API testing

6/30/2020 🡪 10/30/2020 🡪 1/30/21

P36

2/2/21, 2/12, p7

4/1/21, 4/22/21

My own test:

1. **python automate testing endpoint of different version**
2. Database contains user-defined or predefined template for work – flows.
3. **Work flows** are tests that user can perform
4. Database contains previous test data/results.
5. How your python scripts call your own API and verify it.
6. What now is python requests calling end point with GET, PUT, Patch, POST.
7. End point + HTTP methods vs Calling API directly.
8. 🡪 HTTP client API for sending request vs Server endpoint API
9. Web API as endpoint or offering SDK as functional API
10. **Code coverage is done in unit test; coverage > 95%**
11. **Test coverage is done in both end-to-end, and functional testing.**
12. **Regression test for each build. Covers end-2-end and API work flows.**
13. **Functional testing is run separately, for edge cases and features verifications.**
14. **CI/CD**
15. **Trouble – shoot Request Failures**
16. **What is mysql dump files**
17. How to set up endpoint URL for different API version

**Check General**

**API testing**

**API automation testing in python**

[**https://www.altexsoft.com/blog/api-testing/**](https://www.altexsoft.com/blog/api-testing/)

**https://www.fleekitsolutions.com/api-automation-using-python/**

<https://www.sisense.com/blog/rest-api-testing-strategy-what-exactly-should-you-test/>

<https://www.freecodecamp.org/news/how-to-automate-rest-api-end-to-end-tests/>

**Typical My daily testing on API**:

1. Dev. Run unit test before check-in.
2. Dev check in to **feature branch** if unit tests pass and functionality works.
3. **Jenkins CI/CD trigger QA build test**

* **What kind of build, java-based or python based?**
* **How does build work?**
* **Test environment: Build server**
* (Smoke tests) **End to End UI test** + Multiple **work flows on API**
* Plus **New feature test** if available
* Takes about less than 20 mins to finish.
  1. **QA’s task to verify new feature just checked in branch**
* **Test environment: QA server**
* **Does QA server run on Jenkins? How to get the build to test?**
* Verify the feature manually if necessary
* Create python script to verify;
* Once verified, work on integrating the test into master test suit, that is used for **nightly regression test**.

1. Build engineer will merge changes to Master if build tests pass.
2. End of work day, kick off **night regression test on Master branch**

* **Test environment: QA Server**
* Run legacy tests: feature/function tests, UI end-to-end, and API work flows.
* Newly added feature, UI or API work flow tests
* Nightly regression test takes about 5-7 hours for now; as we add more test cases, it will run longer in the future. But, we also optimize tests so it will not be too long.
* Regression test of steel thread(**Happy path**) of the sprint should be developed as the sprint progress, with any bug fixes added.
* Focusing regression test with a combination of steel-thread and **known breakages**; regressing the **most used and most fragile** area (bug fixes and new features)
* Regression should be happening throughout the cycle, and continuously expanded to cover any new issues that arise as well as new steel-thread stories.

1. **Scrum and Sprint** meets every week to discuss **new feature/ user stories** development, **testing** and **bug fixes** schedules.
2. **Once identified as formal release in sprint, Build/dev-ops engineer will run smoke test on Staging Server**.

* We can have a staging server, that can mimic production environment.
* Who maintains it and runs it; (Build Engineer)
* **QA mostly focused on Feature/functional Verification + End to End test.**
* **Do we run performance test in here**? (**performance Test run By build Enigneer** when Formal release is decided. Normally, performance is similar.

1. **Exit Criterial**

* **Unit tests** implemented for new functionality and are all green before checking in to branch.
* 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. Done by QA on CI triggered by CI feature branch check-in.
* 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 coverage (i.e code covergae has either preferably improved or remained static since last sprint).

1. **Scrum meetings**

* As a tester and experienced working on Agile, **We had daily scrum meetings** to discuss our targets to be met for the respective sprint.
* The targets as a whole impacted minor targets in test strategy and items to be tested immediately.
* Test Automation was used for repetitive test cases.

1. **Scrum Master** (more or less like a **project manager or program manager**)

* Classically, the SM role is to *facilitate* progress by finding ways to clear or work around any impediments that arise during a sprint, to lead meetings, to ensure that meetings stay on-topic and stay within the time frame, to assist with coordinating resources, and so forth. SM is basically there to assure the project is on target and on schedule. Ideally, the SM role should *not* be held by someone who has another role in that project.
* If I've interpreted your question correctly, your SM is also a dev on the project team: this is, as you've experienced, not ideal. If this is the case, I'd suggest your first priority would be to try to arrange future projects so that each team member has exactly one role - that is, your SM is not a developer (or QA) on that team.
* Regardless, I think you should try to have someone take Scrum Master training and become **a dedicated SM** for as many project teams as possible - it's very much an administrative/resource-juggling role compared to the Dev/QA roles which can and should be blurring in the teams

1. 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 very sprint**. IMHO this is the wrong answer because you are starting to fallback on the older waterfall habits of 'testing in the quality at the end' instead of making sure that as each item is done you are truly finished with it and don't have to think about it again.
2. Automation should itself be a task in Kanban/Scrum. It's possible to get to the end of a sprint and not have all tasks on the sprint done but supposedly, while you're running the sprint, you prioritize things so that stuff for any one user story is not half done. **That allows you to ship SOME. Anything left over is backlogged to the next sprint, that includes automation**.
3. "Scrum is the best agile approach. In scrum, **each sprint produces an increment, which is a potentially releasable product**. Here, **each increment must satisfy all acceptance criteria and pass the different categories of tests**. Regression tests are a tiresome activity, especially in an agile process, which is characterized by **nonstop changes and frequent deliveries**.
4. **It's all part of being ready to ship at the end of any sprint.** scrum means everything should be done is done in the end of the sprint
5. Then get your CI system setup so that the **regression tests** run on a **regular basis (I'd recommend nightly)**
6. **Master Regression** - would be a big set of test folders with merged test cases under **each functionality** and **the new functionality or changed functionality** will update existing test cases
7. ITERATION Regression - After every iteration ends, we are doing a manual regression of the stories/test cases in the next iteration. We test all the functionality of the previous iterations. In the upcoming iterations we do the iteration + 1 testing.
8. **Example of a sprint flows**

* We use to do gather as a team for Sprint Planning meeting(Where we decide, what can be delivered and which user stories to be delivered first/last in a sprint)Our sprint is **for 2 weeks duration**.
* Scrum master will be ready in Sprint planning with any planned leaves from key resources and any Public holidays coming in a sprint etc and will decide the Total Team Effort.
* Including tester's, BA, Dev's everyone will gather and it's **Scrum Master** turn to **explain each user story with Acceptance Criteria of completing it** on Digital screen.
* Then Dev's, Tester's will discuss on the **flow** about the effort involved or any dependencies etc.
* We play [**"Planning Poker"**](https://www.mountaingoatsoftware.com/agile/planning-poker) game to give **individual estimates**, which are called story points. We decided that 1 point = 1/2 day work every company has their own definition of a story point.
* Finally, after noting down every individual story points **Scrum master** will come up with **Actual Story point** (which is the most common Estimate given by the team members)
* Let's say PersonA thinks it's gonna take 2 points PersonB- 3 Points PersonC- 3 points PersonD- 1 point PersonE- 5 points
* Then the most common estimate we got from team is 3 points so this will become the Estimate for the Story.
* I like to break each user story down in three different passes:
* First pass, identify functionality and features
* Second pass, identify any limits imposed by a given environment, operating conditions, etc. This can be things like performance/throughput, duration, security.
* Third pass, identify:
* Research [**Exploratory testing**](https://sqa.stackexchange.com/questions/30079/is-there-any-standard-or-pattern-of-doing-exploratory-testing/30081#30081) and figure out ways to come up with new test ideas.
* Interfaces (other software, files, databases, peripherals, etc.)
* Inputs and outputs (records, messages, files, reports, objects, etc.)
* Configurations (System level, application level, hardware, network, etc.)
* Usage scenarios (normal "happy path", error handling/recovery, backup, installation,etc.)

**Create test in Sprint**

1. In agile projects every sprint timeline we have:

* **new functionality** to be tested (the size of this part is nearly the same every sprint)
* **regression & verification** of bug fixes (and this part becomes bigger from sprint to sprint)

### New functionality

No automation, More informal scripted tests (test cases), Less Exploratory testing

Commonly testing in agile is started when new code arrives, but in this case testing loses precious time. Ideally tester should participate in scrum meetings to have full understanding of user stories, so he could represent end user interests. This early phase gives you sketches of test cases.

Then as software arrives you check it with this informal test cases (extending them according to your new knowledge of the software). This is the first phase of testing, scripted manual testing (there should not be too much exploration) to check basic functions of software. This is how you test **new functionality**.

### Regression & verification

More automated tests, Less scripted tests (test cases), and Exploratory testing as a separate part of testing process

As soon as core functionality becomes stable (business process or part of it could be accomplished) you should start with automation. **Core business functionality should be validated with automated tests.** This will save you time on **regression & verification** to do exploratory testing without scripted test cases but with great mind efforts. Be sure Exploratory testing is nearly useless if the basic features are not working.

