SENG 426 – Summer 2022 – Course Project

Lab Assignments

Dr. Issa Traore
07/05/2022

Table of Contents

Project Execution and Deliverable

The project/lab will be performed in teams of six. Students must form their team at the beginning of the term and notify the lab TAs by sending an email to both TAs:

Sanjay Dutt (sanjaydutt.india@yahoo.com) and Hadeer Ahmed (meresger.hs@gmail.com).

There will be a forum in Brightspace through which you can search for teammates.

Important Note: Team formation is entirely the responsibility of the students. Neither the instructor (Dr. Traore) nor the TAs will be involved in dispute resolution between teammates. If you are unhappy about your teammates, you can change group at the end of each deliverable. However, we will not arbitrate any disagreement over performance issues among teammates.

Deliverables are due on the due dates listed below by 11:59pm the latest. The deliverables must be submitted through Brightspace. The labs will be marked by either Mr. Sanjay Dutt or Ms. Hadeer Saad. Students are encouraged to check the lab schedule and organize their work around it.

Note: It is strongly recommended to read the current document entirely before starting any lab activity.

Project Learning Outcome and Deliverables

The project centers on applying advanced testing techniques and tools for checking and improving the quality of a web application system. The focus will be on continuous integration (DevOps), automated functional testing, performance testing, and security testing.

The main objective of the project is to learn, by applying the quality control techniques seen in class and your own software engineering background (both in terms of design and coding), how to expose and fix reliability, performance, and scalability issues and software vulnerabilities in enterprise software systems.

The objective of this document is to give an overview of the basic functionality of the system under testing as well as the amount of work that needs to be done to complete the different steps of the project. Below is a list of the project activities and tasks along with the due dates of the various deliverables.

Project Deliverables

Part 1: Requirements Specification (5%)	Due May 29, 2022
Part 2: Continuous Integration Setup (5%)	Due June 12, 2022
Part 3: Automated Testing & Continuous Integration (14%)	Due June 26, 2022
Part 4: Performance & Scalability Testing (14%)	Due July 17, 2022
Part 5: Security Testing (12%)	Due: August 1, 2022

System Overview and Requirements

Basic Functionality

Vega Absolute is a Cyber security company. The company has a public website for its employees and customers, and a private network. The site provides basic information about the company corporate structure, products, and services, and allows customers to use various services provided by the platform.

As a Cyber security service provider, the reputation of the company relies not only on the quality of their services, but also on their ability to protect their own network and website.

Basic requirements for using the Cyber security services are as follows:

1. A user must register first to create a new account.
2. Once the customer details are provided, by default, the requested account will be in deactivated state until an Administrator approves the account. The user will not be able to use the services until the account is approved.

Site Users and Permissions

The system supports five types of users – regular visitor, customer, employee, and admin – with the following capabilities:

Regular site visitor:

- Can access and view public pages and register to become a Vega Absolute user.

Customer:

- Can use cyber services provided by the Vega Absolute platform such as the Password Manager.

Employee:

- Can use platform services
- Can check resources tab to see uploaded files by admin

Admin:

- Enable registered users
- Can change Role of registered User to "Staff" or "User"
- Can upload files at Resource tab

Git Repository

The Vega Absolute webapp Git repository is hosted on Github server at the following link:

https://github.com/UvicSENG426

During the course you should follow the GitHub flow branching model and some of the best-practices related to task management. In practice, that means every feature or bug fix should have its own Github issue ticket with one feature branch created for each ticket being worked on. When merging back to the master branch, create a Pull Request (PR) so that at least one other member of the group can perform a code review of the proposed changes. The reviewer(s) should evaluate the PR and propose improvements or alternate implementation if necessary. After the review process is done and at least one reviewer has approved the PR, it can be merged to the master branch and the Github issue can be closed.

Note that no commit should be added directly to the master branch except for merges from feature branches as described above.

Each ticket created should have a description, a priority, an estimation of how long it will take to be fixed, and the **origin** of the issue, that is, which process led to its discovery.

Possible origins include requirements specification, design, coding, unit testing, integration testing, and system testing. Feel free to select any subset of origins that is relevant to the development process of your team. Note that the data gathered per origin will be used later to compute quality metrics.

There are two common ways of estimating tasks: Story points and hours. Pick one of the two methods to be used throughout the course considering the pros and cons of each method.

To help guide the creation of tickets, there will be a template available on GitLab's new issue page. To use it, select it from the "Description" dropdown menu.

Part 1: Requirements Specification & Backlog Building (5%; due May 29, 2022)

You are expected at the end of this phase to deliver a requirement specification report that documents the user stories of the application and to build a backlog of known issues based both on the report and the knowledge acquired from the process.

This phase is divided into 2 tasks, one for each of the above-mentioned goals, however, before starting the actual process you should first fork the repositories from https://github.com/UvicSENG426. You will find two repositories here: vega-web (Frontend) and vega-spring (Backend).

Task I: Requirements Specification

Provide a requirement specification report that documents the user stories of the application, which can assist in its quality control process.

Once you are able to clone the Vega repositories, open it using your favorite IDE, and run the application. Read the README.md file for detailed instructions.

Once the application is running, start interacting with it by trying the different functionalities offered and identifying all its features. Then, separate each feature into one or more user stories that collectively describe every capability of the application. Write every story in Gherkin following the best practices presented in class.

Note that the user stories represent the expected behavior of the system regardless of any defect that might be present. Therefore, you should wear your Product Owner hat and describe features as you think they should be (given the time constraints of the course) and not as they are implemented. You do not need to mention any bugs and poorly designed aspect of the application in the report, but you should keep track of them for Task II below.

Finally, propose a new feature -- **Do not pick functionality too trivial or too complex** -- for the Vega App by writing user stories in Gherkin which describe it in detail. Your team will be implementing this new feature in the 3rd Milestone. Note that the TA will confirm the validity of the proposed functionality as part of the delivery's feedback and might request the selection of a different functionality should the proposed one be outside the required difficulty range. The final feature must be confirmed by the TA before delivery of Part 3.

Task II: Backlog Building

The goal of this task is to build a backlog of known issues based on your requirement specification report.

The process consists of creating a GitLab issue for every bug or poorly designed aspect of the application identified in Task I. As described above, each issue should have a description, a priority, an estimation, and an origin. You should use the related stories from the report as the description since the stories describe the expected behavior of the application after the defect are fixed. That means each description can be composed of one or more user stories.

For the priority, consider the scope and importance of the defect and how much value will the fact of resolving the ticket provide to the users.

After filling the backlog with known issues, it is time to add tickets for new desirable features of the application. Create GitLab issues for the following features:

Vega Absolute Vault

It is a vault where customers can store their secrets.

Users should be able to manage their secrets

1. User should be able to add a new secret.
2. Secret Entry should contain the following information:
    a. Name of the secret
    b. Date/time of the secret when secret was created.
    c. Data -- text, image, file, encrypted keys.
3. On Manage Secret screen. User will see a list of secret entries.
4. While sending over the network, secrets will be encrypted.
5. Secret will be encrypted before saving to the database.
6. User can share a secret with other people.
7. User can perform all CRUD(Create/Delete/Update/Delete) operations on secret entry.
8. User should be able to filter Secret entries by entry date (date from / date to).

Admin role with a simple reporting

1. Admin can see a screen with all added secret entries of all users and manage existing secret entries (read, update, create, delete).
2. Admin should also see the report screen with the following information:
    1. Number of added entries in the last 7 days vs. added entries the week before that. Please, include the current day in those stats.
    2. A regular user should not be able to access this reporting screen or access its data.
3. Ensure a user can add a maximum number of Secrets per day. Adding more will result in error on screen. Admin will set the limit for each user.

New functionality proposed by your team

As before, the issues should be described by the related user stories. As for their priority, since these features are considered highly desirable, mark them as high priority.

At this point, your backlog should have several tickets with varying priority. In Part 3 you will pick the most important ones to work on.

The deliverable of this phase consists of:

1. The requirement specification report containing:
    i. The user stories of both the already implemented and proposed features.
    ii. A comparison of the pros and cons of using Story points or Hours for task estimation.
    iii. The definition of which of the two estimation methods was chosen.
    iv. A table with the title, priority and estimate of each ticket created.
2. The GitLab issues forming the backlog.

Part 2: Continuous Integration Setup (5%; due June 12, 2022)

The two main goals of this part are to configure Docker to facilitate development and setup a Continuous Integration (CI) server in anticipation of adding new features, creating automated tests and fixing defects in the application (in the next part).

The Vega Absolute application comes with several functioning Docker Compose files, including the application's Compose file which runs an application container and a MySQL container as well. Your task is simply to configure this file according to your environment and use it to deploy the application during development.

The next task consists of setting up a Microsoft Azure CI/CD Pipeline to do the following, after each push to the group's Git repository:

- Build the application for production.
- Scan the application using SonarQube.
- Run the integration tests.
- Notify the group should any step fail.

Task for this Milestone.

1. Create an organization named "SENG426Labs" on AzureDevOps.
    a. Add your team members.
    b. Add TAs to that organization.
    c. Import GitHub repositories.
2. Create a Build Pipeline for both "Vega-spring" (backend) and "Vega" (frontend). The Build pipeline will do the tasks below:
    d. Compile it.
    e. Run Unit test cases.
    f. Run Integration test cases such as Selenium tests.
    g. Add a SonarQube scan stage to the Pipeline so that the application's source code is analyzed after build – Note that at this stage the SonarQube scan results don't have to be analyzed yet.
    h. Destroy the resources.
    i. Build Artifacts and push it to Docker registry.
    j. Add a notification mechanism to inform at least one member of the group when any of the deployment steps fails. This can be accomplished in several different ways, such as by sending an email, posting to any external notification system or by integrating with the group's Slack, if the group has one.
3. Set up the release pipeline to deploy the application on Microsoft Azure.
    a. Publish the Build Artifacts on Microsoft Azure.

The deliverable of this phase is a report containing:

1. A step-by-step demonstration of the Docker container being deployed with screenshots.

2. An explanation of (1) what Docker Containers and Images are, (2) what is the difference of purpose between the Docker and the Docker-compose tools, and (3) how is the Vega app image created including which tools are involved in the process.
3. A step-by-step description of the Microsoft Azure and the GitLab setup with screenshots and an explanation for each configuration choice and action taken.
4. The description of every change made to the application files in order to support the Continuous Integration process, including the build pipelines and the Docker file(s);
5. Screenshots of the SonarQube report and issues pages.
6. Screenshots showing the build history, trends, and other Microsoft Azure reports.
7. An example of a notification when a build, scan, deployment, or test fails.
8. A commentary on the utility of Continuous Integration (CI) for software development.
9. A small proposal (one or two paragraphs) on how the CI structure could be improved or otherwise augmented in this scenario. For instance, mention how stages could be changed or reorganized and which tools could be added or better used so the CI process is improved.
10. A commentary on the GitHub flow model and a comparison with other commonly used branching models.

Part 3: Automated Testing & Continuous Integration (14%; due June 26, 2022)

This part consists of 2 separate sets of tasks. Although it is not required, you may conduct Tasks I & II in parallel. In this case, you can split the team so that some of the members can start fixing bugs and implementing the new features while the remaining members can work on the testing. During this phase you will use the CI server previously set up to maintain your code integrated into your testing environment.

You must also track the actual time spent working on each issue and update the respective ticket with this data before closing it. At the end of the assignment, calculate the following according to the chosen estimation method:

- If story points: The team's velocity, as an average of the number of points completed per week.
- If hours: The accuracy of the estimation, as the quotient of the actual time spent vs. the estimated time.

Finally, consider how the value would change (if it would change at all) with time if the team becomes more experienced and more familiar with this project.

Task I: Automated Functional Testing

The goal of this task is to improve upon the initial automated integration test suite provided by creating new test cases that cover all the user stories related to:

1. Account creation by customers. Activation of account by admin only after which customer should be able to access the account.
2. Role Change should be only done by Admin I.e., ROLE_STAFF or ROLE_USER.

3. In Vega vault, CRUD operations of secret by customer.
4. Secrets should only be visible to other customers only until unless it is shared among other customers as well.
5. Transfer ownership of secret from one customer to another customer.
6. File Upload by Admin and that should only be visible to Admin and Employee.

The test cases must be implemented using the **Selenium WebDriver** library in either Java, JavaScript (Node.js) or Python. Document every test case by explicitly identifying the user story it covers.

Each test case must be self-contained and test only a small facet or variation of the user story. This means that to cover each user story, at least one, but usually several test cases will have to be created. Include both positive and negative test cases when necessary and test every possible variation of the features listed above.

Note that it is possible that some bugs will prevent some test cases from passing. You should prioritize those bugs for Task III so that at the end all your test cases pass. If you find more bugs during this task, add them as GitLab issues and follow the same process.

Every day, collect test execution results, and in the end, plot the defect arrival graph (i.e., Number of Defects vs. Day) and comment it by discussing the different quality characteristics of the released application before and after this task was performed. Using the defects information collected from the tickets, provide the defect dynamic model (i.e. table or matrix) and calculate the defect removal effectiveness (DRE) for the different phases.

Task II: Implement New Functionalities and Fix Bugs

The goal of this task is to simulate the day-to-day development of an application. As such, you are tasked to implement new features and fix bugs in the application. In practice, this means picking from the backlog the tickets that have the highest priority, and which collectively fit the time available.

This includes implementing the 3 new features as well as fixing the most important defects of the system and, if there is time, a few lower priorities but quick-to-fix issues.

Note that a proper feature PR (Pull Request) consists of the feature code as well as new integration test cases as described in Task I.

Before delivering the assignment, generate a release of the Vega app. This is the version which will be marked. Note that the version number should be bumped to match the release tag generated. Be sure to push everything to the GitLab repository.

Deliverable for Part 3 (Tasks I & II)

The deliverable of this phase is:

1. The release (tag) available on the group's repository containing the updated code and the test suite.
2. The GitLab issues and Pull Requests.
3. A report containing:

i. The test cases with test data values.
ii. A description of the uncovered bugs.
iii. The defect arrival graph.
iv. The defect dynamic model and computed DRE measures
v. Commentary on different quality characteristics of the released application before and after this part of the assignment.
vi. A description of how the new features were implemented.
vii. The new user stories.
viii. Screenshots showing the Jenkins' build history, trends, and reports.
ix. A table with the title, original estimation and actual time spent on each ticket closed.
x. The velocity or estimation accuracy calculation.
xi. The consideration of whether and how the value would change with time.
xii. A commentary on whether doing CI provided any benefit or hindrance during this assignment that would be different in a real professional environment.

Part 4: Performance & Scalability Testing (14%; due July 17, 2022)

The goal of this phase is to perform load and stress testing of the web application by varying progressively the workload and measuring response time, throughput and CPU utilization; and then plotting and analyzing the obtained results. Before each run, you should populate the database with the multiple number of customers, each having at least one secret.

This phase consists of the following tasks:

Task I: Load Test

Load test the **server-side** functionalities by creating a test plan simulating the behavior of a user while interacting with the system based on the following scenario:

Each virtual user should start by visiting the home page of the application and logging into the system only once. Then, at each iteration, the user performs different actions with different likelihoods:

1. Navigate to profile page 100% of the time.
2. Navigate to Vega Vault page 50% of the time.
3. Create or delete the secret 5% of the time.
4. Create and then transfer ownership of secret to other customer 2% of the time.

Run the scenario in loop over 60 minutes starting with 1 user and adding a new user every second until there are 200 concurrent virtual users, and collect the following data:

1. Average, min and max response times per action (sampler) and aggregate.
2. A chart of the response times over time per action (sampler) and aggregate.
3. The error rate per action (sampler) and aggregate.
4. A chart of the response codes over time.
5. A chart of the number of active threads over time.
6. Average, min, and max CPU usage.
7. Average, min, and max memory usage.

Based on the above test results, calculate the software availability using the aggregate model presented in class.

Using the measured aggregate response time, and the selected think time, and maximum number of (tested) virtual users, calculate the average system throughput (based on the response time law).

Using the measured resource utilizations, calculate the maximum achievable system throughput (based on the Throughput Bound Law).

Provide a detailed description of the system's behavior as the number of users increases and after it stabilizes according to the data collected. Report if you notice any abnormal behavior and how would you further test the performance of these functionalities.

Finally, you should note that:

1. Each virtual user has its own login information which is not shared among users, and it will only login once, at the beginning of the test.
2. Each virtual user acts sequentially, meaning it will only perform the next action after the previous request is finished.
3. The action probabilities are independent.
4. There should be a realistic think-time between each action.
5. Since this test is only concerned with the server performance, each simulated action should be mapped to the relevant back-end calls. For instance, when requesting a new account, the user must retrieve the list of branches before actually making the new account request.
6. The JMeter application should run in a different machine than the target server. The test should run in non-GUI mode collecting only the results file (.jtl). After the test, load the result file into the JMeter test plan to generate the statistics and charts. This is done to reduce the influence of the JMeter application execution and collection process on the test results.
7. The machines should be on the same LAN to reduce network interference in the results.

Task II: Stress Test

Using the same test plan as above as a basis. Modify the test plan so that 1 virtual user is added every 2 seconds up to a total of 500 virtual users with a timeout of 20 seconds; set up a remote test environment with 2 or more JMeter servers (slaves) and answer the following:

1. How many concurrent users are necessary to increase the average response time to more than 2 seconds?
2. How many concurrent users are necessary to increase the error rate to more than 5%?
3. How many concurrent users are necessary to degrade the system's performance enough, so it becomes unusable for the majority of the users? Can this even be achieved, and if so, how does the system behaves in that scenario? Does it enter an unrecoverable state, hangs or crashes, or does it return to normal after the stress is reduced?
4. Do you believe the system's performance allows it to be released into production? Consider the hardware, network, user base vs. peak concurrent users, variations in user behavior, performed actions, think-time etc.

Include screenshots or charts to corroborate your answers.

As with the load test, the target server should be on a different machine than each JMeter server, but they should all be on the same LAN. The JMeter master can share the machine with 1 slave, therefore a minimum of 3 machines will be required.

All those restrictions are in place in order to reduce external interference on the test results and because of the risk of saturating the client machine instead of the target server, so it is important to follow them thoroughly.

Remember that each virtual user should have its own credentials, this includes virtual users of different slave machines.

Deliverable for Part 4 (Tasks I & II)

The deliverable of this phase is a zip file containing:

1. The JMeter test plans created.
2. The result files (.jtl).
3. In the report:
    1. Document the tasks.
    2. Comment the obtained results.
    3. Explain your test organization (test elements and structure) and configuration choices.
    4. Describe the steps taken, configuration changed, and commands executed to run both the load and stress tests. Include screenshots when possible.

A single report can be provided for both sets of tasks (i.e. stress and load tests).

Part 5: Security Testing (12%; due: August 1, 2022)

The goal of this phase is to conduct threats and vulnerabilities scans for the application. