These are the requirements that I have chosen to test, I've handpicked a mix of requirements that not only demonstrate diverse testing techniques but are also crucial to the project's success.

Lifecycle Approach

I will now share a little about the lifecycle approach I've opted for and how it aligns with the stages where these requirements will be tested and some risks due to my approach. I will generalize this for all my requirements.

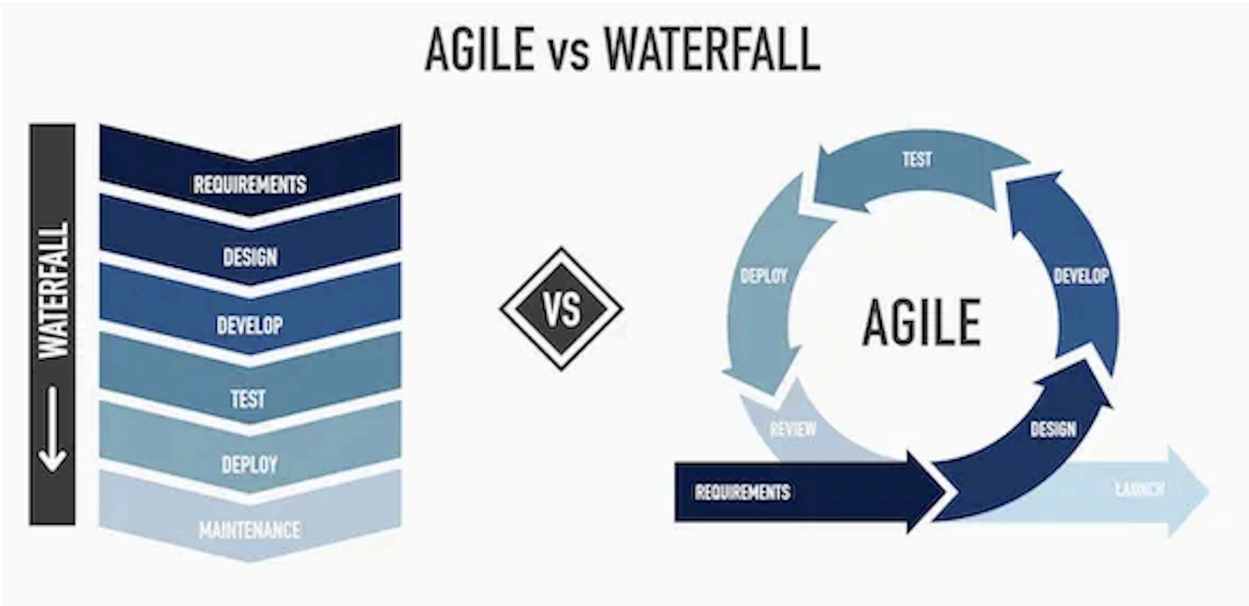
It's important to recognize that the initial development of this software adhered to a Waterfall model and bypassed the formal test stage before deployment. Therefore, for already implemented requirements I will align with the waterfall model because most of the development work is done, and what we need now is a thorough run-through for validation and verification of the software's functionality.

My aim for future development is to lean into an Agile approach. This decision works well with the recently adopted CI/CD pipeline. Developers pick up feature branches based on the requirements and design documents, develop these features, conduct tests, and then merge them into the development or production branches. At this point, our CD pipeline kicks in, handling the deployment seamlessly. This process really embodies the Agile philosophy of iterative development and rapid deployment, as depicted in the figure below, and is how I will handle requirements not yet implemented and deployed.

Risks due to lifecycle approach

A primary risk lies in the potential discovery of significant errors or oversights at this late stage in the development lifecycle. Given that the software was initially developed following a Waterfall model, any critical issues identified now, after most of the development has been completed, could be particularly costly and time-consuming to address. This is because making changes at this point would likely entail revisiting and altering previously completed work, potentially triggering a cascade of additional modifications and re-testing. Additionally, if these changes are substantial, they could disrupt the stability and integrity of the existing system, leading to further complications.

Shifting to an Agile approach for testing new features in a system initially developed under a Waterfall model introduces new risks. With rapid development cycles, there’s a danger that testing may become too focused on new features, neglecting the thorough examination of how these features interact with the existing system which was developed in the waterfall model. This could lead to integration issues or regression bugs, where new updates inadvertently disrupt previously stable functionalities.