P.S.: please do not mix **Exploratory testing** with **Monkey testing**. The first is about emphasizing the dominant thought process involved in unscripted testing, but second one is about random input to the tested system. Professional tester should not do monkey testing (there is an exception: randomization of automated tests, but generally this is not monkey testing)

In agile projects every sprint timeline we have:

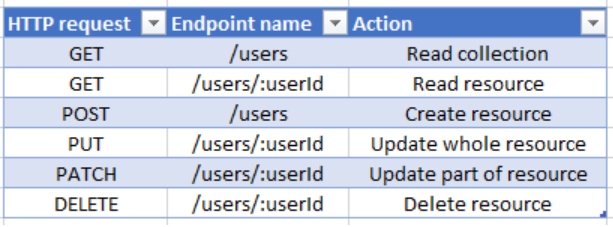
Test cases also get lighter compared to non-scrum testes: if you've been creating test cases at a detailed "click this link, enter that text, click this button" level, you can (thankfully!) stop doing that and look more towards a lighter style

Generally speaking, **test cases come from the user stories and acceptance tests**: as an example, **an acceptance test** might be *"As a logged in user, I expect to see a list of all my orders and their current status when I navigate to the order management page"*. This could produce test cases that look a bit like this:

* If the user is logged in, there is a link to the order management page on the main page.
* If the user is not logged in, there is no link to the order management page.
* The order management page lists all orders for the logged in user, sorted by date from the most recent to the oldest, and shows the status of each order.
* good unit tests will cover business logic and allow your manual testing to focus on end-to-end scenarios
* **Test the happy path/steel thread first**. This is the minimum necessary functionality to satisfy the user story/acceptance tests/use case.
* Exploratory testing sessions should be documented and time-boxed ("Spend an hour exploring the behavior of the order management page").
* In **a waterfall environment typically there is a testing phase that happens after development is done and is pushed towards the later stage of development**. This of course does not mean all companies should immediately switch to Agile as some teams/projects work better in that type of environment but is just some of the pros/cons between the two.

1. **Test Flow 4**

* In my current shop, the **API**s are written in **Java**. **Unit and integration tests** are in **Groovy** (so they can use the Java classes), while **end-to-end functional** tests are in **Python**. For most of our projects, **test code lives in the same repo as the application code**.
* For any Agile project, this is what I recommend for API testing.
* Keep your API test code along with your Dev code base
* Automate preferably using the same language that your developers use
* Automate using toolsets that integrate well with continuous testing pipeline
* **Integrate all the API tests with CI pipeline**
* Use stubs and service virtualization to test hard to find data
* The biggest benefit is that anytime there is a **fork / pull request** QA has to determine if a test needs to be added / modified. Once you have all of that in the pipeline, you will know the quality of the change as soon as the code is compiled.
* A generic framework should be independent of the API's behavior. There is another better approach to automate REST APIs which is using the **DataDriven approach**. Place **all your request building** and **response validation data,** that includes **endpoint**, **http-Method**, **request body,** **headers, query-params**, etc in an **excel sheet** to drive your test data. Write your Java code to access this data.



You could **create (or delete) a particular user directly in your database as a part of the test set up**. That way you avoid using the very calls you are trying to test. You can also delete the user as a part of tear down of the test.

So for testing GET in your case, create a user during set up, send the GET request and delete the user in the tear down. For CREATE, send POST and delete the user in the tear down. For DELETE, during set up insert the user directly into the database and make the DELETE request.

## Test flows

Let’s distinguish between **three kinds of test flows** which comprise our test plan:

**1.** **Testing requests in isolation**– Executing a single API request and checking the response accordingly. Such basic tests are the minimal building blocks we should start with, and there’s no reason to continue testing if these tests fail.

**2.** **Multi-step workflow with several requests –**Testing a series of requests which are common user actions, since some requests can rely on other ones. For example, we execute a POST request that creates a resource and returns an auto-generated identifier in its response. We then use this identifier to check if this resource is present in the list of elements received by a GET request. Then we use a PATCH endpoint to update new data, and we again invoke a GET request to validate the new data. Finally, we DELETE that resource and use GET again to verify it no longer exists.

**3.** **Combined API and web UI tests**– This is mostly relevant to manual testing, where we want to ensure data integrity and consistency between the UI and API.

We execute requests via the API and verify the actions through the web app UI and vice versa. The purpose of these integrity test flows is to ensure that although the resources are affected via different mechanisms the system still maintains expected integrity and consistent flow.

**--------------------------------------------------------------------------------------------------------------**

**--------------------------------------------------------------------------------------------------------------**

**--------------------------------------------------------------------------------------------------------------**

**API/UI testing Scenario**:

1. Scenario 1.

* Product is being developed by one team of 13 devs, 2 qa and 1 builid engineer/ dev/ops.
* We are planning to implement CI and CD also.
* On **pull request** components should be deployed to a test env and my tests executed there.
* Then if successful components will be deployed to staging env 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.

# [**Testing a Python REST API**](https://sqa.stackexchange.com/questions/13539/testing-a-python-rest-api)

* API is implemented in Python using Flask with Restful
* outputs JSON.
* **It is relatively small, has about 40 endpoints**. most of which have GET, POST, PUT and DELETE request methods.
* You can run tests against endpoints in the cloud, in a private staging environment (VPC, behind firewall, etc.) or even on your localhost.

1. [**API tests against production DB with no endpoints to set up test data**](https://sqa.stackexchange.com/questions/42193/api-tests-against-production-db-with-no-endpoints-to-set-up-test-data)

* I'm creating regression tests for one of our APIs. I have no problem on the **QA test environment**.
* When it gets **deployed to a staging env for final tests before going to production**.
* **Staging DB should be a replicate of production DB, BUT a different DB.**
* There should be at least Three databases**, one on the QA env, Staging DB (replica of production DB** and **production DB**.
* Some of the endpoints don't have an equivalent for deletion/change of a resource, so if I create a resource via a certain endpoint, I can't delete/change it using a different endpoint of the same API
* In my particular example, there is not an endpoint for creation of a resource that I need to use/test later
* The problem I'm trying to solve is as follows:
  + - There's endpoint /Cards/{ean} that creates a card. The card has to be created for a particular user, if not, I get **404**:
    - if (email == null) return NotFound(string.Format("The email is required when issuing a new card."));
    - The card (its ean) also has to exist, or I get 404, and has to be in a particular state, or I get **409** Conflict:
* All in all, the problem seems to me like there's no straightforward way to create test data.
* There're a few possible solutions I can see at the moment:
* I can **create a new card directly in the database** (insert into Card...) before testing the API, so then I could use the card number I created prior to my API tests.
* **Devs create a new endpoint** that'd be able to change card states (return them to the state before the tests), or create new cards. Obviously, I have no idea if devs would be very cheerful about this.
* We **start using a different database** where I can do whatever I want. I think this is quite nice, but requires some resources that might not be available for such a purpose.
* I **forget about the staging environment and test it only on dev**. But it seems reasonable to run some tests even on staging as well, so I don't like this very much.

1. [**API tests against production DB with no endpoints to set up test data**](https://sqa.stackexchange.com/questions/42193/api-tests-against-production-db-with-no-endpoints-to-set-up-test-data)
2. **Why are you automatically syncing staging data to Production?**

**The purpose of staging** is to mimic the production environment and make sure things work in production once the product is deployed.

1. **How to achieve it through postman**

I can create a new card directly in the database

CI/CD has many powerful tools, tools like Jenkins, Octopus, etc supports Powershell and batch commands, etc. You can **add a build step** for creating a table for the card in the database and then after postman scripts step finishes add a final step to remove or restore the DB to production.

1. **Risk with your approach**

I use some of the production data (cards),

That would be the worse thing you could do. Scripts won't always work as we expect, a small change in output might change the program flow and could result in unintended changes in production cards.

If you are going to replace the entire table with production DB then it would be safer. Do not try to individually manipulate or restore values that could affect production settings.

**Devs create a new endpoint that'd be able to change card states**

This is a good solution not only for now but also for the future. Such API endpoints avoid unintended DB changes by inexperienced persons. It can make sure no other table or fields are affected by providing proper access control.

**Staging database should be able to manipulate and rebuild as when needed.** It should mimic actual production one but should not automatically sync two-way.

I forget about the staging environment and test it only on dev.

This would be highly risky as you wouldn't be able to **predict how the product behaves in actual world**.

**Summary:**

1. Ask for more API endpoints to manipulate the database and thereby avoiding unnecessary DB changes
2. Make changes to the table in Staging DB, but don't sync production with staging. Restore changes and sync staging with production at the end of tests.
3. If you want to sync staging to production add separate build step, don't automatically do it. Trigger it only when needed
4. [API testing single layer](https://sqa.stackexchange.com/questions/32514/api-testing-single-layer)

* I would like to start writing some API tests for one of our APIs, but we are seeing a lot of issues with stability of other APIs and I'm not really sure how to handle that problem.
* To clarify I have an **API A that uses API B,C,D and so forth**. And currently I spend a lot of time just figuring out that most bugs are not because of A, but instead of outputs from the other APIs. Preferably I would like to stop doing that type of debugging, because it blocks my own workflow. i.e. I cannot go forward unless my tests are working, and they won't work until another team for the other APIs fixes their bug
* to expand on our API testing. We have a framework written in C# to be able to easily **send requests and store responses**. So, first we **manually check an API result** (request X should always return response Y) and we **store the JSONs** used in files. We then write a test method using these JSON files to generate the same request every time, and thus to **validate that the response is still exactly the same as when we saved it**.
* As a tester. You shouldn't do any debugging and investigate what is the root cause of the test failure (the only acceptable type of debugging is to reproduce the test flow manually to make sure there is no logical mistakes in your test code). This is the job of dev guys. Moreover **you're doing a black-box testing**. You should not care of what particular implementation underlies the "box" (API A) you're currently testing.
* You have API. **You should have API specification**. So having API specs, makes it pretty easy to write the tests having no working service at all. So just **write the tests like the service is working fine** and raise bugs once your tests fail in test environment.

**Things can’t be automated:**

* Ask Dev for thorough unit testing.
* Ask Dev to provide API
* **Break the work flows into smaller segments, that can be automated**.

Test Procedures

1. Company 1

When We receive a build of any web based or mobile based application this process we generally use.

1. **Smoke testing** of the build :- Which consist a **short positive flow** of application.
2. **Functional testing**:- Which consist of all **negative and positive testing** using test cases.
3. **Exploratory testing**:- This type of testing is done for **extra scenarios** which occurs to a tester during functional testing, it may be done out of scope of requirement to check how application is behaving, but before it a functional testing is needed to run all positive and negative test cases on build.
4. Company 2

We are building an application using a completely test-driven approach. As developers, we are very familiar with unit tests but haven't been exposed to integrated / functional / acceptance tests. Hence this post.

The application exposes a **web UI (HTML resources), which invokes a secured REST API (with JSON serialization)**, which then delegates to a core domain model (transactional application services and business entities) independent of any delivery mechanism. **The REST API is to be made public eventually**. Persistence is achieved using a relational database and an ORM. A standard **Java / Spring / Hibernate** technology stack is used.

If we wanted to do pure TDD, we would be writing the following tests:

* **Selenium / WebDriver tests for the web UI**
* integration tests for the REST API
* integration tests for the application service
* integration tests for the repository (persistence)
* unit tests for the web controllers (REST API)
* unit tests for the application service
* unit tests for the validator

By integration tests, I mean tests that target a fully functional application deployed on a production-like environment. By unit tests, I mean tests that target individual classes for which collaborators / dependencies have been replaced by test doubles (mocks, stubs, whatever).

Clearly, every test has a different perspective (user-centric for web tests, API consumer-centric for API tests, developer-centric for others) and **validates different assertions** (**HTML element contents for web tests**, **HTTP responses and status codes for API tests**, **database contents and exceptions for application tests**, edge cases for unit tests, etc.). But there's also a lot of overlap between all these tests, particularly when it comes to fixtures (e.g. user cannot register if other user with same email is already registered). As such, we could avoid writing some of these tests by focusing on the higher layers (e.g. **only write the web test for the user registration with duplicate email case**), but the lower layers would then be exposed if they were to be reused by a different client (e.g. application services used by a batch client).

**Regression Test**:

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.

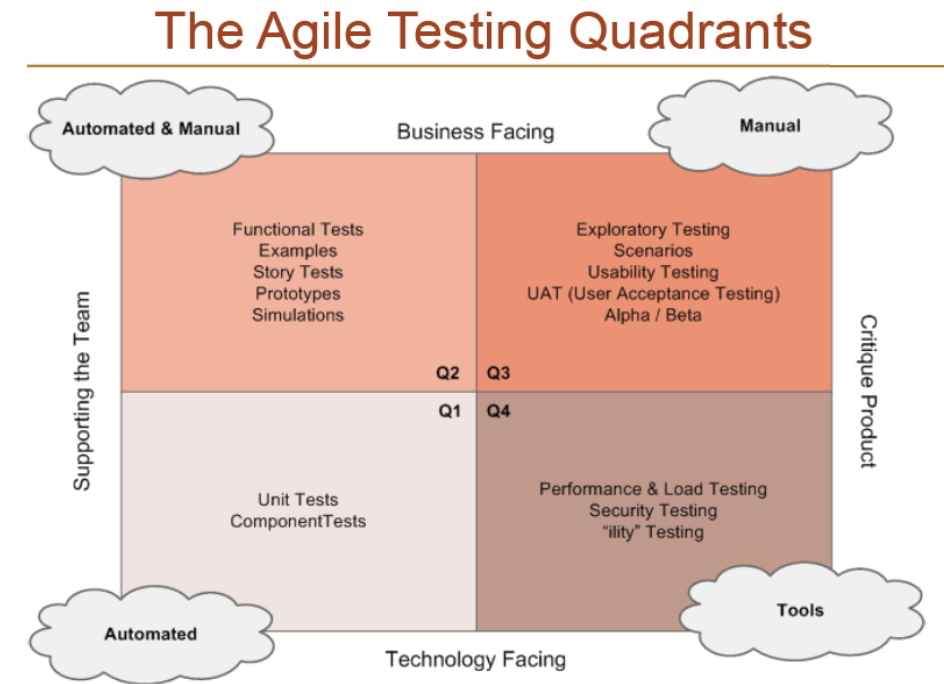
By **focusing automation on the combination of steel-thread and known breakages**, you're likely to be effectively regressing the most used and most fragile areas all through development, and can use any "found" time to add automation for any other areas you consider potentially risky.

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.**

**Scrum/Sprint**

* **DOD: Definition of Done**
* **User Story**
* **Story point**
* In most agile approaches, there is an assumption (derived from lean) that you want to **build quality in rather than checking for it later**. **The definition of done** established a standard intended to promote a particular level of quality of any product and the team is expected to apply whatever skills necessary to build an increment of the product that reaches that needed level of quality. You gather the teams together (or trusted representatives of the teams) to **define the minimum definition of done** for the whole product that stretch across teams (though individual teams are always welcome to be more stringent).
* I would take the proactive approach and write the **Acceptance Criteria** and then **have it reviewed with the Product Owner** (or whoever wrote the User Story) and Devs. I recommend writing it in the[**Gherkin Given-When-Then**](https://www.agilealliance.org/glossary/gwt/) format. The example given in the agile alliance website is like this:
* **Given** my bank account is in credit, and I made no withdrawals recently,
* **When** I attempt to withdraw an amount less than my card's limit,
* **Then** the withdrawal should complete without errors or warnings

****

* I focus on my developers doing a good job in Q1
* I work to write good automated UI tests in selenium and capybara for Q2
* Exploratory testing is essential and is represented in Q3
* this is where I cover those 'hard to cover',.
* 'not worth automating', 'visual',
* 'judgement involved',
* 'domain knowledge required' forms of exploratory testing.
* Performance and load in Q4

1. **Scrum meetings**

* As a tester and experienced working on Agile, **We had daily scrum meetings** to discuss our targets to be met for the respective sprint.
* The targets as a whole impacted minor targets in test strategy and items to be tested immediately.
* Test Automation was used for repetitive test cases.

1. **Scrum Master** (more or less like a **project manager or program manager**)

* Classically, the SM role is to *facilitate* progress by finding ways to clear or work around any impediments that arise during a sprint, to lead meetings, to ensure that meetings stay on-topic and stay within the time frame, to assist with coordinating resources, and so forth. SM is basically there to assure the project is on target and on schedule. Ideally, the SM role should *not* be held by someone who has another role in that project.
* If I've interpreted your question correctly, your SM is also a dev on the project team: this is, as you've experienced, not ideal. If this is the case, I'd suggest your first priority would be to try to arrange future projects so that each team member has exactly one role - that is, your SM is not a developer (or QA) on that team.
* Regardless, I think you should try to have someone take Scrum Master training and become **a dedicated SM** for as many project teams as possible - it's very much an administrative/resource-juggling role compared to the Dev/QA roles which can and should be blurring in the teams

1. 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 very sprint**. IMHO this is the wrong answer because you are starting to fallback on the older waterfall habits of 'testing in the quality at the end' instead of making sure that as each item is done you are truly finished with it and don't have to think about it again.
2. Automation should itself be a task in Kanban/Scrum. It's possible to get to the end of a sprint and not have all tasks on the sprint done but supposedly, while you're running the sprint, you prioritize things so that stuff for any one user story is not half done. **That allows you to ship SOME. Anything left over is backlogged to the next sprint, that includes automation**.
3. "Scrum is the best agile approach. In scrum, **each sprint produces an increment, which is a potentially releasable product**. Here, **each increment must satisfy all acceptance criteria and pass the different categories of tests**. Regression tests are a tiresome activity, especially in an agile process, which is characterized by **nonstop changes and frequent deliveries**.
4. **It's all part of being ready to ship at the end of any sprint.** scrum means everything should be done is done in the end of the sprint
5. Then get your CI system setup so that the **regression tests** run on a **regular basis (I'd recommend nightly)**
6. **Master Regression** - would be a big set of test folders with merged test cases under each functonality and the new functionality or changed functionality will update existing test cases
7. ITERATION Regression - After every iteration ends, we are doing a manual regression of the stories/test cases in the next iteration. We test all the functionality of the previous iterations. In the upcoming iterations we do the iteration + 1 testing.
8. **Example of a sprint flows**

* We use to do gather as a team for Sprint Planning meeting(Where we decide, what can be delivered and which user stories to be delivered first/last in a sprint)Our sprint is **for 2 weeks duration**.
* Scrum master will be ready in Sprint planning with any planned leaves from key resources and any Public holidays coming in a sprint etc and will decide the Total Team Effort.
* Including tester's, BA, Dev's everyone will gather and it's **Scrum Master** turn to **explain each user story with Acceptance Criteria of completing it** on Digital screen.
* Then Dev's, Tester's will discuss on the **flow** about the effort involved or any dependencies etc.
* We play [**"Planning Poker"**](https://www.mountaingoatsoftware.com/agile/planning-poker) game to give **individual estimates**, which are called story points. We decided that 1 point = 1/2 day work every company has their own definition of a story point.
* Finally, after noting down every individual story points **Scrum master** will come up with **Actual Story point** (which is the most common Estimate given by the team members)
* Let's say PersonA thinks it's gonna take 2 points PersonB- 3 Points PersonC- 3 points PersonD- 1 point PersonE- 5 points
* Then the most common estimate we got from team is 3 points so this will become the Estimate for the Story.

1. **Sprint flows**

* **Project Initiation:** Get an understanding of the project(business)
* **Release/Project Planning:** Participate in sizing stories && Create Test Plan
* **Each Iteration:** Estimate tasks, Run regression tests, Collaborate with customers on acceptance tests, Write/automate/ execute new story tests, pair with test with other testers and developers, perform exploratory testing.
* **System Test / End Game:** Perform Load Test, Compete Regression Testing, Perform UAT, Perform Mock Deploy, Participate in release readiness.
* **Release to Prod/ Support:** participate in release to prod. participate in retrospectives.

1. Sprint flow ideas

* When the Scrum master creates sprints, 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.

1. Sprint flows\_2

* SCRUM  
  Good Automation coverage  
  Follow Sprint model.  
  Pair Programming  
  Mandatory Code Review by peers.  
  Well defined Acceptance Criteria  
  CI / CD Model to take care of continuous integration / deployment  
  **One push deployment**

1. Sprint flow 3

* We implemented the code freeze only for one specific purpose - to **deliver clearer code to stage env.**
* where
* All the stories that have been made during this sprint are still working and not broken **on the last or any another day of a sprint**
* Critical functionality is working properly(smoke testing)
* **In Staging we conduct again smoke testing and then if ok - regression testing (full) after pushing to production, basing on risk analysis we conduct partial regression testing.**
* **So the devs can't push any new commits to dev only about 2-3 hours in the end of a sprint. With automating all of these it will take for 20-30 minutes.**

1. Sprint flow optimization

* **Product Backlog Items** (PBI)
* If you cannot finish a PBI or user stories in one sprint, **it is too big and needs to be split up into smaller parts**.
* Finishing a PBI means it has to be **shippable** and conform to your DoD, which normally means it has to be tested.
* There is **absolutely no reason why a team couldn't build and test in the same sprint**. If you have a good DoR, **a tester can start preparing tests as the code is being built,** and executing those tests can be done in a matter of minutes - with plenty of time left to fix any found defects.
* **Regression tests, load tests, integration tests are maintained in the same sprint as where an increment is delivered**.
* **If you cannot finish coding on time to allow for testing within the sprint, your PBI is simply too big and the team should consider thinking about smaller increments**.
* A separate testing team should not be able to find actual defects: why did your team deliver something that did not conform to specifications - that seems to be the most essential part of your DoD?
* **If *new* specifications pop up during post-sprint testing**, they are *new* wishes and can be handled as any other change request. If *actual* deviations from original specs is found, your PBI should not have been considered finished. In order to check if your team's work matches the requirements, you need to... test! Which means that even with a separate test team, you still need testers in your own scrum team.

# [**Area of responsibility of a test manager within the agile process?**](https://sqa.stackexchange.com/questions/40145/area-of-responsibility-of-a-test-manager-within-the-agile-process)

1. User Story testing

* User Stories provide good insight into how a product should behave. While they seldom contain explicit testing criteria, there is often much implied that can be extracted from a user story which the test can develop into more formalized testing.
* I like to break each user story down in three different passes:
* First pass**, identify functionality and features**
* Second pass, identify any limits imposed by a given environment, operating conditions, etc. This can be things like performance/throughput, duration, **security**.
* Third pass, identify:
* Research [**Exploratory testing**](https://sqa.stackexchange.com/questions/30079/is-there-any-standard-or-pattern-of-doing-exploratory-testing/30081#30081) and figure out ways to come up with new test ideas.
* Interfaces (other software, files, databases, peripherals, etc.)
* Inputs and outputs (records, messages, files, reports, objects, etc.)
* Configurations (System level, application level, hardware, network, etc.)
* Usage scenarios (normal "happy path", error handling/recovery, backup, installation,etc.)

**Create test in Sprint**

1. In agile projects every sprint timeline we have:

* **new functionality** to be tested (the size of this part is nearly the same every sprint)
* **regression & verification** of bug fixes (and this part becomes bigger from sprint to sprint)

### New functionality

No automation, More informal scripted tests (test cases), Less Exploratory testing

Commonly testing in agile is started when new code arrives, but in this case testing loses precious time. Ideally tester should participate in scrum meetings to have full understanding of user stories, so he could represent end user interests. This early phase gives you sketches of test cases.

Then as software arrives you check it with this informal test cases (extending them according to your new knowledge of the software). This is the first phase of testing, **scripted manual testing** (there should not be too much exploration) to check basic functions of software. This is how you test **new functionality**.

### Regression & verification

More automated tests, Less scripted tests (test cases), and Exploratory testing as a separate part of testing process

As soon as core functionality becomes stable (business process or part of it could be accomplished) you should start with automation. **Core business functionality should be validated with automated tests.** This will save you time on **regression & verification** to do exploratory testing without scripted test cases but with great mind efforts. Be sure Exploratory testing is nearly useless if the basic features are not working.

P.S.: please do not mix **Exploratory testing** with **Monkey testing**. The first is about emphasizing the dominant thought process involved in unscripted testing, but second one is about random input to the tested system. Professional tester should not do monkey testing (there is an exception: randomization of automated tests, but generally this is not monkey testing)

In agile projects every sprint timeline we have:

Test cases also get lighter compared to non-scrum testes: ***if you've been creating test cases at a detailed "click this link, enter that text, click this button" level, you can (thankfully!) stop doing that and look more towards a lighter style***

Generally speaking, **test cases come from the user stories and acceptance tests**: as an example, **an acceptance test** might be *"As a logged in user, I expect to see a list of all my orders and their current status when I navigate to the order management page"*. This could produce test cases that look a bit like this:

* If the user is logged in, there is a link to the order management page on the main page.
* If the user is not logged in, there is no link to the order management page.
* The order management page lists all orders for the logged in user, sorted by date from the most recent to the oldest, and shows the status of each order.
* good unit tests will cover business logic and allow your manual testing to focus on end-to-end scenarios
* **Test the happy path/steel thread first**. This is the minimum necessary functionality to satisfy the user story/acceptance tests/use case.
* Exploratory testing sessions should be documented and time-boxed ("Spend an hour exploring the behavior of the order management page").
* In **a waterfall environment typically there is a testing phase that happens after development is done and is pushed towards the later stage of development**. This of course does not mean all companies should immediately switch to Agile as some teams/projects work better in that type of environment but is just some of the pros/cons between the two.
* Request the team to adopt a **services-based architecture**(SOA/Microbreweries). **Testing APIs are easier than testing UI**, especially on creating Automated tests. Automated API tests can be executed really fast and will give quick feedback.
* UI Automation- **Automate only stable and high ROI tests**. We should be really wise while choosing the right candidates for UI automation or else we will end up spending more time in maintaining the UI automation scripts than testing it

**QA routine work while waiting for new Build in Sprint**

* **Build/run load tests** for new/recent work (given the maturity of your test automation you may already have this under control)
* **Review existing test automation for obsolete or ineffective tests** (You have no idea how much I wish I could reach this point)
* **Review and refactor existing test automation code.** In any rapid development environment, automated test code can get dated quite quickly.
* **Review and update older customer documentation**. In my experience this can become out of date fairly rapidly if development is quick.
* Review other documentation to make sure it's up to date. This can include (but is not limited to) use cases, business requirements, database dictionaries, functional requirements, test documentation...
* Work with product owners to refine any stories in the backlog - or just go in there and review them and ask questions anyway. Testers typically have a unique combination of breadth and depth with a product they're familiar with and can often pick up potentially problematic changes before they go to code.
* **Review the user stories and defects in the current sprint and start planning how to test them**. If there's configuration that's needed, it can save a lot of time to set up as much of the configuration as possible before the story/defect is coded.
* Investigate results of [your monitoring and logging instrumentation](https://assertible.com/blog/testing-and-monitoring-in-production-your-qa-is-incomplete-without-it);
* **Extend your monitoring and logging instrumentation**;

**UI test Practice**

1. The best way to test the UI is to make sure that you have UI tests that

* Can run continuously in branches (post push to remote) using a CI system
* Use **good element identifiers** that are robust and unique and not brittle
* Use **Page Objects** to extract and name element identifiers
* Use a custom DSL as appropriate to extract common functionality into named routines
* Differentiate and tag test types such as smoke, happy, sad and optional

Are easy available and often in the same code repository as application code

Do not perform data combinatoric testing that should be done through unit and integrated tests

Do not duplicate unit and integrated tests

Do not otherwise perform the task of missing unit and integrated tests

1. **Test constantly and don't wait**. Test early and often. Shift Testing left. These are good guidelines to go by but there are subtle parts to the implementation of these practices.
2. The difference between this and **testing a back end service** is primary two things
   * speed. browsers tests take minutes and hours. unit/integrated take milliseconds and seconds
   * control. browsers are foreign asynchronous devices over a network that can drop packets.
3. [**Characterization Testing**](https://en.wikipedia.org/wiki/Characterization_test) (aka Golden Master Testing) is a means to characterize the behavior of the test object to protect it against unintended changes, regardless of its correctness. In order to do so, the results of a previous (typically stable) version—the golden master—serve as the oracle for the tests (also known as [consistency oracle](http://www.softwarequalitymethods.com/Papers/OracleTax.pdf) since it compares the consistency between two versions).
4. Many systems are well designed so that the **GUI is just some glue between the user and the underlying APIs**, which is why such systems often don't need GUI-based E2E tests at all.

