully design your Selenium testing framework with DB testing framework that they will work together independently. Using Selenium framework with other testing framework is not a taboo.

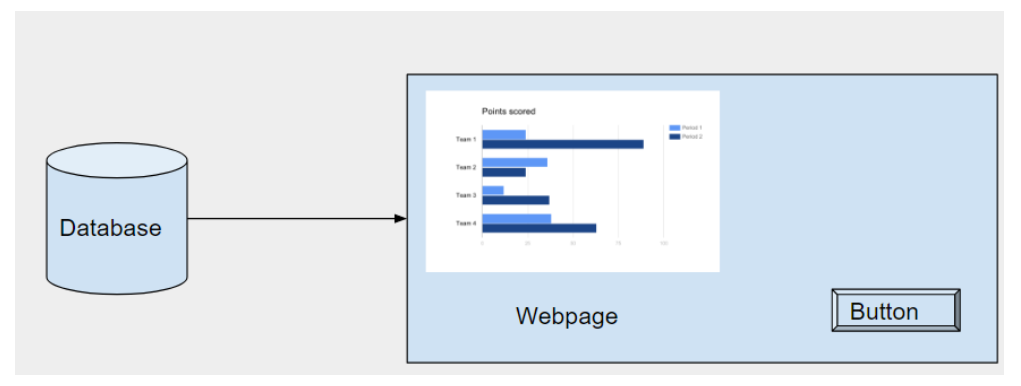
What I can suggest:

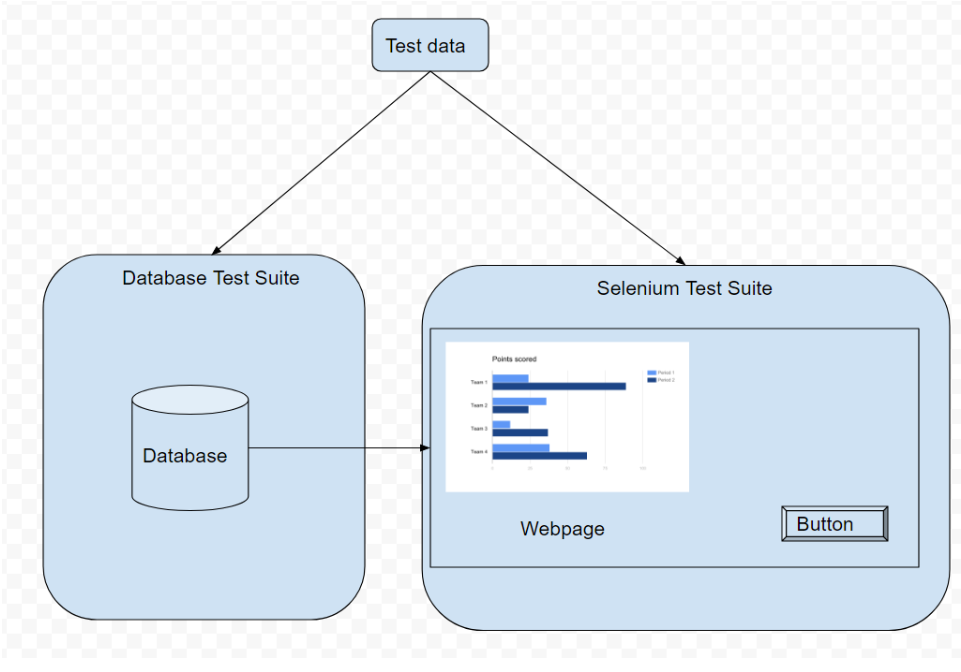
* Using data-driven test approach as you said.
* Build a front end testing suite using Selenium, execute tests, read back graphical information, convert them if necessary and compare results against expected values from your test data.
* With Selenium, you can also verify buttons are clickable, URLs are reachable and graphs are displayed.
* **Build a back end testing suite using a DB testing framework** of your choice, execute tests and compare results against expected values from your test data.

Say, you are testing a website that requires you to make a selection from a dropdown list and a graph is printed in return.

* You can use Selenium to navigate to this URL, click on a dropdown list, make a selection, **reads back the graph displayed**, converts it to a numerical value and **compare it against your expected value** from you test data.
* Or, you can make a **direct SQL query** from your DB test suites to the DB server, **compare the returned result against your expected value** from your test data.

When the button is clicked, this webpage will fetch a whole bunch of numerical data from its database, convert numerical data into a graph and display it on the website.

*  Selenium test suite will click the button, read the generated graph, convert the bars on this graph back to numerical values and compare those numerical values against expected results. Selenium does not know how the graph is drawn or where the data come from; hence front end testing.
* Database test suite will simply make direct SQL queries into the database; those SQL queries will fetch the same bunch of raw numerical data from the database; database test suite can then compare them against expected values from the test data file.
* **You may wonder what kind of SQL queries you need to pass onto the database server in order to fetch the same set of data, you may need to ask this webpage's developer in order to learn which SQL queries to use**.



# [How to deal with automated test cases that always fail?](https://sqa.stackexchange.com/questions/3874/how-to-deal-with-automated-test-cases-that-always-fail)

I'm facing a lot of automated GUI tests that fail one or more verifications during their run. All the failings points have been reported in our tracking system and they are known issues but get "ignored" as having low priority.

Development team promised to have those issues fixed "in the next few months", and this was many months ago.

Should I edit the tests to pass (for example, commenting out the verifications)? Or should I keep bashing on the dev team until they fix those minor display issues? What are the "best practices", if any, in this case?

One possibility here is to build into your testing the ability to flag a failure as a "Known issue" which is then reported with each run of your automation.

**Pytest use xfail, mark reason as know issue**

# [Tracking manual test cases which are automated](https://sqa.stackexchange.com/questions/2758/tracking-manual-test-cases-which-are-automated)

In such situation, when the tests are written somewhere in Excel or Google docs, it will be a big headache for automation.

Sure, to track the automated/not automated tests, you could assign some IDs for manual and automated tests and then track the tests by IDs.

And of course you can mark the tests as “automated” in the spreadsheet but after some time the headache comes…

1. The other testers will not rely on the automation and they will execute the “automated” tests manually.
2. The automation based on manual tests will have a lot of messy code, because the “manual” structure is not always good for automation.
3. The manual and automation tests will have their own live and after several months those two branches will be completely unsynchronized.

The way to solve this problem is making the manual tests **automation driven**. As an approach you could use the [Specification by Example](http://specificationbyexample.com/). (And here is [my collection on this topic](http://blog.zhariy.com/search/label/Specification%20by%20Example)). As the tools, you could look at [FitNesse](http://fitnesse.org/), [Cucumber](http://cukes.info/), [Concordion](http://www.concordion.org/Technique.html).

**Sure, you have a huge legacy of manual tests. But why not to start with this approach for new tests and new product functionality**?

**Do you add a news column in spread sheet with automated, not automated status.**

**That's the approach I use.**

**I typically have an overall document (either Word or Excel) which lists test cases, and indicates either the document to be used when they are manual test cases, or the script to be executed when they are automated**

With manual testing you can always improvise and adjust your tests on run time and look into unexpected conditions and handle them well.

While in automation testing the script will do only what they are programmed to do. They will not handle unexpected conditions or any change in the AUT (Application Under Test).

* **running a query and checking the output** against a saved known good file using one of the many comparison utilities (WinMerge is a nice free one).
* **running a performance monitor tool** to check for memory leaks, excessive memory use, and so forth.
* building a quick and dirty throwaway script to start and stop an application multiple times over to try to capture an intermittent shutdown error.
* building a throwaway script to generate a lot of data for use in a test.
* building setup scripts and teardown scripts to allow me to quickly generate a specific environment for testing something.

**When automation is needed**:

* **80/20 Rule** - if it's part of the 20% of the application that gets 80% of the use, it's likely to be a target for automated regression.
* **Release regression** - if it's something that must be checked for each release, it's likely to be a target for automated regression.
* **Critical path** - if it would make the application or major parts of it useless should it break, it's likely to be a target for automated regression.
* **ROI** - if it takes a long time to do manually and is error-prone, AND it needs to be done regularly, it's likely to be a target for automated regression. An example here: at my previous workplace, tax calculations were critical to the customers. It took a team of 3 testers 5 days to run a partial tax regression test. Automating the full tax regression took several months, but once it was completed it could run every night and would catch any changes that broke the tax calculations. Within a year that automation effort was into positive ROI in terms of the time it saved the testers. In terms of regressions that didn't reach the customers, the ROI was a lot higher.
* **Complexity** - if there are a lot of moving parts (i.e. interacting applications and services) it probably isn't a good target for automation. The individual pieces may be automated, but the end-to-end flow probably won't be.
* **External dependencies** - if it relies on a third-party application or interface, it probably isn't a good target for automation. Credit providers fall into this category: not all of them leave their test environments open after an interface to them is certified, and any bugs in the environment are out of your control.
* **Dependencies on hardware requiring physical manipulation** - The simplest example here is checking a printout (which *can* be automated, but doing so is painstaking and horribly error-prone). If your software causes some physical change to hardware, then it probably isn't a good target for automation.
* **"Hardness"/"Softness"** - if it's something that has exactly one correct result, it may be a good target for automation (such as the correct outcome of tax calculations). If it's "soft", such as whether an image displays well or whether a screen layout looks good, it's probably not a good automation target.
* **Coverage.** Automated tests are much faster and give much better test coverage. What's not covered is potentially wrong.
* **Reproducibility.** Because timing and inputs are almost the same in different test runs the results are more consistent than with manual tests. Errors which occur are usually reproducible; not so with manual tests.
* **Logging.** Automated tests have automated logging and evaluation. The quality of the software is immediately visible.

# I once joined an organization that ran automated overnight tests which emailed a completion report to a large group. After a few days I noticed that it always indicated an overall "Pass", yet my manual tests indicated major failures. When I dug in and looked at the automation, I found that the code was a mess and had no chance of actually detecting any real bugs. I removed most of the automation, and replaced it with a much smaller, shorter test that actually was capable of detecting some bugs.

* **Have different test sets, one small "smoke" test set containing only a few tests to run on each CI build, and a more thorough one running over night.**
* Avoid Sleep statements in your tests. Those are almost always a sign for bad test code, as they make your tests slow and/or unreliable. On fast machines, you're wasting time, and on slow machines the sleep-time might not be sufficient. Automated testing tools usually provide means to do some sort of synchonization, for example waiting for a dialog to appear or disappear

[Are Exhaustive “Brute Force” UI Tests Worth It?](https://sqa.stackexchange.com/questions/2648/are-exhaustive-brute-force-ui-tests-worth-it)

Lately I've been asked to write some automated GUI tests of the following style:

* Open the application
* Open ALL dialog boxes
* In each dialog check for ...

Are these types of tests worthwhile? What are some benefits of these tests? In some cases they can be very simple to write, but in others they require extensive knowledge of many unrelated parts of an application. As well, they can take a long time to finish, finding issues or not. Sometimes I feel these tests just aren't worth it, but I could be wrong. Any thoughts?

Resorting to rigorous UI testing can be a sign of untestable code, that isn't being tested at lower levels well enough. UI testing should be essentially just making sure the UI responds the way it should - and not functionality testing (with some exceptions, e.g., **end to end tests**). Unit tests, API tests, component tests, etc. should cover functionality. UI tests should be a small part of the overall testing effort in most cases.