1. User Authentication - Sign Up Requirements

Requirement Title	**Users can create new accounts with a unique username, email and password once all of the restrictions below are met.**
A&T Needs	This requirement needs to ensure that the form correctly processes the form data when it is all valid. This means that it must accept the data and create a user for our web application. In this test case, we will not consider invalid or garbage data as that is covered in the other requirements and test cases. A user being able to create an account is quite a high-level requirement as otherwise new users will not be able to use the service, we should therefore put quite a few resources into this requirement. To validate this requirement, in terms of the front end, we will need to ensure that the correct and expected response is sent to the backend for valid form data. To test the correct account creation, we need to check that a valid entry is made in the database upon when the backend received a request in this format. We will need some synthetic data to run these tests on, and some way to mock up the database and backend responses if we are not doing a system level test which integrates the frontend and backend functionality. I opted not to do a system level test, as that will require a test database, as we do not want to create fake users in the real database every time the test is run. This is outside the scope of the work I can put into this course, as the trylinks database setup is quite complicated and would need another instance of the frontend and backend in a test environment.
Assessment of A&T plan	We cannot be certain that every valid form data is accepted, just the ones we test. The large input space of values for username, password and email pairs make it impractical to test for every input and therefore we should use a smarter approach, we can make use of equivalence partitioning and boundary value analysis to select

	test cases effectively.
Testing Phase	This requirement can be tested early in development, as it does not rely on any of the systems except for the registering process whereby a user gets added to a database. However as this requirement is already developed, we will do it in a separate testing phase, akin to the waterfall model. I also aim to implement automatic tests that allow for continuous integration testing, as sign up functionality is a good indication of whether the FE/BE are up and working, and whether users can access the site (sign in will be the same).
Required Instrumentation	This tests a pretty complete and complicated functionality that spans the front and backend, so we will need: -> Some way to feed responses to the backend, and mock up a database, which can later be inspected for correct account creation, possibly Mocha/ Jest as a Test Harness -> We should have a generator for synthetic but realistic user data to test a broad range of valid inputs. -> Will need to learn to use Sinon.js for mocking backend responses and database interactions, enabling isolated testing of the frontend and backend components.
Assessment of Instrumentation	There are much better methods of instrumentation if time and resources allowed, however unfortunately this is not the case and I cannot learn everything I need for this course in time. For example, instead of mocking the server responses and database, which already takes quite a few resources, we could use the actual front end and back end instanced under a test environment using tools such as a fully integrated test environment using tools like Docker to containerize the actual frontend and backend services which would provide a more realistic test setting and facilitates end-to-end testing that closer mimics the production environment. This is generally not recommended as this relies on all modules being integrated and skips straight to system testing however my application is already mostly complete with all the parts integrated, so therefore this would be a good approach if I could gain the skills or knowledge in time. The positives of my approach, which involved more scaffolding, is that it is more efficient, and we can test the FE and BE independent of each other, preventing dealing with CORS and other issues at this point. However, a real test without mocks under a test environment should definitely be done in some cases to ensure functionality, if resources allow. We could also use the Capture and Replay technique to simulate real user interaction through the frontend, which is more accurate than sending and receiving requests.
Type of testing	We will use structural testing first, after the functional testing for the smaller requirements to do with user authentication has been completed. We will use statement coverage and branch coverage to ensure that all of the FE's lines of code for this class are executed.

Requirement Title	**Users will be required to confirm their password another time during the Signup process, and if the two passwords do not match, the user will be prompted and the account not created.**

Software Testing Test Plan

A&T Needs	This is a much simpler requirement than the last as it does not require communication between the front-end and back-end. This requirement merits a small to moderate allocation of resources to help user experience, especially as the app currently has no password reset functionality, so we definitely do not want users to sign up with a password containing a typo that could have been ea The code must simply check if two textboxes have equal text, then display a prompt to the user if this is not the case. Furthermore, the code needs to check that a request to the backend is not sent if a user tries to create an account where the fields are not even. Now let's think about what is required in order to test this. We should decompose the requirement into two main components, the front-end logic for password matching, and the UI component for displaying the prompt. We will need to develop unit tests for the password confirmation feature, and ensure that it is accurate, the form should be invalid when the passwords do not match. The passwords used for match tests should be partitioned into class based test cases. Then, we will need to do some UI testing to make sure that a prompt is displayed correctly when the passwords do not match, and taken away when they do match. Finally, we must ensure that the correct set of code is executed when the sign up button is pressed and the password is not matching. This must NOT send a request to the backend. Inputs: Password1 : The first password entered by the user Password2: The second password entered by the user Outputs: passworNotMatched: Boolean indicating if the passwords match Specification: passwordNotMatched controls visibility property of prompt (visual inspection) passwordNotMatched is true only if passwords match (unit test) passwordNotMatched controls whether request is sent
Assessment of A&T plan	***The following paragraph has changed, we gained the resources to do DOM testing*** For now the plan is to only check the visibility of the prompt to the user using visual inspection and cross browser testing, as it is quite a simple piece of code that does not have consequence other than usability, however, If I had more resources I could make use of some UI testing frameworks like Selenium to programmatically check if an element is rendered. There are also some more interesting techniques like DOM state checking, which would allow for easier automated testing and reduce expensive human time. Checking if 2 passwords are equal is tricky as the input space is too large but that would be the most effective. Knowing that errors usually occur on the edges, I believe that it is better to select test cases by dividing them into classes, e.g. both fields empty, both fields same string, both fields different string…. As well as test representative values from each class, such as using symbols and special characters where both fields match.
Testing Phase	Hypothetically, since this feature is largely front-end centric and independent of the

	backend logic, testing can commence early in the development cycle, even before backend integration. This also aligns with Chapter 4's emphasis on early detection in the development cycle. However, since our testing is occurring on a completed software, we will conduct testing in a separate testing phase, outside of development.
Required Instrumentation	-> Utilize a testing framework like Jasmine for writing and Karma for executing unit tests specific to Angular applications. We will use a comparison-based oracle with a small number of simple test cases derived from the classes mentioned above.
Assessment of Instrumentation	In an ideal scenario with more resources, a more extensive use of automated UI testing tools would be beneficial, this would be the capture and replay technique from Y&P with some sort of prompt detection. This would interact directly with the GUI instead of manipulating the code and functions, and inspecting the variables/dom which is more accurate to what a user experiences and may find stuff we miss. However, it is also quite expensive, as there is a lot of setup and learning of new tools needed to execute everything through the UI and have some way to visually detect the prompt, which are limited resources for a 10 credit course. This is also likely only feasible for a small number of test cases as it will be more resource intensive.
Type of testing to use	Functional testing

Requirement Title	**The username must only contain alphanumeric characters, and be between 5 to 20 characters in length, if these conditions are not met then the user will be prompted and their account will not be created.**
A&T Needs	A very simple requirement that will require the following functional testing: Input Validation: The primary focus should be on validating the input for the username field. Tests should ensure that only alphanumeric characters are accepted. Length Restriction: Ensure that the username does not exceed 20 characters and is not under 5 characters in length Error Handling: Verify that appropriate error messages or prompts are displayed when invalid data is entered. We will once again use class based test cases. I will need to create both valid and invalid inputs for the usernames. Valid usernames will include various alphanumeric combinations of characters 5 to 20 in length. Invalid inputs will test usernames containing non-alphanumeric characters, and usernames exceeding 20 characters in length or under 5 characters in length. We will also need to generate edge cases, such as a 4,5,6,19, 20 and 21 character usernames, and usernames with exactly one non alphanumeric character. When this requirement is not met, we must not make a request to the backend, this will be tested exactly as in the requirement above.

	Error handling testing will be visual inspection based on the variable state as in the requirement right above.
Assessment of A&T plan	This plan covers basic functional requirements for input validation and length restrictions but may not fully account for edge cases in user input as humans are prone to error. Furthermore, due to lack of performance and access to user feedback I am not exploring usability testing which definitely needs to be done to track if this functionality maintains a good experience for our target users. The assessment mentioned in the previous requirement also applies here too.
Testing Phase	Hypothetically, since this feature is largely front-end centric and independent of the backend logic, testing can commence early in the development cycle, even before backend integration. This also aligns with Chapter 4's emphasis on early detection in the development cycle. However, once again we will conduct this in a separate testing stage as per the waterfall model.
Required Instrumentation	-> Implement functionality to generate mock user input that satisfies the test cases, including the edge cases. We do this instead of manual test cases to account for bias, and so that they are always different (but logged), for every test run in our eventual ci pipeline. -> Utilize a testing framework like Jasmine for writing and Karma for executing unit tests specific to Angular applications.
Assessment of Instrumentation	In an ideal scenario with more resources, a more extensive use of automated UI testing tools would be beneficial for the user prompt instead of relying on visual validation and a boolean variable.
Type of Testing	Functional Testing

Note : We will now move on from the user authentication requirements in order to highlight other types of testing strategies, A&T needs and instrumentation.

3. Non-Functional Requirements

Requirement Title	**Every field that contains user input must be validated and sanitized to prevent common exploits like SQL injection, XSS, etc.**
A&T Needs	This is quite important to test, as a malicious user can easily delete our whole database if our input fields are susceptible to these exploits, so I should put some effort to convince myself that we are as safe from SQL attacks as budget allows. The first step, as this is an already developed piece of functionality, will be to conduct a comprehensive review of the existing codebase. I will focus on how user input is handled, and will evaluate what libraries are used and if they are generally safe, or vulnerable to SQL injection. This can help decide which fields we put more manual effort into, or maybe completely change to safer libraries. Following this, we need to ensure that every user input field validates and sanitizes incoming data, in order to mitigate risks such as SQL Injection. We will use a risk based approach, and focus first and most on fields that interact more directly and critically with the database. Once happy with the manual code inspection, and the required changes

	have been made, I will need to introduce tests to give more confidence that the program is secure from these vulnerabilities. I will then Develop unit tests that simulate SQL attacks, to see how the application handles and reacts to these, e.g. an attack that tries to drop the SQL tables. I will then do some manual penetration testing on each field and log the outcomes.
Assessment of A&T plan	Whilst my A&T plan is a good start and moderate defense against SQL injections by providing a strong foundation and guarding against attacks, through code review, unit testing and manual penetration testing, security is an ever evolving field, and regardless of the 'secure' libraries I use to manage input, or the manual penetration testing strings I come up with, new vulnerabilities to exploit input fields, or stuff I missed is bound to come up if an attacker puts enough effort in. If a bigger budget, time and skill existed, I could do much more to secure the application. I could use some static application security testing in order to scan the code for SQL Injection vulnerabilities without executing it e.g. using SonarQube. Alternatively, I could use a Dynamic Application Security Testing tool like Burp Suite to interact with the running application and test it for security vulnerabilities. Finally, I could hire professional penetration testers to test the software at regular intervals.
Testing Phase	Testing will begin as soon as the input validation, sanitization and secure libraries are verified to be, or are implemented, which in our case occurs after the initial development phase.
Required Instrumentation	Utilize OWASP's SQL Injection Payload List as a comprehensive source for test strings. This list provides a wide range of attack vectors, ensuring thorough testing of your application's defenses. Consider categorizing these payloads based on the type and complexity of attacks for more structured testing. Postman or custom scripts can be effectively used to simulate requests to the backend. This approach allows me to automate sending various SQL injection payloads to test how your backend handles malicious inputs. May need to implement a controlled test database environment that should be secure and isolated environment, Using a postman may cause CORS issues which may need to be bypassed by turning off CORS and testing in a local environment.
Assessment of Instrumentation	Currently, I plan to use some test strings generated by security minded individuals, however this is just a start. With more time and resources, the inclusion of automated security scanning tools would significantly enhance testing efficiency. Tools that can automatically send a variety of malicious payloads and analyze responses would provide broader coverage. Implement sophisticated monitoring and logging mechanisms that can detect, alert, and record potential security breaches in real-time, providing invaluable data for ongoing security analysis.
Type of Testing	Fault-Based testing

Requirement Title	**After submitting correct login details, the user should be allowed into the website within 3 seconds of hitting the login button.**
A&T Needs	This is a system level requirement, as it will not be realistic if measured in a sandbox where it is the only functional part of the web app nor will it work if the backend/front end is mocked as we need a more realistic estimation of how much load will be on the server per real user. We will need to implement a performance testing approach that mimics real world scenarios, this will use synthetic data to simulate user behavior, including using the app normally and logging in. We will need synthetic data and users that are in the database that we will login as. We will need to implement an accurate timing mechanism to measure the duration from the moment the login button is pressed, until the user is fully logged in and the next intended page is loaded.
Assessment of A&T plan	This is a new web app with no known users, so it will be very hard to simulate real-world usage accurately. Adding to this, we do not know the required load that the application has to be able to handle, especially since the app is very niche so it is hard to create an accurate load model.
Testing Phase	This is a system level test that occurs quite late and can only be tested on a complete system.
Required Instrumentation	-> A deployed system on which we can perform e2e tests. -> The test user in the db who can be used to perform actions. -> Some sort of simulator to simulate user activity on the website. -> A system to count the time it takes for a user to be logged in (redirected to the dashboard) after they submit their details.
Assessment of Instrumentation	There is no guarantee that this requirement will continue to be met under different values of load, which we are not planning to test here. This should be done when more resources are available. However even if we do this, this is a brand new application, and hence we have no real user data to generate realistic simulation of user activity at a realistic load. The system should be automated in some manner, so that future improvements to the efficiency of the process, or when using a different computer(server) can once again validate that this requirement is met, but again we lack resources for this.
Type of testing	Performance based

Requirement Title	**2.4.2 The system should be easy to maintain, it should handle this by having an organized Github Repo with a CICD pipeline, standardized commits and feature branches as well as a develop and production branch which are each deployed automatically when changes get pushed.**
A&T Needs	This focuses more on system maintainability and organization, this aspect is more about development practices and infrastructure setup rather than functional software testing, but I believe that it is still important to assess and audit to ensure compliance. I must review the repository regularly to ensure that it's organized and uses the correct commit and branching strategy, in our case conventional commits, standard branch naming conventions and the branching strategy. I must assess the CI/CD pipeline, ensuring that it automates tasks like builds, tests and deployments effectively. I must review the commit history to ensure its adherence to standardized commit messages, and automate this when possible using Git hooks that enforce standards. I must produce a github wiki, educating new developers on contribution guidelines and rules so that this is maintained in the future.
Assessment of A&T plan	This plan may not be very practical as it relies heavily on manual inspection, and whilst there are a few techniques to ensure compliance (like github hooks) and some techniques to educate developers, things can get out of hand quickly on a public repository without manual inspection.
Testing Phase	This is a test that will have periodic reviews and a document that will keep track of issues, which can be used to update the wiki on contributing guidelines and common pitfalls.
Required Instrumentation	Git hooks to enforce compliance with standard commit messages. Documentation in the wiki for contributing guidelines.
Assessment of Instrumentation	The instrumentation is too limited and requires too much manual oversight, however as I am currently the only developer and I do not expect this niche product to grow too much, I believe that this is a good start with potential to improve in the future using techniques such as custom scripts to enforce branching conventions, PR templates, protected branches.

3. Test cases gotten from UML diagram (System Level, end-to-end)

A&T needs : An end to end test will be written for each of the requirements below. This will be done using a library called cypress.js, which will use the code i write to automate a chrome browser, which will connect to the domain of the hosted website and carry out automatic end-to-end testing.

A&T assessment : This is a good way to test an already deployed application, as it is the least time investive way to gain some confidence in the system performance as a whole, not having to do scaffolding means that the system can be tested faster, but there are also downsides hence why unit and integration tests should be further developed in the future.

Testing phase : This occurs at the end of development, as these are end-to-end system tests that rely on the system (or parts of it) being up and running.

Required Instrumentation: A cypress.js installation, a test user to login with and perform e2e tests.

Cases to test that have been gotten from UML model:

1. When a user visits trylinks.net, they get redirected to trylinks.net/welcome.
2. If a user clicks the sign-up button, they get redirected to trylinks.net/login.
3. If a user, who is still unauthorized tries to visit trylinks.net/dashboard, they get redirected to trylinks.net/welcome
4. If a user, who is still unauthorized tries to visit trylinks.net/tutorial, they get redirected to trylinks.net/welcome
5. If a user, who is still unauthorized tries to visit trylinks.net/interactive, they get redirected to trylinks.net/welcome
6. If a user enters an invalid username, password and hits the sign-in button, the sign in fails and the user is prompted.
7. If a user enters a valid username, password par and hits the sign-in button, the sign in succeeds and the user is redirected to trylinks.net/dashboard
8. If the user clicks signout from within the dashboard, they become unauthorized and are redirected to trylinks.net/welcome
9. If the user clicks Launch Links Interactive Mode, they get redirected to trylinks.net/interactive

.
.
.
.

.

(see UML diagram for the rest)