However, **the need of GUI-based E2E tests is something you can't decide per se**—if you need them, you need them. The thing is that nowadays you can quite easily scale out these tests. For instance, ["UI Testing at Scale with AWS Lambda"](https://aws.amazon.com/de/blogs/devops/ui-testing-at-scale-with-aws-lambda/) shows how to leverage Function as a service (FaaS) to run every test in parallel. This way, **your entire GUI tests may run in less than a minute**.

Therefore, I would say you can **have fast GUI-based E2E tests within your CI pipeline**, but it is not always feasible. Analyze the given project, calculate the risks, determine the trade-offs, decide.

1. We have tests which take up to five minutes to **just prepare all the test data** using UI (and those data are are time-sensitive so they cannot be reused, and complex enough so they cannot be just injected to DB

# [**Creating selenium scripts from requirements**](https://sqa.stackexchange.com/questions/45426/creating-selenium-scripts-from-requirements)

* Is it possible to create selenium web-driver scripts just looking at test cases, use cases or requirements. I always wonder **how the scripts could be created in advance when the functionality has not been implemented in the product**. How could you get the locators and all the details of the application flow to add in your scripts? In a fast paced environment where testers are mostly writing test cases before development, how an automation can be done in that phase by just looking at the requirements?
* Yes, it is possible. One way I've done these in the past is having the requirements and the ***UI mockup of a page or feature.*** If you can align those two together, then you can start **creating PageObject classes** for that page/feature. You create the constructor, the element variables (**without the specific locator**), and any methods that use those variables. You can also create the test cases the implement the PageObject and even start adding your assert statements.
* Then, once development is finished, all you need to do is find the **right locator** (css selector, xpath, id, class, etc) and plug those into your element variables in the PageObject.
* Will it always work exactly as written? No, but you just gave yourself a great head start and now you just need to modify and adjust a few things to get it working.
* **The key to getting this going is having the UI mockups**. Those give you enough of the application flow, the element types (buttons, links, forms, etc) to work with, which you can plan for in automation tests. If you're just going of the requirements or use cases without any UI, then it's much more difficult to get this head start.
* I've also been asked this in interviews before and gave this answer, which always got a good reply
* In another scenario, testers are delivered with the implemented functionalities in the environment where developers keep making changes by adding or improving functionalities, the test script fails at some point due to those changes.
* In this scenario, while you can doing some of the things described above to get a head start, you can also consider waiting for application/feature stability before completing any Selenium work, especially if **there is potential for element locators to change**. If the UI or the application logic drastically changes, then no head start will help.
* I've had great success over the years of doing it this way regardless of a new feature/page.

# [**Selenium Test Design Struture - Test Granularity for a Large Project**](https://sqa.stackexchange.com/questions/20763/selenium-test-design-struture-test-granularity-for-a-large-project)

# **Setup**

Currently my organization has great number of what I would describe **as "Functional Tests" written in an excel document**, with the following structure.

* Press button X -> Grid enters edit mode
* Attempt to add invalid value Y to field X in edit grid -> validation is shown...
* Delete item X by pressing... -> item is removed from the grid

These tests have been working well for our organization, but as our product increases in size we have decided to use Selenium to automate these tests. Following **the**[**Page Object Model**](http://www.seleniumhq.org/docs/06_test_design_considerations.jsp#page-object-design-pattern) and Selenium [Best Practices](https://mestachs.wordpress.com/2012/08/13/selenium-best-practices/) I have converted these tests into **several hundred(soon 1000's) of Selenium automated tests**.

when translated into automated selenium tests it soon becomes readily apparent that they have a large number of **dependencies** on each other. **if one dependency is broken, many later tests will fail for reasons not readily apparent to the test runner**.

# **Current Design**

What I've done to solve this issue of dependencies and run order execution is to structure my tests in the following manner:

* ClassInitialize(Each test class represents a page) - **Create a Bunch of Test Object/Dat**a to be Manipulated
* Test 1 - Access already create Test Object, navigate to subitem, create/edit/delete
* Test 2 - Access already create Test Object, test validation
* Test 3 - Access already create Test Object, create/delete

The tests can now be run in any order, they no longer have any dependencies on each other (only the data/objects created in the "ClassInitialize" class).

However, now I no longer have the granularity that I achieved with my hand written tests, as now, any time I interact with items that I didn't create in the setup step, I need to **create and delete them just to test the edit functionality**. Also the "Class-Initialize" class becomes huge and encompasses much of the functionality that I'm trying to test in the first place.

As your description, your test suite has already provided a good level of granularity, but when converting them to automated tests they get a number of **dependencies** (**prepare data, object** .. etc). My suggestion is that you can **do the dependent steps directly in DB**.

About the granularity, we just convert test case to automated test, so it will be the same level of granularity as what your test suite has provided.

**Original** Do you or your team have permission to access database ? If yes, you can **prepare pre-condition data directly in database**

* Write some store procedures to setup pre-condition step.
* Run these setup on Test account.
* Create a job to clean everything on Test account each day.

**JS thought**:

Lets say the flow is : **log-in** , **create a test-flow**, (**verify the flow exists), edit the flow** and verify the changes**; remove the flow** and verify it.

**How to prepare for data**:

1. Create the user credentials so that log in should work.
2. Create a test flow through API or manually or DB under the known user login;

Test “Verify the flow” function

1. Since we know “Verify the flow “ function works; We can run Test “Create the flow “, and then RUN test “Verify the flow”
2. “remove test “ should be run on Pre- Created flow through API or GUI or DB; Not from the flow created during test.

**Avoid running one test with more than one dependency or test conditions**

Scenario 1: **Posting Test**

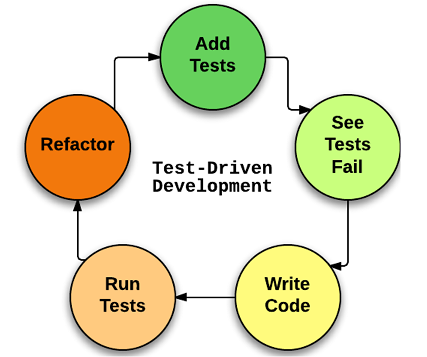
1. Login
2. Create a post
3. Verify that the post is visible

This one has one test “Create a post” + “verification test”, if unsuccessful, we don’t know if “Create .. “ fails or “the verification “ fails. Solution is **making sure “verifying function**” works first.

Scenario 2: Removal test

1. Login
2. Create a post
3. Remove the post
4. Verify that the post is gone

The goal is to verify the “Removing functionality”, but if “The create” fails, then We can’t verify the “Remove part”. So we have to add one more step, which is “Verify the create first” or run the “create test” first.



It sounds like you could use [**Cucumber**](https://cucumber.io/)**;** a tool used for Behavior Driven Development. The process goes something like...

1. **Review Acceptance Criteria** in the ticket
2. Write down each **Scenario** in your .Feature ([Gherkin](https://cucumber.io/docs/gherkin/reference/)) file
   * This should be done by the Developer before they write a single line of code, so as to better understand their task before completing it
   * Reviewed by BA and QA as it clear text in git or can be copy+pasted into the ticket
3. Dev writes code for the new functionality
4. Now the **Scenarios** can be wired up to test methods via [Step Definitions](https://cucumber.io/docs/cucumber/step-definitions/)
5. Tests run and pass before the PR (new feature code + tests) is submitted

**Test reports**

1. **A bug report** containing the below could be meaningful:  
   Number of test cases passed, failed and pass percentage
2. **Analysis and classification of failures** : This is very important. A brief failure classification helps understand whether it is an automation issue, application code bug or environment issue. If a large % of test cases fail due to automation/environment issue it alerts us of the bad health of the automation suite
3. As @Tristaan pointed out having the list of bugs with the developers in loop can help to keep the DEV informed about the priority and timeline of the bug
4. **Number of TCs blocked by a bug** - This again can help the developer understand how critical the bug is. If a bug blocks 300 test cases, no matter even if its a trivial issue, it needs to be fixed ASAP. You can add the DEV's manager as well in these situations

Maintaining history of test results is also important. If I want to find **the delta of test cases that passed in the past release and failed in the current one**, the history helps. I usually organize test results release-wise in a shared folder.

If you're doing standups, then you can report anything the dev group needs to hear about there (or at least tell the relevant developers that you need their time).

1. **Use spreadsheet** to keep track of test progress

Your **test plan document** gives the name of the test case spreadsheet and a link to the file location. So in the Project X test plan, there's a line that reads something like "Test cases are documented in ProjectXTestCases.xlsx" and you've created an anchor to the network location of ProjectXTestCases.xlsx

1. I'd look at using spreadsheets to manage the test execution, working something like this:
2. The **master list of all test cases** for the project is either a Word document or an Excel spreadsheet, **linked** from the **Test Plan**.
3. **Each test run** can be either a separate spreadsheet or a worksheet within a single spreadsheet, depending on how complex the system is. For a highly complex system, I'd use one spreadsheet with worksheets set up for each logical grouping of test cases (e.g. configuration test cases would be one worksheet)
4. The first worksheet on any test run would contain a **summary of test results**, including a list of failed test cases (create these as internal links to the actual test case and result by doing a copy - paste as link).
5. The summary would also include **pass/fail percentage** possibly itemized by test case priority.
6. If a defect is severe enough to block further testing, this would also be noted on the summary
7. If you have no other means to **track defects**, I'd use a **separate spreadsheet and link to it so that you can number and link any defects you find**.
8. Your spreadsheets would live in a network share where every member of the team had access to them. For multiple testers, you might consider sharing them.
9. 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.
10. 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.
11. **Use spreadsheet-2**

unless you have a clear demand on automated processing I would tend to storing results in Google Sheets. Here are few points for the choice:

1. You can easily manage the data from wherever you are since it is world-wide accessible service (and has the client applications implemented for all the famous platforms)
2. Sheets provide some data analytics tool-set
3. You can easily control the access of other people to your data
4. Google Sheets support programmatic usage [via API](https://developers.google.com/sheets/api/)

**Automation Framework**

**Release schema** of **Release.MajorVersion.MinorVersion.BuildNumber**, automation versioning needs to happen at the Release.MajorVersion level.

* **Build libraries.**  Basically, each library follows old Unix philosophy "**Do one thing and do it well**", so they are useful and easier to maintain.
* Introduce **one coding style** so it is easy for new comers to read existing code.
* Introduce **cross-team code reviews** from time to time, so people familiarize with code of other people, build and enforce coding standards.

1. **Test flows**

In my organization mostly the developers write the automated tests. Automation test UI /API and **regression tests live with the application codebase**. Only **one master branch**.

* App Code and test automation code resides on GitHub.
* The developer works on an application feature, make changes to the automation script, checks it in GitHub, **create a PR** and after review merges the code with the master.
* **Jenkins build** kicks-off, **runs UI**, integration, and UI tests and **deploy the code in "Dev"**

If I understand the situation correctly, then either (1) you're promoting the test code into master before the production code, or (2) your regression test suite includes tests that are not yet in master.

* Test code and production code are always promoted together.
* **Regression test process to run tests only from the master branch**.
* Putting tests in a separate repository will **increase the cost of keeping the test code and production code compatible**.

1. **Test Flow-2**

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.

Now, the question is**, what do we test**? Do we spend 5 days running the full bunch of tests on V1.1 or do we run only specific tests for the

affected code and pray/hope the components do not break?

Assuming that you **can set up your tests to run in parallel**:

* 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.
* **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).
* Smoke or quick test part covers (ie samples) all functionality including in 12 hours duration. if this test-suite has even single failure, consider it is showstopper. Start next testing with this build only once fix is available and verified in Smoke test.
* Any failure in remaining testing (of 5 days) should be treated as low priority bug and it should be part of planned bug fix in future release.
* Now The question is **what should be done when smoke test is failed and new build is awaited.**
* We should start testing new build, with the same QA test routine.
* **Verifying new features and bug fixes first**.
* Should we continue with reaming testing? this is interesting as suppose bug is reported from the subsequent testing 1. It is possible that bugs reported from this testing could have same cause due to which smoke test has failed. Effectively wasting testing /analysis/development team effort. 2. It is possible that this bug is unrelated to bug reported in smoke. Effectively the idle period is utilized in finding "real" bug.

1. **Test Flow 3**

## **The test code is running against the application code**

So you develop a test suite in the same general code base as the application because of the following factors:

* When the application is changed it may (even should) break tests and you want to know about that when changing the application.
* **When new functionality is added you need automated tests for it**
* **When functionality is removed you need to update the automated tests.**

Basically doing these things separately is a hallmark of a waterfall process and that may be what you are expecting. Your colleagues on the other hand may be working on an Agile process and thus see application and test code being written hand-in-hand.

* Yes, store the test-code as close to the application code as you can.

If you **develop with feature branches** I would also build (or extend) the end2end tests in the same branch. The feature is not complete if it does not contain the end2end tests.

* **Tests are versioned with the feature**, making it easier to support multiple versions.
* **Feature branches can run all tests and can be released separately from other branches (e.g. merged to master after all tests pass)**
* No merge conflicts
* Forces developers and testers to work together on a feature.
* It minimizes handovers
* Developers can maintain/fix tests with ease during feature development

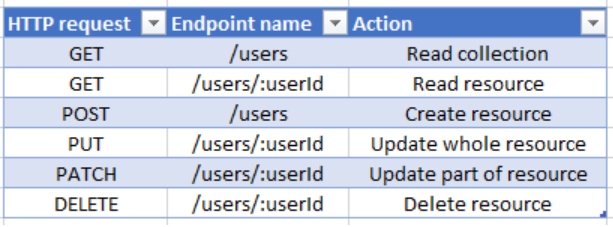
I can only come up with pro's for using the same branch. **Get rid of your automation branch.**

Different environments should be handled by configuration files. The version deployed to an environment will also contain all the tests so no problems here.

* Very Simple and straight approach where the testing team uses the same code repo on which dev is working, And we add the API test cases just like the unit test case in Test package of the same repo. **PROS:** less work because the testing team need not create DTOs etc., test cases will be available to the dev team, **CONS:** every time we **add new test cases**, we will have to do a **pull request to merge the test cases**, test cases will require immediate attention in case of dev code change otherwise there would be compilation failure.

1. **Test Flow 4**

* In my current shop, the **API**s are written in **Java**. **Unit and integration tests** are in **Groovy** (so they can use the Java classes), while **end-to-end functional** tests are in **Python**. For most of our projects, **test code lives in the same repo as the application code**.
* For any Agile project, this is what I recommend for API testing.
* Keep your API test code along with your Dev code base
* Automate preferably using the same language that your developers use
* Automate using toolsets that integrate well with continuous testing pipeline
* **Integrate all the API tests with CI pipeline**
* Use stubs and service virtualization to test hard to find data
* The biggest benefit is that anytime there is a **fork / pull request** QA has to determine if a test needs to be added / modified. Once you have all of that in the pipeline, you will know the quality of the change as soon as the code is compiled.
* A generic framework should be independent of the API's behavior. There is another better approach to automate REST APIs which is using the **DataDriven approach**. Place **all your request building** and **response validation data,** that includes **endpoint**, **http-Method**, **request body,** **headers, query-params**, etc in an **excel sheet** to drive your test data. Write your Java code to access this data.



1. **Test-Flow 6**

# [**Testing multiple API contracts as part of 1 test**](https://sqa.stackexchange.com/questions/35721/testing-multiple-api-contracts-as-part-of-1-test)

I have a situation whereby, say, I have one API which creates an employee(POST); one which then returns the employee(GET) and one which deletes the employee(DELETE).

You could use BDD #Cucumber here. **Given, When, Then** keywords are its keywords**. In Given step** method trigger API POST to create the user. **In When step** method, trigger GET to retrieve user details. **In Then step** method, delete the user. This way your scenario as well as your testing looks complete.

You could **create (or delete) a particular user directly in your database as a part of the test set up**. That way you avoid using the very calls you are trying to test. You can also delete the user as a part of tear down of the test.

So for testing GET in your case, create a user during set up, send the GET request and delete the user in the tear down. For CREATE, send POST and delete the user in the tear down. For DELETE, during set up insert the user directly into the database and make the DELETE request.

**Building flows**

All the calls in a collection get executed sequentially. This offers us the opportunity to test whole flows instead of just single calls. One such a flow for a /posts resource is:

* Get a list of all posts
* Fetch the first post in the list
* Update the post

## Test flows

Let’s distinguish between **three kinds of test flows** which comprise our test plan:

**1.** **Testing requests in isolation**– Executing a single API request and checking the response accordingly. Such basic tests are the minimal building blocks we should start with, and there’s no reason to continue testing if these tests fail.

**2.** **Multi-step workflow with several requests –**Testing a series of requests which are common user actions, since some requests can rely on other ones. For example, we execute a POST request that creates a resource and returns an auto-generated identifier in its response. We then use this identifier to check if this resource is present in the list of elements received by a GET request. Then we use a PATCH endpoint to update new data, and we again invoke a GET request to validate the new data. Finally, we DELETE that resource and use GET again to verify it no longer exists.

**3.** **Combined API and web UI tests**– This is mostly relevant to manual testing, where we want to ensure data integrity and consistency between the UI and API.

We execute requests via the API and verify the actions through the web app UI and vice versa. The purpose of these integrity test flows is to ensure that although the resources are affected via different mechanisms the system still maintains expected integrity and consistent flow.

END-To-End vs Contract-based testing.

**Availability**

Checking whether a particular API function (or a few of them) is down is the most basic question. That can be answered with a single API call.

**Functionality**

Checking whether a sequence of API calls intended to work in a scenario is unavailable or behaving incorrectly in some way would be considered shallow but broad coverage.

One example is continuously scrolling a map while the app visualizes points of interest around your point of view. This operation is simulated by **a sequence of API calls** feeding a geolocation.

Another example would be sending a search request in an online store, reviewing search results, and retrieving details of a found item. This operation is simulated by a sequence of API calls where **data retrieved from the previous call** might be used as parameters for the next one (for example, “item id”).

Broad coverage also involves checking whether typical calls, such as valid data or a generic response, yield an invalid response or is incorrect in some way.

Before looking at how well the API performs, the tests need to determine if the API functions at all, and if it works as intended. This starts with checking basic functionally like creating and deleting data via API calls as appropriate. The tests also look for missing functionality bugs where the API calls for a feature that isn’t there.

Once the functionality is there, the API test criteria are used to determine if **arguments and actions are working correctly**. For cases that aren’t working correctly, the testing focuses on **handling errors and warnings accuracy**. The tests also examine if the API is handling errors in a way that avoids unnecessary crashes. Finally the API tests examine if response data is correctly handled and formatted to specifications. This step checks if the [REST](https://www.apicasystem.com/blog/stress-testing-restful-apis/?__hstc=215227080.725166821da610bbcf2e67b1c214d9fd.1593557970526.1593557970526.1593564304102.2&__hssc=215227080.2.1593564304102&__hsfp=1112991701) or SOAP requests are valid, and if the returned JSON or XML data is properly formatted.

**Input parameters**

Input is what is sent into an API function when the call is made. Testing from the input perspective requires covering all variations of data input, including all equivalence classes of valid input and invalid input.

For example, boundary testing for a numeric field would include testing at the min boundary, one lower than min, way lower than min, one higher than min, one lower than max, at the max boundary, one higher than max, and as much as possible higher than max.

You also would include data variations to test valid and invalid format, precision, encoding, and any other applicable business rules. For example, for a web service “GetUserByID” with **“ID” as a parameter**, a thorough testing would require a sequence of calls like the following:

* Provide a legitimate ID
* Provide no ID (null input)
* Provide a non-numeric ID
* Provide a valid but nonexistant ID
* Provide a legitimate ID with the lowest number available
* Provide a legitimate ID with the highest number available
* Provide an ID with heading or trailing spaces

**Output parameters**

* Output is data that were returned from the API call. Data may come in different forms, like response code, response header, and response body returned by a web service.
* Testing the output aspects requires reproducing all variations of output. To provoke a desired output, you need to know what to request. A peek into a database would be of great help, as you can find and inject peculiar field values there. For web services, you have **to reproduce all intended HTTP response codes**.
* Whenever a data set is returned, reproduce cases like none, one, and max. For example, consider an online store returning a list of matching items. There could be none, or just one, or the returned data set may exceed the maximum number that can be displayed on one page—or the returned data set may be so big that it will slow down or crash the front-end.

**Flow testing**

* A special testing aspect is simulating **continuous usage of the API**. Imagine users constantly searching for merchandise in that online store. The intent of such a simulation is not a verification of something particular, but rather observing if the software responds consistently over time or begins to manifest different errors.
* Flow testing also includes simulation of **timed-out and interrupted calls**. If your system has functionalities for rejection and rollback of transactions, make sure to reproduce them, and verify what happens with data records. For example, let’s say you’re booking a movie ticket online. Typically, it’s a sequence of screens for movie selection, date and time selection, seat selection, and payment. As soon as seats are selected, they become reserved. But what if a payment was never made or didn’t come through? How many seats will be lost due to a fake or failed booking? To manage that problem, **the booking sequence has a timeout**. If the final call doesn’t succeed, all reservations should be released, and the customer should not be billed. While testing through the API, make sure to reproduce situations like this and investigate how the system handles a timeout and performs a rollback.
* Another aspect is testing with a **high volume of data** in input and output. That might be a request for an unusually large data record or set of records, or continuous throughput of data. This is similar to testing the system state, but the focus is on stressing amounts of data that the system should be able to handle.

**Structure Your API Testing Needs**

* The lists of test design ideas above might look frighteningly long, but that’s because we reviewed very broad examples aiming for many kinds of risks. In practice, your level of coverage probably will be more constrained and specifically guided.
* It is wise to test more important functions sooner and to spend more time on crucial aspects of a product. That calls for having a structured model and for having risks and priorities identified and monitored as the software development project unfolds.
* Data-driven API testing can enable feedback much sooner and more often during development while being just as comprehensive as classic functional black-box testing. Testers looking to diversify their skills should consider learning some coding in order to test their programs at the API level.

**Security**

* For the app I work on, we always make sure to test **authentication** and **authorization.** Even when we think we've got that mastered, mistakes can slip in.

**Error handling**

* is another area where we focus. We want **our users to get back a helpful message** if possible. We want to make it easy for people to use our API.

## **Reliability Bugs: Does API Testing Work Consistently?**

Getting the API to work once is a good start, but the API needs to work every time. API testing helps identify bugs pertaining to integration across different systems. Just because one application and server are working with the API, this doesn’t mean different servers and other applications are experiencing the same reliability. Additionally, the [tests help identify connectivity issues related to getting a response from the API](https://www.apicasystem.com/solutions/api-testing/?__hstc=215227080.725166821da610bbcf2e67b1c214d9fd.1593557970526.1593557970526.1593564304102.2&__hssc=215227080.2.1593564304102&__hsfp=1112991701).

**Performance**

testing [if the API works as expected, and if it works continuously under heavy load](http://apiux.com/2013/05/29/api-hierarchy-needs/). [how many requests the application can handle before response times start to spike](https://www.apicasystem.com/solutions/api-testing/?__hstc=215227080.725166821da610bbcf2e67b1c214d9fd.1593557970526.1593557970526.1593564304102.2&__hssc=215227080.1.1593564304102&__hsfp=1112991701).

the API tests should shift to gathering data that describes **performance response times** versus **the number of simultaneous users accessing the platform**. This information is used to predict how the API’s infrastructure will behave as the platform’s user base grows, helping your business identify things like how to expand infrastructure and how much power is necessary to handle peak demand.

API testing can also be very helpful with [**monitoring performance**](https://www.apicasystem.com/monitoring/monitoring-integrations/?__hstc=215227080.725166821da610bbcf2e67b1c214d9fd.1593557970526.1593557970526.1593564304102.2&__hssc=215227080.1.1593564304102&__hsfp=1112991701), which is especially useful when the development team makes changes. Measuring average response times before and after an update can tell the developers if the changes made the platform more or less efficient. The more efficiently the application performs, the less it will rely on powerful hosting infrastructure. Testing can also examine [individual endpoint performance, and provide greater insight into why an API might fail](https://www.apicasystem.com/services-api-testing/?__hstc=215227080.725166821da610bbcf2e67b1c214d9fd.1593557970526.1593557970526.1593564304102.2&__hssc=215227080.1.1593564304102&__hsfp=1112991701).

API testing is part of the integration testing process and should [determine if requests return data, check for event triggers activating, and verify data updates on both the client and host sides](https://www.tutorialspoint.com/software_testing_dictionary/api_testing.htm)

* Return Value based on input condition - The return value from the API's are checked based on the input condition.
* Verify if the API's does not return anything.
* Verify if the API triggers some other event or calls another API. The Events output should be tracked and verified.
* Verify if the API is updating any data structure.

## Test Cases for API Testing:

Test cases of API testing are based on

* **Return value based on input condition:** it is relatively easy to test, as input can be defined and results can be authenticated
* **Does not return anything:**When there is no return value, a behavior of API on the system to be checked
* **Trigger some other API/event/interrupt:**If an output of an API triggers some event or interrupt, then those events and interrupt listeners should be tracked
* **Update data structure:**Updating data structure will have some outcome or effect on the system, and that should be authenticated
* **Modify certain resources:**If API call modifies some resources then it should be validated by accessing respective resources
* Understanding the functionality of the API program and clearly define the scope of the program
* Apply testing techniques such as equivalence classes, boundary value analysis, and error guessing and write test cases for the API
* Input Parameters for the API need to be planned and defined appropriately
* Execute the test cases and compare expected and actual results.

|  |  |
| --- | --- |
| **Unit testing** | **API testing** |
| * Developers perform it | Testers perform it |
| * Separate functionality is tested | End to end functionality is tested |
| * **A developer can access the source code** | * Testers cannot access the source code |
| * UI testing is also involved | * Only API functions are tested |
| * Only basic functionalities are tested | * All functional issues are tested |
| * Limited in scope | * Broader in scope |
| * **unit-test: Usually ran before check-in** | * **API test**-Ran after build is created |

**How to do API Testing**

API testing should cover at least following testing methods apart from usual SDLC process

* **Functionality testing:** The test group should manually execute the set of calls documented in the API like verifying that a specific resource exposed by the API can be listed, created and deleted as appropriate
* **Usability testing:**This testing verifies whether the API is functional and user-friendly. And does API integrates well with another platform as well
* **Security testing:**This testing includes what type of authentication is required and whether sensitive data is encrypted over HTTP or both
* **Automated testing:**API testing should culminate in the creation of a set of scripts or a tool that can be used to execute the API regularly
* **Documentation:**The test team has to make sure that the documentation is adequate and provides enough information to interact with the API. Documentation should be a part of the final deliverable

## **Best Practices of API Testing**:

* Test cases should be grouped by test category
* On top of each test, you should include the declarations of the APIs being called.
* **Parameters selection** should be explicitly mentioned in the test case itself
* Prioritize API function calls so that it will be easy for testers to test
* Each test case should be as self-contained and independent from dependencies as possible
* Avoid "test chaining" in your development
* Special care must be taken while handling one-time call functions like - Delete, CloseWindow, etc...
* **Call sequencing should be performed and well planned**
* To ensure complete test coverage, create test cases for all **possible input combinations** of the API.

## **Types of Bugs** that API testing detects

* **Fails to handle error conditions gracefully**
* **Unused flags**
* Missing or duplicate functionality
* **Reliability Issues**. Difficulty in connecting and getting a response from API.
* **Security Issues**
* Multi-threading issues
* **Performance Issues.** API response time is very high.
* Improper errors/warning to a caller
* **Incorrect handling** of valid argument values
* **Response Data** is not structured correctly (JSON or XML)

## Challenges of API Testing

Challenges of API testing includes:

* Main challenges in Web API testing is **Parameter Combination, Parameter Selection, and Call Sequencing**
* There is no GUI available **to test the application which makes** difficult to give input values
* Validating and Verifying the output in a different system is little difficult for testers
* Parameters selection and categorization is required to be known to the testers
* **Exception handling** function **needs to be tested**

## **How to Authenticate to a REST API**

So far you’ve seen how to interact with open REST APIs that don’t require any authorization. However, many REST APIs **require you to authenticate to them** **before you can access specific endpoints**, particularly if they deal with sensitive data.

There are a few common authentication methods for REST APIs that can be handled with **Python Requests**. The simplest way is to pass your **username and password** to the appropriate endpoint as **HTTP Basic Auth**; this is equivalent to typing your username and password into a website.

requests.get**(**

'https://api.github.com/user',

auth=HTTPBasicAuth**(**'username', 'password'**)**

**)**

A **more secure method** is to get **an access token** that acts as an equivalent to a username/password combination; the method to get an access token varies widely from API to API, but the most common framework for API authentication is [**OAuth**](https://oauth.net/2/)**.** Here at Nylas, we use [three-legged OAuth](https://members.orcid.org/api/oauth/3legged-oauth) to **grant an access token for user accounts** that is restricted to scopes that define the specific data and functionality that can be accessed. This process is demonstrated in the [Nylas Hosted Auth service](https://docs.nylas.com/docs/hosted-authentication" \t "_blank).

Once you have an access token, you can provide it as a bearer token in the request header: this is the **most secure way to authenticate to a REST API with an access token**:

my\_headers = **{**'**Authorization**' : '**Bearer** {access\_token}'**}**

response = requests.get**(**'http://httpbin.org/headers', headers=my\_headers**)**

There are quite a few other methods to authenticate to a REST API, including [digest](https://2.python-requests.org/en/master/user/authentication/#digest-authentication), [Kerberos](https://github.com/requests/requests-kerberos), [NTLM](https://github.com/requests/requests-ntlm), and [AuthBase](https://2.python-requests.org/en/master/user/authentication/" \l "new-forms-of-authentication" \t "_blank). The use of these depends on the architecture decisions of the REST API producer.

**Use Sessions to Manage Access Tokens**

[**Session objects**](https://requests.readthedocs.io/en/master/user/advanced/#session-objects) come in handy when working with Python Requests as a tool to persist parameters that are needed for **making multiple requests within a single session**, like access tokens. Also, managing session cookies can provide a nice performance increase because you don’t need to open a new connection for every request.

session = requests.Session**()**

session.headers.update**({**'Authorization': 'Bearer {access\_token}'**})**

response = session.get**(**'https://httpbin.org/headers'**)**

#

Essentially, this is a common dilemma in designing an automated test. The design of a test is heavily dependent on it's intended purpose in the development lifecycle.

You stated that these are regression tests...implying that the current functionality is working correctly and the primary purpose of the test is to expose bugs caused by changes in the product code. Essentially, IMHO regression tests are a superset of unit tests, but are broader in scope.

**Regression test case:**

**Should be manually script-tested working; then integrated to regression suite.**

**Regression testing should be mostly working test cases.**

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 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. 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 tying to find previously undiscovered bugs, and test the robustness of that particular component is a variety of conditions.

**Terminology**

**Defining Acceptance Criteria Using the “Steel Thread”** Concept      About Contributions    Follow Send Message  Many times, teams struggle to define clear acceptance criteria. Acceptance criteria define the boundaries of a user story, and are used to confirm when a story is completed and working as intended. However, many times the acceptance criteria ends up being what the story is not supposed to be, or does not include enough information about the functionality expectations of the story, so that the teams don’t know exactly what to test. However, there is a way to simplify the process of defining acceptance criteria called the "Steel Thread" concept. The term "Steel Thread" refers to the idea that the system’s main functionality is like a "thread" that runs throughout the system. Everything is based on this thread, and it is therefore, very important. Its importance is what makes it strong, like "steel". The way to use the steel thread approach is to decide as a team just what constitutes the steel thread. This should be the main function or the primary function of the user story that provides some tangible outcome. Then the acceptance test can be focused and targeted to verify that everything within this main function or "steel thread" is working correctly. This does not mean that there is only one acceptance test, but that each acceptance test that is written cannot go outside of the limited scope of its "steel thread". The following is an example of using the steel thread approach for a user story that reads: "As a user, I can configure widget B to display in one of the three primary colors of blue, red, and yellow." A set of acceptance tests would be: 1.            User has access to the configuration options for System X 2.            User has a selection list available of the three primary colors of blue, red and yellow 3.            When a user sets System X to blue, it displays in blue 4.            When a user sets System X to red, it displays in red 5.            When the web master sets System X yellow, it displays in yellow 6.            When a user has no other options other than blue, red or yellow Now, if you need to check and make sure no one else has those abilities, you should write a different user story. This one might be: "As a user, I do not have access to System X configuration options". Then you would write another set of acceptance tests for this steel thread. By following the steel thread approach, your team can insure that each of the user stories is fully functional, and you can complete one user story in a particular iteration, without any unnecessary bleed over.  
  
more at: <https://www.agiledevelopment.org/agile-talk/111-defining-acceptance-criteria-using-the-steel-thread-concept>

[**Scrum**](http://scrumguides.org/) is the Framework in which a sprint takes place. A **Sprint** is a defined time period for developing features for a product. The maximum time for a sprint is 30 days (can be shorter but not longer). During a sprint the development team develops new features for the product. When the sprint is finished a new version of the product is available. This product could be shipped to the customer.

﻿

The definition of **sprint in Scrum** is quite simple. Like any other Agile methodology, Scrum is based on iterative cycles. They are called sprints. The length of a sprint may vary from 1 to 4 weeks. It depends on the complexity of the project and the amount of code that is to be written during the sprint. The average sprint lasts about two weeks. Such length is convenient because it allows the developers to write enough code to show the intermediate product to the Product Owner.

**Scrum is an agile framework** which many claim to work by (but if you actually spend time reading the Scrum documentation, you will see that few actual do).

A **sprint** is a Scrum iteration (cycle).

I recommend you read the following:

1. [Getting Agile with Scrum](https://www.mountaingoatsoftware.com/uploads/presentations/Getting-Agile-Scrum-Agile-Development-Practices-2010.pdf)
2. [The Scrum Guide](https://www.scrumguides.org/index.html)
3. [The Scrum Primer](https://scrumprimer.org/)
4. [Do Better Scrum](https://www.agile42.com/en/agile-info-center/do-better-scrum/)
5. [How To Fail With Agile](https://www.mountaingoatsoftware.com/articles/how-to-fail-with-agile)
6. [Kanban-vs-Scrum](https://www.infoq.com/minibooks/kanban-scrum-minibook/)

**Sprint** In product development, a sprint is a set period of time during which specific work has to be completed and made ready for review.

Each **sprint** begins with a **planning meeting**. During the meeting, the **product owner** (the person requesting the work) and the **development team** agree upon exactly what work will be accomplished during the sprint. The development team has the final say when it comes to determining how much work can realistically be accomplished during the sprint, and the **product owner has the final say on what criteria need to be met** for the work to be approved and accepted.

**Scrum**, the most popular agile framework in software development, is an iterative approach that has at **its core the sprint** — the scrum term for iteration. Scrum teams use inspection throughout an agile project to ensure that the team meets the goals of each part of the process. The scrum approach includes assembling the project’s requirements and using them to define the project. You then plan the necessary sprints, and divide each sprint into its own list of requirements. Daily scrum meetings help keep the project on target as do regular inspections and reviews.

**Sprint** is the minimum frequency at which a team that follows a scrum method can achieve tangible progress that can be demonstrated to the business owners or customers.

**Sprint is usually a few weeks (Commonly 2 weeks).**

In a different tutorial, we discussed how to [web scrape with python](https://medium.com/towards-artificial-intelligence/web-scraping-with-python-6d01f5e9378f). The goal of web scraping was to access data from a website or webpage. Well, sometimes a website can make it easier for a user to have direct access to their data with the use of an API (Application Programming Interface). This basically means that the company has made a set of dedicated URLs that provide this data in a pure form (meaning without any presentation formatting). This pure data is often in a JSON (JavaScript Object Notation) format, which we can then parse through and extract what we need using python.