I agree with Aniket, this kind of exhaustive UI test is not worth it. Another thing to consider:

Awhile back, I inherited a manual test case plan from someone who left the company. He had been doing things like checking the list sorting behavior in every area of the app. He wasn't a programmer and didn't understand that the code behind the dropdown box for the sorting mechanism was *the same each time*. He was over-testing. This is the kind of thing you'd run into if your test plan called for interacting with every widget possible.

Remember you're doing *functional* testing of the behavior of the application under test, not testing whichever UI framework the developers chose to use. Chances are you'll find any UI-specific bugs when you are executing your functional tests anyway.

1. Depending upon your UI, cost of implementing will probably be high. Don't forget to sum the cost of maintaining it when your UI changes in the next release and so on. This itself should warrant second thoughts.
2. UI tests can be flaky. In my experience accuracy of UI test cases is pretty low. I'd imagine this to get worse with brute forcing.
3. If you have 10 elements are there going to be 10! iterations?!
4. Time and resources required to execute these TCs and then investigate failures might outweigh the benefits.
5. If you do implement and execute something like this, will you be 100% sure that your UI is near perfect? :-) I guess not. There's your answer.

# [“how many servers our company should purchase for this web app” with load/performance testing?](https://sqa.stackexchange.com/questions/7809/how-to-estimate-how-many-servers-our-company-sould-purchase-for-this-web-app-w)

# host the app on as close to a production environment as possible to eliminate "noise" from your load/performance testing. In the case of a test server, you'd want to keep everything else off it, have any database hosted on a separate system if that's your likely production configuration, and then start exploring the application behavior under load.

# The key thing to remember here is that the purpose of load and performance testing isn't to determine what kind of equipment the application needs. It's to locate and eliminate as many bottlenecks as possible.

**Starting without any guidelines, how do I measure performance?**

This is kind of like regression testing: you start by deciding on a set of paths through the system. **If you are testing an existing site, you may be able to use the access logs to figure out which actions happen the most**. If you are testing a new site, you don't have any users, so you have to guess.

Here are some criteria you can use:

* **Choose paths that all or most users are likely to use every time they visit the site.** For example, a YouTube user is likely to search for videos and view videos every time they use the site. They are unlikely to change their preferences every time.
* **Choose paths that you expect to be server-intensive.** If you know an action requires the server to do a lot of work, include. If you know an action requires locking a resource, include that too.
* **Choose paths that need to be fast.** We would like for every action to happen instantaneously, but some actions are more important to be fast than others.
* Third, and most import, **when doing performance testing**, you need to isolate the thing being tested as much as possible. Ideally, **you shouldn't be crossing the internet to get to the web app, or parts of the web app**. You should have a **dedicated host, dedicated client machines, and a LAN connection that crosses as few boundaries as possible between the two**. If you have to test a remote site, you should make sure that the client machines are at that site as well. If you're using AWS, set up client AWS machines. The more variables you have to deal with when doing performance testing, the harder it is to make a valid conclusion when testing.

**Front end testing is important and is often part of User Acceptance testing**.

The most popular approaches uses Selenium along with a programming language - Java, Ruby, Python or C#

The basic principle is that you create an instance of a Selenium object called 'WebDriver' that represents the instance of use of a specific browser, for example chromedriver for chrome.

You then write tests using this infrastructure. You refer to web page elements, you 'click' on links and buttons, you type text in input fields, you submit forms, etc. Basically if a user can do it, you can usually replicate that behavior using selenium commands through the selected language and framework. In some cases is done through a test framework tool, for example for Ruby on Rails (Application Framework), using RSpec (Test Framework) and Capybara (Selenium API)

# Yes I consider selenium end to end testing because everything –

# the web server,

# the database,

# the application server,

# the browser and

# the representation of the user clicking and typing –

# are all involved and being tested.

# I currently and previously work with CI environments. Previously with Jenkins (yuch!) and currently with CircleCI (yeah!, my recommendation for a CI tool). These tests do run as part of our CI build for every pushed commit. When we need to test against other browsers or devices we add on a provider like SauceLabs or Browserstack to provide the infrastructure for that.

**A GOOD test suite ultimately comes down to a few things**:

* how hard/easy is it to **add new tests**? Can you add new tests to your regression suites every time a bug slips out to the customers without needing to spend ridiculous amounts of time coding? (if you can do it with a few lines of data in a test data file, more power to you)
* how hard/easy is it to **update the tests** when **the application GUI changes**? If you're effectively rebuilding your tests every time someone changes the tab order, you probably won't have a high trust test suite.
* how hard/easy is it to **update the tests** when **the process flow through the application** changes? If it can be done quickly and easily, chances are your test suite is going to be more trusted than if it's a major rewrite.
* what data is your test suite communicating and to whom? Some things to consider outside the number of tests run/passed/failed are known failures (where something is reported but hasn't been fixed yet), coverage statistics (which may or may not be useful to your application, depending), **an overall summary of tests run/passed/failed over time** (an ever-growing number of tests which mostly pass tends to build confidence).
* What data are *you* communicating and to whom? No matter how good your test automation is, people are going to look at you (and/or your team, depending on the structure of your company) first, and impute trust based on how much they trust the information you give them.

# [what is the best way to write automated end to end user scenarios?](https://sqa.stackexchange.com/questions/3229/what-is-the-best-way-to-write-automated-end-to-end-user-scenarios)

When automating user scenarios, we typically cover an **end-to-end user flow**. Unit testing recommends that each test method should be independent and only test one thing. But things are different in system acceptance testing, as we have to cover a user flow through the application. Consider an ecommerce application where the user visits typically follows the below steps:

1. visit homepage
2. search for a product
3. select the product
4. add the product to cart
5. click checkout
6. sign in
7. enter payment details
8. submit order
9. wait for confirmation

Now **each of these steps need to be verified as we flow through the user journey**. So, there is dependency in test methods, e.g. step 3 depends on step 2 which breaks unit testing practice. Also if we put all steps in one test method it is not easy to detect what went wrong if the test failed. And if we wanted to isolate each test method so they are independent of each other, we have to repeat the initial steps over and over again.

What I have done in the past is to **use TestNG to provide this dependency on previous test methods**. But I wanted to know how do you handle user journeys in your testing?

# [Is it a good idea to iterate through several browsers in one test using Selenium WebDriver?](https://stackoverflow.com/questions/30062264/is-it-a-good-idea-to-iterate-through-several-browsers-in-one-test-using-selenium)

I am trying to run a simple test on multiple browsers, here is a mock up of the code I've got:

String url = "http://www.anyURL.com";

WebDriver[] **drivers** = { new **FireFoxDriver**(), new **InternetExplorerDriver**,

new **ChromDriver**() };

@Test

public void testTitle() {

for (int i = 0; i < drivers.length; i++) {

// navigate to the desired url

drivers[i].get(url);

// assert that the page title starts with foo

assertTrue(drivers[i].getTitle().startsWith("foo"));

// close current browser session

drivers[i].quit();

}// end for

Most of the time it is pointless to *immediately* run your test against multiple browsers. Most of the problems you run into as you are developing new code or changing old code is **not due to browser incompatibilities**. Sure, these happens, but most of the time a test will fail because, well, your logic is wrong, and it will not just fail on one browser but on all of them. What do you gain from getting told X times rather than just once that your code is buggy? You've just wasted your time. **I typically get the code working on Chrome and *then* run it against the other browsers**.

(By the way, I run my tests against about 10 **different combinations of OS, browser and browser version**. 3 combinations is definitely not good enough for good coverage**. IE 11 does not behave the same as IE 10, for instance. I know from experience.)**

Moreover, the interleaving of tests from multiple browsers just seems generally confusing to me. I **like one test report to cover only one configuration (OS, browser, browser version)** so that I know if there are any problems exactly which configuration is problematic without having to untangle what failed on which browser.

**Driver.switch\_to.window()**

* **switching between tabs, not different windows**
* driver.switch\_to.window(driver.window\_handles[1])
* **driver.window\_handles is a list**
* **window\_handles[0], first tab**
* **window\_handles[-1], last tab**
* when you click and generate a new tab, driver stays on the same page and does not switch to new tab automatically.
* driver.current\_window\_handle
* **driver.close() , close window pointed by driver.current\_window\_handle; if you close it, and try to call driver.close() again, will encounter**

selenium.common.exceptions.NoSuchWindowException: Message: no such window: target window already closed

print(driver.current\_window\_handle)

* will also encounter selenium.common.exceptions.NoSuchWindowException

[Best approach to Automation testing for this scenario?](https://sqa.stackexchange.com/questions/29066/best-approach-to-automation-testing-for-this-scenario)

I'm automating a web application and I have 2 things there :

1. An **administrator side**, typically used for **creating users**, **managing users**, modifying the content etc (the list goes on)
2. A **user login side** to view such content etc ( **need to be tested**)

How do I handle the automated testing for this?

Key here is how to **prepare data before running test**.

Both ways should be good.

do I start by

* **setting up the test via the administrator UI** to create a user,
* Then need an admin user account too.
* Create an admin account through UI (something can be hardcoded by dev for conveniences); create a user
* Verify user login and infor
* then navigate to the user side, login and verify what I need?

**OR**

* setup the test by **inserting a user into the database**
* using them for any user-side tests? Inserting via the DB for any administrators when testing that side?

I'm confused on where the flow of a test should truly begin,

* Create admin account (which needs certain credentials too)
* Login as admin
* Create user, setup appropriate privs all via UI
* Login as said user, execute the test
* Verify the results

#

**It depends**

**If your application has an API, handle all the permissions and business logic through API tests. They're faster**.

My approach, assuming that I had the ability to restore a test database to use as my oracle would be:

* **You could start your tests by restoring your pre-configured database then running your tests Restore database as part of the test run setup.** The database should have all user accounts needed for the main tests, as well as some content to use for edit tests so that every test is independent.
* **Group back-end and front-end tests.** Run the administrator tests as one test set, and the front end tests as a second set. It's up to you whether you use the same database or not.
* **If possible, anything that permanently changes data should be an API test**. Creating your user, validating content, editing users... the outcome of these actions should be API tests rather than UI tests. If possible, **your tests from the UI end** should focus on things like **UI validation, required fields, and so forth**.
* **Extract navigation into utilities functions.** Your tests will be cluttered if you try to include login and navigation to your target page as part of the test, particularly since you're going to be doing a lot of similar actions there. Use helper routines and/or **POM** methods.
* **Automated full flow tests should be minimal.** At most one or two tests to cover the end-to-end functionality - the actions of loading multiple pages, entering data, saving, logging in, and logging out, will add a lot of time to your tests when you try to bring them all together.

This assumes that you're testing a well-designed application **where the UI uses API calls to communicate with the back end**. If your application is less modular and has **the business logic tangled with the UI**, you may have no choice about running a lot of end-to-end UI tests.

that is some great info thanks Kate, sadly the **business logic is heavily tangled with the UI** so i'm going to distribute widely across chrome instances on GRID and have quite a few tests, ive been restoring a sql server .db at the start of each run and using that starting state to keep the tests independent.

##

I have a "customer side" and an "administrator side" of a web application. I want to make my tests as solid as possible, so should I be **adding new customer(s)** on my administrator side before each test and **removing them upon completion**? I'm concerned that if something fails on the removal, the next run will potentially kick up an error of someone existing when trying to create them again.

# [Data creation and deletion for happy path automation](https://sqa.stackexchange.com/questions/44743/data-creation-and-deletion-for-happy-path-automation)

Seems like lots of ppl like to use API calls to create test data or prepare certain data. This will mix UI test with API tests. NOT a good idea (jan)

**if an automation engineer had to write a test for commenting on a post in a group on Facebook and did not have direct access to the db, but instead to the rest api, the automated test would need**:

1. A user
2. A group
3. A post in the group

Since a test **should not depend on data from previous tests**, should this test create a group, then create a post and then create a comment on the post? Should the same test then delete the comment, delete the post, and delete the group? There are lots of points for potential failures here. What are some best practices for getting an application into the state necessary to perform an atomic automated test? Create and delete the data via the API as part of the test setup and

unless you are validating delete feature, it doesn't make sense to delete using UI as it will time consuming. Can be done using UI session Fixture. You should delete or clean the setup in one go using DB or API, API is more preferred as it ensures more access controls and avoids unintended changes

**You are right. Test should not depend on data from previous test**. But still, you can have test data **created for tests** and **that data can be used repeatedly**.

For example, in this case, **we will create dependent data with session fixture. It's a one time activity**. So, we have an existing user and an existing group and an existing post.

As part of the test:

* User will log in
* Add a comment to the post in the group
* Validate the user was able to comment successfully

As part of **the teardown**, delete the comment (from UI or api, whatever works). Your test can execute any number of times, with the pre-requisite data you created once.

**Why we take this approach**?

* **We don't want to have too many pre-requisites**.
* It will increase the test execution time significantly.
* Moreover, **if the API endpoint to create user/group is down**, your test won't be executed.

We have many scenarios where we do things as part of pre-requisites, for example, creating users etc..but we try to keep it to a minimum.

We have **tried different things**; as you said, creating all the dependent data before the test execution and then cleaning everything after the test execution completes. **But**, we found the above approach as the simplest with practically no overheads.

Not saying this is a bad approach, but how do you handle leftovers from previous tests ? let's say the comment deletion was successful but corrupted the DB for the next test, or even worst corrupted the DB in such a way that will be felt only in few tests ahead ?

**Then**: if there is issues with the deleting part, it will be fixed first.

* Well. We take this approach where the risk for such mess up (like corrupting a DB etc. is very unlikely). If the tear down action has such a huge impact that it can fail 10 subsequent tests, then the approach needs to be revisited. But from my experience, the likelihood of a disaster, by an action like deleting a comment as part of the teardown is very low.

# [How to manage test data in database](https://sqa.stackexchange.com/questions/13762/how-to-manage-test-data-in-database)

# Insert data with sql scripts before the automation.

# Those scripts should clean up database first, then set up preliminary data.

# Create data on the fly

# Creating fixtures for data creation and cleanup. Fixture functions can be API calls or UI.

# Run sql queries commands for data creation/deletion.

**Background:**

* Stack is **Java/Javascript/Mongo**

I want these tests to be fast and independent. So I will be testing one item at a time. **The question comes in when there's something I need existing data for**. I cannot **restore** databases for these tests, so a **DB snapshot is out of the question**.

**Example scripts:**

* create user
* create new entry from a newly created user

**The Questions:**

Should I try to get developers to setup an API method for testing, or create a SQL query to just create the data. My gut reaction is that I don't want to run SQL queries like that to insert and delete data. But I am not sure I can justify asking for those API methods to be created.

# The first thing you absolutely need is a test environment under your control. Even if backup/restore operations aren't possible in this environment, you must have a non-production environment where you can work.

# I'm in a similar situation with a massive enterprise-level classic ASP web application, and an API is out of the question. The code is too tightly interwoven with the GUI.

* If the database uses stored procedures, you can call these to set up the data you need and **later remove** that data to maintain a more-or-less clean data set. Create a verified procedure and then use **the procedure as a session/class fixture** or a database procedure to facilitate creating test data/env **on the fly** as testing:
* you create a user through the normal interface, and
* verify that all the user's data lands in the database in the correct table(s) and format(s),
* then in your next test where you perform an action with a newly-created user **you *use a stored procedure to create the user***, then run the action in the normal interface.
* You can periodically reset the database so that you don't overload it with your test data (where backup/restore is not an option, this is the second-best option)
* After you create your test environment, you can strip it down to just the configurations/customers/options that you are testing with. In situations like mine, where you have thousands of customers all using the same application database, you only need one customer per distinct configuration set, and you may be able to use pairwise analysis to trim things down further, particularly if some of the options aren't mutually exclusive or there are combinations none of your customers use.
* If it's possible to get the programmers to build an API, then do so: it will simplify your automation requirements immensely. Being able to test business logic via API and reserve GUI automation and/or manual testing for end-to-end scenarios where it's not feasible to work with API calls is a huge time-saving both in test development time and in test run time.

"reset" varies. It can be restoring a snapshot, or it can be a script that removes anything other than the settings/data you want. Where I work it's running an import process that sets everything for a single company within the database to the canonical values

The most important thing to remember is **Never, *Ever*, run untested code on a live database** - you could *potentially* do an immense amount of damage or possibly make a subtle change that is hard to isolate and takes years to track down.

I would say that you need to test with the API for robustness and functionality **and** with **both** with a small, *test*, database that you can easily check for changes, introduce "problem" records, etc., and **with an offline snapshot of the real database for load tests** and to ensure that the real data doesn't contain problems that you haven't thought of.

In case you need to use SQL, goes without saying, you should externalize them from your test scripts so that any change in table/sql doesn't impact your test script.

# [Prepare data for groups of test cases](https://sqa.stackexchange.com/questions/42686/prepare-data-for-groups-of-test-cases)

I am going to have a family of test cases that are going to work on a single data set. This data set needs to be loaded into a database as an initial step which is shared between all the test cases.

* **Will the tests modify the data?** - If they will, you'll need to reload the data as a precondition for each test.
* **How long does it take to load the data?** - If it takes a long time to load the data (and what constitutes a "long time" depends on your situation) you will likely want to reuse the data
* **Does the data impact other tests?** - If you have other tests to run that will be impacted by this data, you're going to need clean-up to remove it once you're done, and you won't be able to add it until the first of your tests that use it.
* **Can the tests with this data be isolated?** - If you can isolate the tests using the data, it may be worth your time to isolate them and run them separately to reduce the likelihood of negative impacts to your other tests.
* **Are the tests manual or automated?** - **For automated tests, a method that loads your data and gets called at the start of the test and a teardown method that unloads it could be your best option.** For manual tests, things may be more time-consuming.
* **Creating a single baseline** data for our test environment and manage it using tools like Flyway (Java) or Knex.js.
* This baseline doesn't change too often and will be used for all of our tests.
* Each test will create the additional entities it requires for its operation and will possibly delete them after its successful completion

# [Should release management accept anything less than 100% pass on automated tests?](https://sqa.stackexchange.com/questions/14104/should-release-management-accept-anything-less-than-100-pass-on-automated-tests)

1. Is this known issue or new issue?
2. Is this test-environment related? Such as test script or test data or setup related?

If this is the problem, then definitely you need to improve your automated test cases, so that they execute perfectly (not in terms of approach) but in terms of requirement and coverage.

1. Is this application related issue, which is throwing some errors because of which result is not 100%'.

In this situation normally we refer to the **test plan** in the contract. **Test plan** or **SLA** (shared with management and client) should provide you the value that under which release is acceptable. like in our case one of the client which I worked has clearly said that anything below **100%** is not acceptable (whether it is related to functionality, UI or Language requirement). Hence for us until all test cases are passed we never give a **'GO'** for the release.

So, something should also be there is your case formal or informal communication, if your test plan says than with **'Low'** priority bugs and above **95%** of **Quality index** (i.e. passed test cases) your application is ready for deployment then, you are good to go and there you need to highlight this thing in your Test Report and Release notes (if any) and communicate the same to the release management team too. At this point you will need to educate them that your pass criteria matches the agreed upon condition and so it is a Green light. Create your report in same way where on first look the impression goes that you are meeting the agreed **SLA** and hence, further explanation may not be required.

And, if your test plan or SLA says it should be **100**% then you can't go ahead. This time you will hold on the release and inform the client about the condition that you have 97% pass and 3% fail (with details), then let him decide, because sometimes for **meeting the deadline you can compromise over 1 small feature** but rest of the X-1 features should work perfectly fine, which will change your **97% to 100%**, **but keep this decision with Client or Management as you should not delay the release from your end**.

You should spend more time understanding your application and importance of different features.

**It does not make sense to postpone important release**, which delivers important new features which give you advantage over competition because of a bug in some obscure feature which rarely anyone uses. But you **would not want to knowingly release software with bug** which can compromise safety of people or their lives.

When our tests do not run perfectly, **we follow the same triage process**: investigate the bug, how big a problem it is for how many users, **find any workarounds if possible**, and assess **the danger of new bugs** created by fixing this one, then put the bug in the pipeline.

Of course regression bugs have usually higher priority, because they should be easier to fix, and they deliver functionality users expect.

# [Documenting application test coverage](https://sqa.stackexchange.com/questions/28209/documenting-application-test-coverage)

# we can show

# tests results in a jenkins junit report form, we can show our

# manually supported check list of things covered by the automated tests.

# and we can show that we've gone through our manual test scenarios.

# As far as bug tracker, we can show currently opened issues, opened/closed bugs per release, things that are currently in progress or in testing. This is basically what I use trying to answer the question. just trying to see if I can improve on that and what others have in their projects.

1. **test plan**, ie,

* **a list of scenarios** you'll attempt,
* or **function test cases** you'll run, or whatever. Ideally, these will have ties to new functions in the code, or are marked as being done for regression purposes. The test plan should have been reviewed by the stakeholders, including development, support, and possibly the client/end user.

1. A problem list: jira tracking

* with problems marked as open, closed/fixed, closed/unreproducible, with some sort of customer associated severity associated with each problem.

1. Given the above, at any given point during the test cycle, you should be able to have some idea of what percentage of your testing has been attempted (test executed but unsuccessful), completed (executed successfully), and bypassed for some reason (can't be executed for some reason).

**We have both automated and manual tests** for our actively developed application. A regular Jira workflow in our case is something like this:

**Open -> In Development -> In Testing -> In Automation -> Resolved**

From release to release, we are usually **behind in automating the new features and fixes**. But, we do, of course, **cover the new features with manual tests and test scenarios**.

**How do you keep track of what functionality/features/screens/pages/fixes of the application are covered by an automated test and what are not?**

The ideal situation would be to be able to see, at any given point in time, what parts of the application are covered with manual tests scenarios and what parts are covered with automated tests.

**Should use master spreadsheet to cover test scenario, detailed with test plan/documents.**

* **In master sheet, add one more column for automation done or not**
* **Test status pass/fail or ongoing/not started**
* **Last tested date and version of app. If part of regression, status should be just up\_to\_date.**
* **Jira number to track**

**Test data related files should be part of test code base**

**Jira to track bugs and issues.**

# [Release methodology for complex distributed system](https://sqa.stackexchange.com/questions/7696/release-methodology-for-complex-distributed-system)

I have a problem on deciding how to guarantee good testing of a complex distributed system before release. We have an automated testing infrastructure in place. The problem is that running all the tests **takes around 5 days** (including stress, longhaul etc.).

Let us assume that we start with build V1 of the system and we start testing on that. **After 5 days, we look at the results and notice that there are some failures, either due to dev code changes, some due to breakage in the automation**. In the meantime, devs continue work on their tasks and after 1 day, we fix all these issues, we get a **new build out** (V1.1 - which contains more dev changes + changes for the current testing) and we want to start testing again.

First of all, if running full-scale test takes 5 days, then milestone release cycle should be more than 5 days to accommodate the completion of testing. In other sense, there should be a shorter version of test suites running to check minor-release in a shorter time frame.

1. **Jenkins build server runs smoke/end\_to\_end test for quick sanity check; less than 10-15 mins.**

* **End\_to\_end basic functionalities.**
* **New feature or bug fix tests pass smoke test, meaning passing positive test cases if not enough time for QA to include all test cases.**

1. **QA manually or creating scripts to thoroughly test the feature or bug fix; if works, integrating to nightly regression tests. Reporting any issues found during standalone testing.**
2. **If any failures happens, report bugs through JIRA to get it fixed asap.**
3. **Nightly regression has full-scale testing and making sure new code does not break legacy features**.

* Regression test should focus on functionalities
* Leave load and stress tests to staging test server and only run on major-release.

I'd **start by separating the functional regression tests from the stress and load tests**. These are the tests you **run daily**. If it takes more than a day to complete, consider breaking the functional regression tests into multiple smaller suites that can be run in parallel on **numerous machines**. You want these scheduled to kick off at night so they (ideally) be completed by the time you arrive at work in the morning so you can analyze any failures and report regressions quickly. This gives you a faster turnaround for purely functional tests.

* **For your long-run tests such as the stress tests and load tests**, I'd look at generating short versions that can be run in parallel and scheduled the same way as the functional tests. These won't give you the same level of value as your long-run tests, but they should be sufficient to detect any serious problems. Your long-run tests can then be scheduled to run say once a week and will pick up any other problems. You won't have quite the same level of granularity this way.

If **you *can't* run tests in parallel** (you don't have the equipment, the system isn't going to allow it, you don't have the licenses for your automation environment... It's not ideal, but it happens.) I'd suggest you work this way:

* **If your tests aren't already reporting as they complete, set them up to do this**. If you're running them as a single long suite, **split it into multiple suites set to run in order**. This way you can be analyzing failures before the run completes.
* **Order your tests to put smoke tests and broad functional regression first**. This will give you the earliest possible notification of functional regression issues.

Some things you can do to gather more data faster regardless of whether you can split your automation into parallel runs or not:

* **Each test should report as soon as it completes**. This doesn't have to be a detailed report - a simple X items passed, Y **items failed email** is enough to alert your team to potential problems. **Should have a log that only reports errors / failures or excetptions**
* **After each test, export all your logs** to a **shared network location** where you can analyze them while the automation is running the next test. That way, if there's a failure you can start analysis without impacting the current run. (This assumes that each of your tests is a granular sequence that starts with the AUT closed and closes the AUT at the end of the test.)
* **If your tests use a common setup sequence**, consider turning this into a **database restore/data flush** and reset operation that pulls the data set from a shared network location. I've been in the situation where every test run spent an hour or more configuring data for the actual testing. **The team built a once-a-week data configuration automation run and modified the other runs to simply pull the data from the configuration run, reducing setup for the functional runs to about five minutes**.
* **Consider going through your tests and replacing static waits with component-dependent waits wherever possible**. It's a lot easier to put a static wait for some number of seconds than it is to code a wait routine that will check for a required component, and return an error if it fails to instantiate and become active within a specified time, but a whole lot slower: **if your static wait is 3 seconds and the component exists and is active in 10 milliseconds, the automation will still wait 3 seconds**. A fused wait that checks for the component every 100 milliseconds will only wait 100 milliseconds. If your automation has been around for a while, chances are good that there's a lot of static waits scattered through the oldest code (I've never seen an automation code base that didn't evolve and improve over time).

No matter what strategy you choose, you're going to be compromising between completeness and quick turnaround: your goal is really to get the best possible mix of both.

# [Can we programmatically update locators on UI Change in UI test automation?](https://sqa.stackexchange.com/questions/22596/can-we-programmatically-update-locators-on-ui-change-in-ui-test-automation)

I have been working in test automation for the last 10 years. The problem that keeps coming up and surprisingly is still unresolved is:

**Can we programmatically update locators on UI Change in UI test automation?**

What I mean is, automation teams need to run automation suites periodically, and **many test scripts fail because of locator changes, not because of *Application changes***.

This way we can remove unnecessary overhead of running a large automation suite and analyzing it painfully only to find that all we found are locator changes, not any application issues. And finally ***manually updating all the failed locators with new ones using tools like Developer tools/firebug to find out new ones and manually changing them in the code which is generally the case with most of automation teams***.

My answer to this is **to write robust locators** that are not affected by other UI changes. They should be unique but not overly specific.

So instead of  
div div div span table tr tr td td input#last\_name

use  
input#last\_name  
or  
form.new\_user input#last\_name

which will be more robust and *allow the overall structure of the page to change*

**dev and ui** have the same need for unique identifiers for both application code and also css styling so you'll want to work with them on a common approach when possible.

**update the tests - ideally the page object definitions rather than the test code**.

You also need to be sure to have pages start in the same condition and this means having a **ui test database which is reset for each test**. **Your test database should be different, for example, from staging. It should be empty for each test and each test should setup all the data it needs including reference data.** This can be quite a challenge for some organizations particularly older and larger organizations.

Michael, even after providing robust locators, lot of times they still change as overall structure of page is changed or even the unique locator which we defined itself is changed like name

# [Adapting agile testing is it good approach?](https://sqa.stackexchange.com/questions/1362/adapting-agile-testing-is-it-good-approach)

#1. If you follow weekly release model and every week you do manual testing. This is NOT AGILE testing

#2. Objective of adapting to agile is **automate along with development so that on continuous releases you have enough automation to support qa efforts**

**#3 Jenkins tests upon checking in code on feature branch**.

QA add test cases for new feature/ bug fixes if time allows; otherwise still need to create test plan, manually test and create automation script. If testing has not happened or issues found during testing, feature branch new code does not need to merge to master trunk for night regression test; testing should be done tomorrow; once passed, feature branch code can be merged to trunk tomorrow and run my next day regression test.

# 3.1 agile dev/test: **continuous dev and check in while continuous testing**

Dev doesn’t need to wait to check in until whole feature is developed; once some partial features are developed and pass unit test, it should be checked in and create build and QA can start verifying.

#4. **Continuous updating automation suite** with production bugs / feature changes. I have often seen cases where automation takes second priority when it demands time

#5. Do you follow provided guideline <http://blog.scrumpad.com/2008/11/7-practices-to-agile-qa.html>. They are very good. I have seen developers providing builds without documentation and verbal explanation to feature implementations ?

#6. Context of agile is misused for not providing enough details on implementation / no documentation / push buggy code to production and keep fixing it in iterative builds

Let's say **our sprint ended today** and we automated the regression part,

* but tomorrow we got the new version of the software. Our test cases may not work in the new version if there were UI or any other changes. That means we have to update our testcases **in coming sprint and wait till our next sprint ends** and product owner reviews it and accepts it before we can run the test. It would already be late for the result.

Our management team really wanted 'testing' to be 'done' during the same section of time as dev in the same sprint, rather than next one.

1. Since Jenkins runs test every time a new check in happens; the feature branch gets tested. Only if no issues with positive test, it will be merged in trunk. QA should check in full test cases in regression tests.
2. If you are getting build on alternative days then, once the sprint is started you can start preparing your test cases corresponding to the assigned user stories.
3. Once the build is deployed then you can start executing your test cases, if there is some change in the build then you can execute your test suites.
4. Whenever a new build is out, you should execute your sanity set of the test suite.

# [What changes for testers when they are testing in agile environments?](https://sqa.stackexchange.com/questions/38391/what-changes-for-testers-when-they-are-testing-in-agile-environments)

## **Classic testing is Quality Assurance**

## **Agile Testing is Quality Engineering**

### **Quality Assurance**

**Traditionally** most testing as done at the end of the development. It asks if the product that has been developed meets the requirements, both explicit and implicit.

### **Quality Engineering**

This field is primarily concerned about automation and automated tests. It breaks out testing into: Unit testing Integrated Testing **Automated UI Exploratory**, **Performance & Security**

Over time this leads to the following situations:

**Classic Testing** - the product is built, the developer considers it is working but Quality Assurance has to convince them and others when there are issues. Issues are frequently found late in the process with little time for quality fixes. Fixes are expensive and often affect production. This is common in Command and Control and Waterfall environments.

**Agile Testing** - the goal is to shift testing left and **test as early as possible**. **Ideally many tests are written first and fail without the application code to support them**. **unit tests** written by developers to ensure components work. This makes them executable specifications and when the application code is written that makes them pass, then you are good to go.

As a software testing company, tester has a very crucial role while working in Agile model.

* **When the Scrum master creates sprints**, with user stories.
* **tester reviews** the **user stories** and provides **story points** to each story.
* **Task of developer** is to write the code and perform unit test of his code.
* But Tester needs to ensure that all the story requirements are met. Tester needs to coordinate with Developer, Product Owner, **Scrum master** and ensure quality release after Regression cycle is run.

##

**Good Agile teams test constantly - not just at the end**. If you include test design as part of feature design (and consider how the feature will be tested as it's designed), you won't have to write GUI automation - you can automate at the controller level, or through another abstraction.

* I write automation code with Robot Framework in an agile development environment. So long as you have an idea of the implementation of the code in test, you can **write automation scripts before the code is deployed**. This should be **an exercise in logic and workflow**.
* Once the code is live you can update your script with specific identifiers for each keyword. This approach pre-supposes that you have a library of existing keywords for testing your application.
* Often times **we finish automation after a sprint is over** - **we run all automated scripts nightly so the script still gets run plenty**. Furthermore, even without a nightly automation run, your automation scripts will be viable during **regression sprints**.

2) In most Agile teams (certainly any I've worked with), **user stories are geared toward completing user goals**. This means that very often a piece of an interface is completed in accomplishing a user story and it doesn't change very much after that. Once the tests are written for it, they don't change much either.

**Essential Tests**

* **Acceptance Tests** - By definition, the **user story can't be accepted** if it doesn't pass acceptance testing. The exact form the acceptance tests take will depend on the user story: for back end focused user stories, acceptance tests could well be performed by the tester or by another developer, where front-focused user stories are more likely to see acceptance tests performed by the product owner or customer.
* **Developer and Unit Tests** - Whether automated or not, these tests will happen before the user story is made available to the test specialists.
* **Steel thread Functional Tests** - Functional testing of steel thread for the user story will happen. It may or may not be automated depending on the nature of the user story, the maturity of the team, and the time available.

**Other Tests**

* **Other Functional Tests** - If there is time available (and realistically, even with an agile methodology there is likely to be pressure to complete features within a time frame the team does not find comfortable - I've yet to work in a situation where there is no time pressure), functional testing outside the steel thread will happen. This covers a lot of ground, particularly since even at the user story level if there isn't a really *good* software architect on the team there's likely to be important functionality missed in user stories (especially when user stories are customer-driven and the application serves multiple customers).
* **Regression Tests** - In my experience regression may or may not happen during a sprint, but it will definitely happen prior to release. If tester automation is being written, it will usually fall into the regression test bucket. My experience with automated regression is that *if time is not allocated to develop, maintain and analyze automated regression, it will not be done*.
* **Security/Penetration Tests** - I'll perform functional tests with an eye towards security and penetration issues, but this is a very specialized field, and many testers (including me) lack the skills to properly test an application for security issues. As a result, security and penetration testing is often an afterthought.
* **Performance Tests** - The issues that apply to security testing also apply to performance testing. Teams with good architectural designers will build for **good performance and scalability**, but the **expertise needed to perform good load and performance tests** means that these are often afterthoughts and may not happen.
* **Usability Tests** - Theoretically, usability should be designed in from the start, but without a usability specialists, that doesn't always happen. I've often seen usability issues reported by customers after release.
* **Compliance Tests** - Whether ensuring the software meets the appropriate regulatory standards is a must-test, a should-test, or a will-test-if-there's-time depends on the software: in some industries, such as health care in the USA, it's a must-test. Casual games, not so much because that segment isn't nearly as heavily regulated.
* your QA team members ***need to see your features***.  This means **getting access early**, and often, to the **stories** being developed so they can provide the ongoing feedback that ensures **high quality is delivered within the sprint time box**.

1. **Daily Build Environment.**I’ve heard this called all kinds of things, from ‘Development Lab’, to ‘Staging’, to ‘Dev Test’.  Essentially, this is a place where QA can test things as they are being worked on.
2. **Automated Deployments.**  Your build server should be running regular (daily) deployments to your Daily Build environment.  If you can, give your devs and QA folks the access to push on demand as well.  This works particularly well when you need to get some rapid feedback for a bug fix.
3. **Regression Environment.**Just because QA has seen things throughout the sprint, that doesn’t mean that something they saw on Day 3 is still working on Day 9.  Ensure you have a stabilization period of your sprint and during this time release to a separate Regression environment where QA can run through everything prior to your demo

As shown in the other answers and comments this is a common issue that I've seen in several companies that I've worked in. Thinking it through, I suspect **most companies struggle with the generic issue of allowing enough time for QA, testing, and automation once the feature is complete**.

1) **Testing happens *before, during and after* dev work**. For example, if you practice BDD and write a failing test *before* the app code then you will be one step closer to your goal of keeping up.

Testing of a **particular feature** that is being **created in the sprint** can be done, only if the developer has developed the feature up to some extent. Meanwhile, when the developer is busy developing the feature, a QA should start working on the test plan/test cases on the basis of the feature specification document or the user stories. If the QA team is automating the test cases and using BDD tools like Cucumber, then he must start writing the Cucumber for the test cases to save time. A QA should be in continuous touch with the developers so that he receives at least a piece of the feature which has been developed.

Once the developed module is received, now a QA has an ample amount of work. He should first do a sanity check of the module received, and quickly log the issues identified in a bug tracking tool. Also, communicate the developer regarding the issue. Side by side he should also automate the test case. This cycle needs to be processed quickly so that each module is tested and delivered without any bug on or before the sprint end date. Thus, in other words, the work of a QA starts as soon as feature specs or user story is received in the sprint and the actual testing can be started as soon as the developer develops some module of the feature.

My idea is pretty simple. Prepare regression automation suite and setup in CI & CD pipeline and add this as a post build action.

Your focus during the Sprint should be starting automation of repeatative tasks and push this in CI CD pipeline daily.

If automation taking time for some test cases it is better to done Manual as first round and priorities as per need.

**Scrum Sprint cycle:**

**Meeting 1:** scrum master **assign/review feature or user stories to be implemented and tested.**

**Meeting 2: let’s say 4 days or 1 week later,**

* **will evaluate and make sure feature passed all tests and ready to deploy.**
* **Started a new round of new feature /bug fixes**
* **For a big feature , it can be divided up into smaller features and each sprint only needs to focus on some of the sub-features.**

**give developers story points in the sprint that will get done in LESS THAN sprint cycle days number**. To put this in perspective, in my team’s **two-weeks sprint**, **13 points is the max** and stretch of load to each developer. In my team a 13 point story in complexity means that developers think it takes 7 days to get done. **That’s right 7 days! Not the full two weeks**.

**All developers deploy code that requires QA testing near the end of agile scrum sprint.   
Now QA team members, which seem to have always been outnumbered, brace themselves for the storm, and they start testing this many features in the span of a day or two. QA team members step to the plate and they make it happen! Marvelous 👏🏻👏🏻  
Now developers get feedback on what they missed (a topic for a new article) and now they have to fix it in a day or two before sprint review day, which increases intensity and stress for EVERYONE!**

* A **user story**, by definition, is a description of a **software feature** from **an end-user perspective.**

# [How can I test a user story? Examples please?](https://sqa.stackexchange.com/questions/6450/how-can-i-test-a-user-story-examples-please)

As the other answers have said, you will probably not test the user stories *directly*. The method I've used in the past works like this:

* **Each user story** will have one or more **acceptance tests**. These tests typically cover a high level test scenario (such as "**Given that I am logged in as a customer, then clicking the link 'My Orders' takes me to a page showing all orders for that customer.**" - the exact phrasing may vary)
* Each **acceptance test** can usually be broken down into one or more test cases. The example above could be considered to have three steel thread test cases(assuming that the customer login is pre-existing functionality):

1. After logging in, the link 'My Orders' exists.
2. Clicking the link 'My Orders' goes to the specified order list page
3. The orders on the order list page match exactly the customer's orders in the data source.

* Acceptance tests should probably come from the product owner, but they may be left for you to define (in which case, it helps to work with the product owner if you can)
* Acceptance tests are by default part of the **steel thread** (also known as the **"happy path**" - the absolute core functionality without which the story can't be complete). Tests that verify the correct function of features required by the acceptance test are also steel thread.
  + Tests for handling error conditions may not be steel thread, and tests for edge conditions probably aren't. As a general rule, **save the tests that aren't steel thread until all the steel thread tests pass**.

In my workplace, the acceptance tests typically are the acceptance criteria: "A logged in customer can click "My Orders" and see a list of all their orders in the system" is functionally the same as the **given/when/then** statement in my example and where I work is regarded as the same thing.

[Test reports for Agile software releases](https://sqa.stackexchange.com/questions/9240/test-reports-for-agile-software-releases)

Each **release consists of a couple of three/four weeks sprints**, including testing efforts as:

* Automated unit-tests
* Automated integration-tests
* **Exploratory testing sessions (during the sprint)**
* **Manual end-to-end test (each sprint)**
* **Manual regression suites (each release)**

What kind of release test report do you deliver to an enterprise client keeping in mind [Agile principles](http://agilemanifesto.org/) like:

Working software over comprehensive documentation

I have looked at ["traditional" testing reports](http://www.cambridge.org/nl/download_file/202219/) and they include documenting any findings during testing. Also I have seen test reports including screen-shots to prove the testing happened. This all feels like a great waste of time, because there is a very big chance no-one will read them ever.

**What we provided as a summary document**:

* **What user stories** were included from each group
* Any **documented defects** that were not fixed and therefore constituted a release risk
* Highlights of specific test efforts - these were not detailed, more on the order of "Executed standard automated regression suites on components X, Y, Z" or "Built and executed new performance test suite for component A"

**Defect Overview**

* **How many defects** was created.
* How many defects was fixed/closed.
* How many defects of the created/fixed/closed has a low, medium or high priority. (Maybe you has more types of priorities.)
* How many defects was found in what kind of testing phase.
* If you have different sections/components in your software, how many defects was found in what kind of section.

**Test Cases Overview**

* **How many** test **cases executed**.
* How many test **cases failed/passed** at the end of the testing phase.
* Maybe you should also show what priority the failed test cases has.
* If you have different kind of test sets for the different sections of the software split the report in these. So the people can see what section is the most critical.

We do require to provide a TER(**Test Exit Report after every Master Build** **Regression testing** which is bundle of all the fixes.

# [Fitting regression testing in a Agile/ Scrum development cycle](https://sqa.stackexchange.com/questions/643/fitting-regression-testing-in-a-agile-scrum-development-cycle)

# I've been working as the sole QA person in a 4 person Scrum team developing a new web client in jQuery. I am finding a lot of my time is taken creating and running manual test scripts on the new stories and bug fixes during each sprint.

I'd suggest that **automated tests of the steel thread of the sprint be developed as the sprint progresses, and any bug fixes get an automated test as well**. This way, you build your automated regression as the development progresses with much less pain than manually re-running each manual test script for each new sprint. This also helps to deal with the common real-life situation where the regression/hardening sprints don't happen.

By **focusing automation on the combination of steel-thread and known breakages**, you're likely to be effectively regressing the most used and most fragile areas all through development, and can use any "found" time to add automation for any other areas you consider potentially risky.

I've also found it helps to prioritize potential breakages according to likely use frequency and impact - something that's likely to happen commonly and would take the application down if it broke should have a regression test to ensure that any breakage is caught before deployment.

In short, **regression should be happening throughout the cycle as part of automated and unit tests, and continuously expanded to cover any new issues that arise as well as new steel-thread stories.**

If you have to do extra sprints before you can ship, I think there's something wrong with your definition of DONE. In scrum you should be able to ship (if it makes sense for the business) at the end of every sprint.

In agile projects, we use the definition of **done** to ascertain when to consider **a user story to be ready for acceptance (implemented and tested)**.

Unit tests implemented for new functionality and are all green (automated build and CI ensure this).

* Acceptance/story tests are written and passing.(these are in a BDD tool)
* Regression tests are green with known failures.(automated)
* Enough exploratory testing has been done to ensure the correctness of new feature and to determine it works as expected
* Unsolved defects are available in a DTS or the backlog.
* Code coverage is above x%. New implementation have not caused any regression or impact on code covergae (i.e code covergae has either preferably improved or remained static since last sprint).

# [Is seeding test data in e2e tests a correct approach?](https://sqa.stackexchange.com/questions/32427/is-seeding-test-data-in-e2e-tests-a-correct-approach)

We have this app where creating an account takes about 10-15 min and the process is really complicated. Thus we started using seed accounts which are basically already created accounts that are implemented after each db clean. **Edit: I meant we started using 'seed accounts' which is some kind of setup, so whenever we clean our db, the db itself is populated with that predefined setup (already created accounts)**

* We have plenty of test cases, that tests features on these seed accounts.
* how many **e2e tests (create account + test these features)** should we include? Because seed accounts are mocked accounts which may be different than the real ones created manually by user.

**Seeding data**

You're going down the right path **thinking about seed data** but I would also suggest thinking about how you get the data into the application under test in a way that is quick and reliable.

**For my own seed data**, instead of creating the data that the test depends on as part of my automated test, I create the data before the environment has even been deployed as a bootstrap for the database.

1. If you don't have a **minimal database** to start from, work with your developers to get it created. Ideally **your starting point** is only the data you need to get up and running "as a user" from there.
2. Figure out "the most realistic way" to create the data I want to use for testing. I usually try to mimic how consumers of our app do things.

* For me this is usually **a script using one of our REST APIs**,
* a script running selenium or similar against our app,
* or manual data entry.

1. **Run the script or do manual data entry**
2. Update the minimal database data you started from to persist your changes, **use this updated data to bootstrap your database** for applications under test. **Seems like here, you just need to copy data from your own database to the database used for testing. How to copy data from database to the other?**

When I refer to **bootstrapping the database** I mean loading in the **database schema + a set of data** we want to initialize in the environment. In my case this is basically loading in a **sql dump** that I update periodically.

So i've ran into a bit of a problem, I have a set of automated tests created for a site, were looking into really moving towards a CI based system for testing (Running the test suite after pushing features/etc...)

However in this site and a lot of my automated tests involve **creating records in the database (through the site).** Which is fine and everything, I mean these are features I WANT to exercise.

However....I can't just keep running the tests on the same database, **the test data would just keep filling up**! I could **Delete** the records from the database using the Websites ability to delete things, but **that's insanely brittle and relies me going to an exact "Delete" button on a list to do that**....seems like a poor idea. Plus if that automated test doesn't work, we get extra data...not good. **So here, even it is a chore, on the fly created data needs to be removed.**

The main concept to get used to is **having an empty database before EACH individual test**. Only create the reference data you need for *that* test and delete it after that test. For examples for zip code lookup we have 20,000 records but for testing we need ONE - 90210 - so we just create that one record for the test and delete it afterwards.

**Key here, seeded data stays the same through out the whole testing session.**

**On the fly data creation, will be deleted as cleanup/tear down part.**

**On the fly modifying seed data, changes need to reversed after testing is done as clean-up procedures.**

# [Controlling database state in Selenium UI Testing](https://sqa.stackexchange.com/questions/12218/controlling-database-state-in-selenium-ui-testing)

# We have a big test suite with around 300+ tests. Nowadays we are manually restoring database and creating few Data Objects and hard coding some data to feed into the tests. However, there are a lot of tests that need more data and we are trying to generate them run time with tests. Problem is when database changes or the test suite run on a different machine with a different version of the database they are failing

1) We **reload a skeleton database** at the start of each test run. This has a minimal set of data, as little as possible - a user, minimal config. At one point we had an automated suite to generate this database regularly to ensure it stayed in sync.

* We don't reload the DB for each test, because we don't want to slow down our test run.
* At the time we started the framework, the underlying database was changing quite rapidly, and writing SQL scripts to modify the db directly would have required a lot of maintenance. It is still changing, just not as rapidly.

2) The general approach is that **tests create the data they need via the API** (e.g. customers, orders, etc), and if they make breaking changes (e.g. to config), then they need to undo them at the end of the test. Being able to use the API to set up data rather than the UI makes this much faster to do at run time. You can create run time data with rest API.

It is pretty easy to run the tests on **another machine** with a **different version of the database** - we just need to set up the automation user and change a few config settings (if they're needed by the subset of tests we're running). **We have very little dependency on the actual underlying DB design** - you could change 99% of the tables, and as long as the application code still worked, we'd still be able to run our tests. This has meant that for example, we have been able to use the UI tests to detect issues when making database changes in areas that aren't well covered by integration tests. This might not be an issue for you.

In this case, we actually don't do that within the test framework - we **run the tests from our CI server**, and have a step that reloads the db in there (just something like **"mysql testaccount < skeleton\_database.sql**"). We do actually **have a service written by the developers for use of the integration tests that we could use to run SQL against the db**

1. **One separate database** which contains all the data for the tests: data-driven testing. This means you'll likely have to compose Views (combining tables as necessary) which serve as the data source for test methods.
2. **Insert/Update/Delete scripts as a stored procedure on the application's database**. **Have your test class execute these scripts before running any tests**. This way you force the 'variable' data in the application to a specific state which your automated tests expect.

Explanatory example:

1. For a test method 'Customer login'**, the Selenium database (Application Database prepared with this info before test started as seed data) provides customer@login.com and password 1234**. So the test method expects a customer profile screen after logging in with this data (if not it will fail).
2. The insert/update/delete scripts are executed before running the case. It will (1) delete the existing customer@login.com profile in the application database, then (2) insert the profile with the correct password and settings. This way **we know for sure** our test data is correct. If we don't execute this script, the profile could've been changed by manual tests, the password can be obsolete, ...

I would recommend taking a database snapshot which you can easily (manually) restore before starting the test suite

**UI testing**

My approach to this is to make sure that the UI testing focuses on the functionality of the UI itself. In other words, from a user perspective the text and links are correctly displayed and - in particular - HTML forms and input fields work as expected.

The way to avoid having a large number of UI cases is to **avoid** what I have come to term **'data combinatorial testing'**. By this I mean any test that says 'given this extra piece of data "x", I should see "y". "x" can be data input by the user, data retrieved from a data store, data from an external service, etc. This should be tested on the back end.

An example will probably help:

A business operates in 50 states. Each State has a different welcome message targeted to the local culture. The business might ask 'can we have 50 tests please to make sure we have all the messages correct?'. **It would be possible to write 50 browser UI tests,** one for each state that verify the information. However, the correct way to do this would be to recognize that it isn't "the browser" that determine the text. It is something on the back end that says "when I am passed the text "Wyoming" (for example) I will return the text "Home of Yellowstone", etc. So we should write back end unit/integration tests that test that each state passed to the routine that determines the message results in the correct test. **Then we test one state in the browser to make sure that the lookup works.**

I like to keep a **regression testing google doc spreadsheet** which I then insert links to relevant bugs in Jira. That way you, fellow testers/devs/PMs, can all see previous bugs and get an idea of which areas are vulnerable over time.

Then for each release, you use the same doc**, just adding a new column**. That way, you have everything in one place, rather than multiple excel files that are difficult to go through.

The faster I get those bugs back to my developer, the faster he or she can commit a fix. The earlier in the process I do that, the less likely it will be that I’ll need to re-test things that the new commit might have impacted.

1. **Test the most basic happy path surrounding NEW code, for 10 minutes.** If you aren't able to perform the most basic functions of a new feature, you need to get the code back in the hands of your developer as soon as possible. Chances are, if something is broken here, it could be a longer fix.
2. **Test functionality that impacts a large percentage of the code base as a whole.** This is where we find dependencies that we didn't think could possibly be impacted by our changes. It’s much better to find these early, because they could indicate a need for deeper code reviews, and also wider regression testing and an adjustment to your test plan.
3. **Test the critical functionality, and the most commonly used areas.** What does your app NEED to be able to do? What do most people use it for? What are the potential blockers that are the only reason some users use your site or app?
4. **Test the potential areas of high risk.** Data loss. Load/Performance issues. This should be part of a conversation you have as a team - what are the biggest risks involved in this sprint? What are the worst-case scenarios? This is also the time where it’s most beneficial to test in a stage environment that replicates production as closely as possible.
5. Then normal regression testing on down to edge cases and thinking of creative ways to break stuff (my favorite part).

**HTTP**

# [How to wait for an api request to return a response?](https://sqa.stackexchange.com/questions/29379/how-to-wait-for-an-api-request-to-return-a-response)

HTTP is a synchronous protocol\* so active polling is not an option. You need to wait until client receives response or request times out. There are two ways to constrain synchronous behavior with timeout.

**API**

# [API call returns 400 bad request even when the request is correct?](https://sqa.stackexchange.com/questions/30917/api-call-returns-400-bad-request-even-when-the-request-is-correct)

I made a post api call and got a "400 Bad Request" in the response. The response body has a one line message about an unspecified database commit transaction failure. After checking several times, I found that my request is 100% correct. Other people found the same issue too, despite making correct requests.

Work with the maintainer of the API and explain the issues and ask them to adjust the code on their end so that the error generates a 404 return code ('resource not found', applies to any entity for which a request was made but not satisfied) instead of a 400 - Bad Request.

Failing this or a quick response from the maintainer, when you get a 400, check if the body has the "unspecified database commit transaction failure" message and if so treat the error as a 404, resource not available (it applies to more than just pages, hence 'resource') and act accordingly.

# [Why use an API test if I can get the same information from the website?](https://sqa.stackexchange.com/questions/41301/why-use-an-api-test-if-i-can-get-the-same-information-from-the-website)

Why should I check the server response by using a GET call through an API if I already have a button on the site with the functionality to call the same method?

If you have an **API that's retrieving information** **and a website that displays** it, you test the **API to make sure that the correct information is retrieved**. Then you test the **website to make sure the information is displayed properly.**

Say you're looking for products in a category. Let's say the category is "**hair care products**", and you'd expect to see shampoo, conditioner, hair ties, hair clips, and the like. **If you test the web site, and get** pet shampoo, carpet shampoo, or car shampoo in the list of products that comes back, **you don't know whether the problem is with the website** asking for "shampoo" rather than "hair care products" or with the API (or with the database). You have to check what exactly was requested and then trace the API response to the request before you can work out what was wrong.

**Here's why you should avoid the UI when testing APIs:**

* UI tests are **slow**
* UI tests will not show **API level error codes**
* UI tests require browsers (even if headless)
* UI tests are **unstable and will occasionally fail**
* UI tests increase the feedback time to developers
* UI tests require the code be deployed to an environment
* UI tests are brittle because **UI changes can affect the API call being made**
* UI tests will **reveal UI problems unrelated to the API** but affecting the use of it

If you started with the API request and got the products you expected, then you'd know as soon as you performed the web test that the problem was with the website. If you started with the API request and didn't get the products you expected, then you'd know the problem was with the product categorization or the API. You could make a call to retrieve all products and check their categories to determine the source of the problem

  We recently migrated an complex front end application from angularJS to angular 8 while utilizing the same API for data plumbing with little changes in geographically distributed sub-teams.

It helped us testing components directly to isolate & resolve issues quickly by putting issues in right buckets of **front end** vs. **back end** vs. **services** otherwise testing would have been **very messy, time consuming & expensive through UI**.

# [Should QA or tester mention the root cause for a bug?](https://sqa.stackexchange.com/questions/27962/should-qa-or-tester-mention-the-root-cause-for-a-bug)

For white box testing, maybe. For black box testing, no. The whole point of black box testing is that you're testing based on the specifications, not the code. A **black box tester** shouldn't care why the specifications aren't being met, just that they aren't being met. And knowing details of the internals may actually detract from black box testing.

**Black box testers** have good understanding of the **requirements** and the overall functionality to be tested. But, since they are viewing the system as a black box; just relying on the requirements. Because by definition, black box testing **doesn't require the testers to have in depth understanding of the implementation from coding stand point**. So, generally, project teams assign testers who are very thorough with the requirements, smart while thinking about the possible test scenarios but these candidates are generally less technical (if we compare with white box testers who dive deep into the code).Having said that it is very difficult for black box testers to identify the root cause of a defect when it falls under one of the following:

The [testing manifesto](https://www.growingagile.co.za/2015/04/the-testing-manifesto/) states preventing bugs over finding bugs. With this in mind I think you should do **a**[**root-cause analysis**](http://www.jamesshore.com/Agile-Book/root_cause_analysis.html)**for each defect** that was found. Afterwards find a strategy or experiment to prevent a similar issue in the future.

I think critical thinking QA people could be of real added value during the analysis. Whenever a defect happens I like to grab a developer, QA and business stakeholder and draw a root-cause mindmap and analyse the issue deeper. Do it together, share knowledge, solve real problems on the correct level.

# [API cache testing](https://sqa.stackexchange.com/questions/41375/postman-or-any-api-tools-for-api-cache-testing)

It is hard to test cache functionality! **how would they know the response is cached**.

* Call API with parameters
* Verify results
* Stop data storage (e.g. database)
* Call API again with same parameters
* Verify same results

How would you do it manually? Does that translate to an automated test?

* Check speed of response in difference? Is the second time faster? (if you can measure that)
* Can we make it testable by
  + introducing data in the response saying its cached
  + adding logs of each request to the database and verify it was not accessed by looking at the logs

I usually test by **stopping sql server then query again**, **if result returned mean cache works**.

# [How to verify response body as a test in postman](https://sqa.stackexchange.com/questions/37963/how-to-verify-response-body-as-a-test-in-postman)

**Create** test cases for each assertion

var jsonData = pm.response.json();

pm.test("Verify data ID", function () {

pm.expect(jsonData.data.id).is.to.equal(2);

});

pm.test("Verify first\_name", function () {

pm.expect(jsonData.data.first\_name).is.to.equal("Janet");

});

pm.test("Verify last\_name", function () {

pm.expect(jsonData.data.last\_name).is.to.equal("Weaver");

});

**If you really wish to compare the full body**, you can **create a variable** with the expected outcome in a pre-request script like so:

var **expectedJsonBody** =

{

"data": {

"id": 2,

"first\_name": "Janet",

"last\_name": "Weaver",

"avatar":"https://s3.amazonaws.com/uifaces/faces/twitter/josephstein/128.jpg"

}

}

pm.environment.set("address", JSON.stringify(expectedJsonBody));

in your request test, you compare the response body with the variable.

**After the**[**obvious - Status code is 2xx**](https://learning.getpostman.com/docs/postman/scripts/test_examples/) you need to add tests based on [**content**](https://learning.getpostman.com/docs/postman/scripts/test_scripts/)**:**

var **jsonData** = pm.response.json();

pm.test("Verify first\_name", function () {

pm.expect(jsonData.first\_name).is.to.equal("Janet");

});

**9/21/2020 🡪 10/11/2020**

**9/9/2021**

* We have manual scenarios automated using selenium , but still here we are calculating the coverage of dev code (web app code) from test project , so that we can increase the test coverage to a certain extent , could sound wierd but its a part of requirement.

Coverage.py counts the total number of possible executions. This is the number of executable statements minus the number of excluded statements. It then counts the number of those possibilities that were actually executed. **The total percentage** is the actual executions divided by the possible executions.

Code coverage is a measure of how much code is executed in response to a stimulus (e.g. running a test). Test coverage is a measure of how much of the feature set was executed as a result of a test.

**Code Coverage**

To begin with, let’s learn about code coverage. The main purpose of this metric is to calculate the number of lines covered by the test cases. It reports total number of lines in the code and number of lines executed by tests. Think of it as the degree to which the source code of a program is executed when a test suite runs. The intent is, the higher the code coverage, the lower the chance of having undetected software bugs.

**Subtypes**

Coverage is split into several subtypes – condition coverage, branch coverage, loop coverage, function coverage, statement coverage, and parameter value coverage.

**Let me define the subtypes for the sake of clarity.**

· If testing cover each function calls in code, function coverage is called to be 100%.

· Similarly, if all branches in the code, i.e. all if-else conditions have been tested with every possible input, then branch coverage is said to be 100%.

· If all loop statements in the code are executed then loop coverage and statement coverage would be 100% respectively.

· In an ideal case, for condition coverage to be 100% every single condition with all feasible combination of condition parameters are perfectly tested. One can check the Parameter value coverage by passing all common values to the criteria in the function call.

**Coverage testing tools**

These testing tools are obtainable today for each major languages. **Note, that mostly are relevant only for unit test** and perhaps not for the complete tests performed ( system, integration, Unit, manual and more).

**Now have a look at few of them to understand the code coverage tools scenario.**

**Cobertura** is a **Java code coverage reporting tool**. It is based on coverage and works on any platform with Java 5 or higher. You can use Cobertura with ant, maven or command-line access. While instrumenting classes, Cobertura generates Cobertura.ser file containing basic information about each class. This file is utilized when instrumenting, running, and creating reports. It also uses ASM when instrumenting. On running testing, it creates a report comprising each package, all class within every package & corresponding branch coverage, line coverage, and complexity.

For **Python**, a well-admired tool for code coverage measurement is Coverage.py. It functions in 3 different phases –

· Execution, where it runs the code and monitors it to see what lines were executed.

· Analysis, where it examines the code to determine what lines could have run.

· Reporting is where it combines the results of execution and analysis to produce a coverage number and an indication of missing execution.

This measurement tool extensively used for Go is gocov. This kind of tool provides 4 key commands – test, convert, report & annotate. Making use of these commands, you can generate textual reports with annotations giving coverage information of each function in each go file. A GUI wrapper is also accessible for gocov called as GoCovGUI. It displays the entire report generated by the tool in GUI which is easy to understand. It also has wrappers to generate reports in HTML and XML format.

**Code Coverage:** As a human, we can make mistakes or miss some important issues which lead to errors and defects in a product. Testing thorough makes a product high quality. Performing Unit testing is a good start, but it is sometimes is difficult to measure or improve.

**Code Coverage**

To begin with, let’s learn about code coverage. The main purpose of this metric is to calculate the number of lines covered by the test cases. It reports total number of lines in the code and number of lines executed by tests. Think of it as the degree to which the source code of a program is executed when a test suite runs. The intent is, the higher the code coverage, the lower the chance of having undetected software bugs.

**Test Coverage**

Test Coverage aims to measure the effectiveness of testing in a qualitative manner. It evaluates either the test cases are covering whole functional needs. You can believe it as a type of **black-box testing** when test cases are not written contrary to code but based on user requirements or expected functionality.

**Subtypes**

The common structure and means used for test coverage measurement are **unit testing**, functional testing, performance testing, integration or system testing and acceptance testing.

**Unit tests** are written at a granular level to check if a function/method performs as expected. In **functional testing**, each functionality mentioned in the requirement document is tested. **Performance testing** is normally a method to stress test the code for its stability, scalability, and responsiveness at various workloads. **Integration testing or system testing** is done to test if the completely integrated product works in an expected manner**. Acceptance testing** is generally done at the end of the development cycle. For acceptance testing, the product is handed over to the stakeholders and tested by them to determine whether the product is in an acceptable state.

I'm with Alan on this one - a lot of test automation code will cover error handling which may not get exercised often - and if you're really lucky, it won't ever get exercised (you do include error handling for every routine, right?)

The most effective coverage evaluation in our experience was a two-pronged approach with thorough unit requirement testing (such as in TDD) combined with exploratory testing of the user stories at the system or integration level.

The combination of unit requirement coverage and exploratory evaluation of the user stories at the system level should give you the best coverage indication.

The approach I've seen most often is to have the unit tests as a separate project in the application solution, so they can be run against production code as part of the build process, but are not included in the production code.

Some of the reasons for the separation are:

* Logical separation. You want your unit tests to catch issues with initialization, global variables, and scoping, which is more difficult if they're part of the main code base (depending, of course, on the language you're using).
* No need to worry **about compile-time flags** - it's remarkably easy to miss adding a compile-time flag to keep your test code out of your production build. If you have a separate testing project in the solution, this isn't an issue - but you also get to keep the convenience of matched check ins, because it's all in the version control system, right?
* No shortcuts. Because the **test codebase is in a different project**, you have to perform all the initialization, setup, and teardown. You can shortcut if the test code is part of the application project - and those shortcuts can mean invalid tests.
* You don't have to link test *data* to the application code. This can be a big issue for large and/or complex applications. **The test data source is linked to the test project, not the application code**, so it doesn't slow the build and check in/check out process for the application code (and you are running the unit tests for every full build, right?)

Some people distinguish between

* integration (some parts are used for real, some are mocked) and
* system/end-to-end testing (where is no mocking at all).

For other people, system, integration , and end-to-end are synonyms.

Usually end-to-end testing uses just basic scenarios ([Pareto principle](https://en.wikipedia.org/wiki/Pareto_principle): 20% of use cases cover 80% of functionality), and unit testing tries to cover also more of the corner cases.

For e2e, testing just basic functionality is OK. Cover ALL corner cases in unit tests. If you have more time for e2e, start adding more common corner cases.

Unit tests are more useful to detect where exactly the error is (so covering all corner cases makes more sense, and is easier with mocking).

#

To my understanding the unit tests will mock everything, should i be almost duplicating those tests because the webservice is deployed on the real server and running as part of the application (utilised by the front end)?

Should I be testing the webservice calls for all sorts of input on the data?

OR, should i be writing a handful of tests to flow through the API to ensure its working correctly? Right now I have hundreds of input related tests checking for all sorts of data input which I feel is unnecessary, I am testing each call atomically for input and scenarios like deletion, adding a new, duplication etc.

I'd **firstly look at testing the parameters for a given service**. E.g. if a login API, make sure you get a meaningful error for no username, invalid credentials etc. etc. As mentioned elsewhere, you'd be thinking, "thats's hundreds of combinations". Peter Masiar's point around test most common first then edge cases as time permits. In terms of your test, again using the Login API as an example, in SoapUi, I'd created a **single data-driven test**. **I'd then put my scenarios into Excel**. The **data driven test** calls the Login API for each row in the spreadsheet. In the Spreadsheet, you can put in your params as well as the expected message to be returned....

Test Id Test Description Username Password Expected message

1 No credentials supplied. No user or password supplied

2 No username Bar Invalid credentials supplied

3 No password Foo Invalid credentials supplied

4 Wrong password Foo Barrr Invalid credentials supplied

5 Valid credentials Foo Bar Login Successful

The beauty of the data driven test is you minimise the number of 'tests' in your suite, but get the benefit of testing lots of combinations. Plus, when you think of a new edge case, e.g. invalid chars in username, then you just add a row to the spreadsheet.

Once you have tested a web service for positive and neagative values, then think about **user cases**. E.g. saveCourse and getCourse. In a single test, I'd create a course and then with the values I supplied to saveCourse, I'd attempt a getCourse.

Once you have confidence in the API and the GUI is less likely to change, then I'd look at automating the GUI....

#

Specifically, this is a **Java application** using ***Maven as the build manager*** and Subversion as the version control system, with most developers using a **NetBeans IDE**, and also using Spring heavily along with some new JMockit and AspectJ code.

On a project I've been working on, we've been steadily increasing the code coverage through tests. Some of these tests were done as integration tests and some as unit tests. The integration tests can sometimes take a long time, thus discouraging users from running the entire test suite prior to checking in (and **understandably so**).

**For the checkin-runs you can mark the long-running tests with their own**[**category**](http://www.junit.org/node/508)**and tell the test-runner to exclude those long-runners**

You may also look at [Is there a way to separate long running (e.g. stress tests) out so they're not run by default in Maven 2?](https://stackoverflow.com/questions/251730/is-there-a-way-to-separate-long-running-e-g-stress-tests-out-so-theyre-not-ru)

To answer question 1: Yes, I'd separate the unit test code from the integration test code. They're logically different, are probably built differently, and you're planning on running them differently. I'd also name them differently too for extra clarity. For example, our unit tests are in a ut\_\* directory and named ut\_classname, so I'd put the integration tests in an it\_\* directory and name them it\_something.

With regard to code coverage, I'm not familiar with Maven or Hudson, but one strategy is to run only the unit tests for every commit (= fast) after the main build loop. Of course, the code coverage will be lower. So you have a second build loop that is triggered after the first that will run both fast unit and slow integration tests to give you the combined coverage. Or if the integration tests take a very long time, you could move them from a triggered build loop to an hourly/nightly schedule, although the more frequent the better.

As an overall approach I would follow a **gated principle**. **Run the unit tests first and then if they pass allow the developers to check in,** **once checked in, then run the integration tests and if they passes then commit the code to the main branch or trunk**.

You want to keep your developers agile and allow them to work as quickly and efficiently as possible, but still have the integration tests as a quality "gate".

# Use dummy database

The only thing I would be worried about is ongoing database development, e.g., **new columns or tables being added that the code expects to have there could invalidate your old data**. Just make sure you have an idea of how much work you will go through to keep your dummy DB up-to-date.

* **General workflow**:

1. Jenkins polls for changes in the project every 5 minutes and builds everything if it finds any.
2. Also, I have nightly builds no matter what.
3. Regarding my development workflow, I just run all the tests from time to time during the day and always before committing.
4. I don't find that ideal because **the tests in the device (Instrumenation and Robotium) take a long time**, so I would prefer instead to run only the deviceless tests while developing, and all of them before committing.

##

##

**Background**

### Selenium Test

# Shell Script

##

Test efforts should use **the Agile Testing Quadrants** to see where Unit tests fit in. Also you should respect the test pyramid with most tests being unit tests and a much smaller number should be automated UI. So I recommend starting by making sure this common knowledge is shared.

End to End UI automation should test the front end and whether the user can get through the screens and understand what's going on and the screens work appropriately to provide/collect information.

So... unit testing is intended to:

* make sure a routine works as intended
* verify data combinatorial testing without using the UI
* ensure that a function works according to specification

With this knowledge, you will have a good sense of what should be tested where.

The fact that you have unit tests that test data combinations and different inputs mean that you do not have to test all those various combination through the UI (what I term 'data combinatorial testing).

I am seeing this right now in my work where traditionally we have 1700 UI tests. We have re-worked this to be 70 UI tests and are now in the process of creating the thousands of unit (and integration) tests that the 1700 tests was mostly testing.

# [How to approach setting up test data for a project that has a microservice architecture?](https://sqa.stackexchange.com/questions/42765/how-to-approach-setting-up-test-data-for-a-project-that-has-a-microservice-archi)

**How to organize your repository**

It is not only matter of your test strategy but also how you work as a team.

* Are microservices owned by different teams? Is the code for those microservices in the same repo?
* Who is going to maintain and update those tests? Only you, other testers or also developers of microservices?

I usually **prefer to have tests close to the code they test**, so if anyone new comes to the project, she can start automation with one or two code checkouts.

* How will you launch those tests in a development pipeline? Do you want to have one single status whether all tests passing or failing, or you prefer to launch first tests for single services and only if they pass launch end-to-end tests?

**You do not need separate repos for each group of tests**. Packages enable organizing structure of tests in a way you wish, so it is easy to grasp test coverage looking at your packages names and structure. Then there are test groups that can be used to annotate your tests. They will ease launching separate groups of tests, especially if you want to have separate jobs in your CI server (e.g., Jenkins).

* Whether to have a separate repository for service stubs, mocks and other helping tools is another question. Are you going to use those tools across other testing projects as well? Are other people going to use those tools outside of your project?

.

In this case, we actually don't do that within the test framework - we run the tests from our CI server, and have a step that reloads the db in there (just something like "mysql testaccount < skeleton\_database.sql"). We do actually have a service written by the developers for use of the integration tests that we could use to run SQL against the db - so I guess we could use that if we wanted to call it from within the test framework, but as it happens we don't. – [testerab](https://sqa.stackexchange.com/users/103/testerab" \o "4,985 reputation)

* Selenium test suite and DB test suite are built and executed independently, e.g. they are two separate test suites.

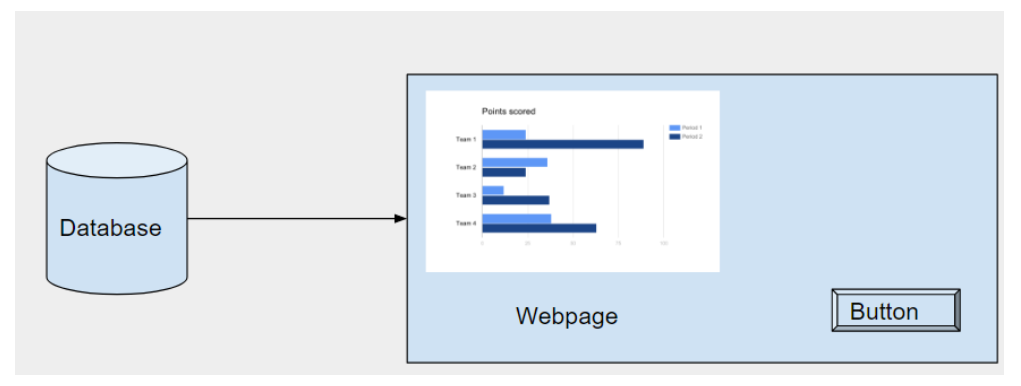
## **Scenario Walkthrough:**

Front-end need to use the data from database. If the DB testing and Selenium test are built and executed independently,

* How can selenium know the data sent from database?
* If selenium cannot know the data, then how can selenium compare with the data showing in the graph and data received from database? Or should I use data driven test to do this part?

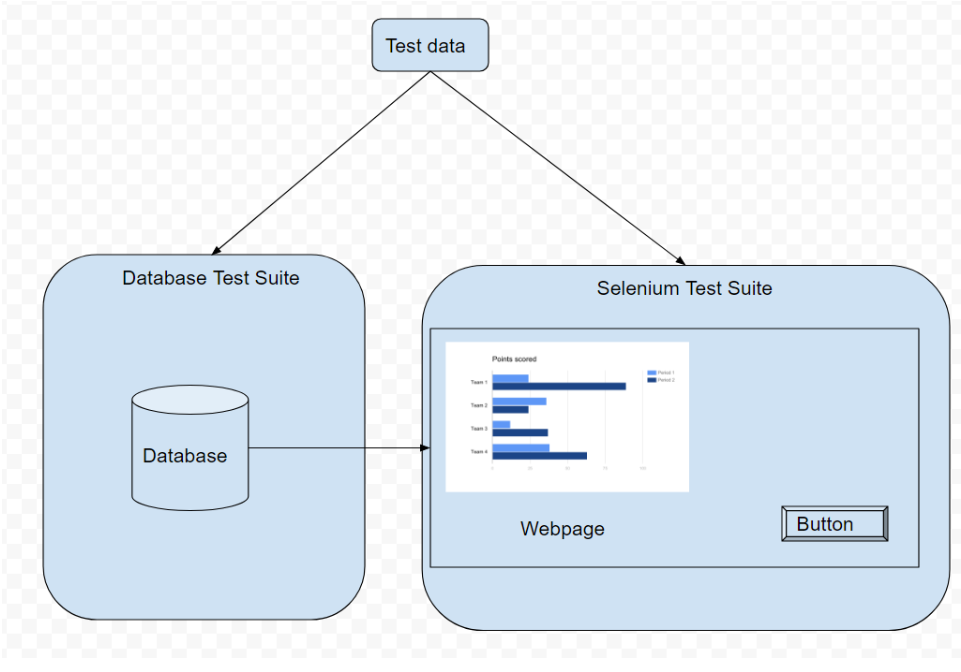
Say, below it is the website under test.

* When the button is clicked, this webpage will fetch a whole bunch of numerical data from its database, convert numerical data into a graph and display it on the website.



After you take over, you have developed two test suites, one is Selenium test suite and another one is Database test suite.

* You have created a data file that contains some pre-conditions and expected results. How you collected those data? You can achieve those data from playing with the website and / or making SQL query into the backend database.
* Selenium test suite will click the button, read the generated graph, convert the bars on this graph back to numerical values and compare those numerical values against expected results. Selenium does not know how the graph is drawn or where the data come from; hence front end testing.
* Database test suite will simply make direct SQL queries into the database; those SQL queries will fetch the same bunch of raw numerical data from the database; database test suite can then compare them against expected values from the test data file.
* **You may wonder what kind of SQL queries you need to pass onto the database server in order to fetch the same set of data, you may need to ask this webpage's developer in order to learn which SQL queries to use**.



# [How to deal with automated test cases that always fail?](https://sqa.stackexchange.com/questions/3874/how-to-deal-with-automated-test-cases-that-always-fail)

I'm facing a lot of automated GUI tests that fail one or more verifications during their run. All the failings points have been reported in our tracking system and they are known issues but get "ignored" as having low priority.

Development team promised to have those issues fixed "in the next few months", and this was many months ago.

Should I edit the tests to pass (for example, commenting out the verifications)? Or should I keep bashing on the dev team until they fix those minor display issues? What are the "best practices", if any, in this case?

# [Differences Between Testing a Web Application and a Desktop Application?](https://sqa.stackexchange.com/questions/3068/what-are-some-unobvious-differences-between-testing-a-web-application-and-a-desk)

* Desktop software usually requires **installation**. Web applications usually do not.
* But web applications are sometimes expected to be running 24x7. This can make upgrades and maintenance more of a challenge to plan and execute (and thus test)
* In addition to browser versions mentioned by others here, you may need to worry about browser **add-ons**
* You may also need to worry about the myriad **browser settings** that could cause issues with your web application
* Some desktop applications are designed for single users. By definition, almost all web applications are multiple-user systems. You need to worry about **concurrency**, what happens when the same user is logged in multiple times, etc, etc.

I do not think the UI testing is very different: field validation, default values, resizing, **scalability**, and so on.

You probably need to support more than one brand and version of web browser, and perhaps even some mobile devices. You may want to s**eparate your business logic tests from your browser-level tests** so that you do not repeat every test on every browser.

Security testing is different for web applications. This is a big subject, and it helps to have someone who specializes in that kind of thing.

For a web app, controls can be tested by input simulation as well as by JavaScript API calls (ie calling click() or onhover() methods). This can lead to subtly different behaviours between the cases

* **Usability** - Desktop applications tend to have a help file built in, where web applications should be more or less self-explanatory.
* **Load times** - this one is a big pain point. Not everyone has broadband (and we won't go into how much I despise pages that have so much advertising attached that they take eternity to load despite broadband) and there are quite a few people who operate on extremely limited connectivity. You should look to test that your site is functional within a reasonable time over a 28k modem connection (typical speed in older apartment blocks in the USA) or that a low-bandwidth version of it is available and easily located.
* **Cross-browser and operating system compatibility**. This is particularly important if your site uses a lot of client-side processing or the latest functionality from a specific browser. If the browser in question is Internet Explorer, you lock a lot of people out of your site.
* Clear indicators of processing - if your site needs to do any potentially lengthy processing (such as live validation of credit card information) you need to make sure it disables as much as possible and shows a clear indicator that it's working or users will assume it isn't and try again - which can cause problems.
* If you're testing a GUI that interacts with some back-end service, and your only concern is that what you enter in the GUI gives you the expected outcome when the service is done, it doesn't matter what type of back-end service is used (An example might be testing credit card authorization in your system with a new protocol - you don't care how the protocol is implemented, all you care is that the test card number that should give you a approval actually does give you an approval).
* On the other hand, if you're testing some feature or aspect of the service itself, then it matters a great deal: you need to know what format to put the request in, what format the response should take, and you need to be able to read the logs if the response doesn't give you the result you expect. If you're automating, the kind of service has a big impact on how easy/difficult it is to automate in your specific tool.

First, try other websites. Ideally, ones that are logically close to the website that you're testing. For example, if the website is hosted on AWS, try other sites hosted on AWS. You can use traceroute and the like to determine where the website is logically located.

Second, look at logs. You can see from apache logs when a request came in, and possibly how long it took to complete, depending on what's being logged. You may be able to follow a request from apache/the web host to the database and back.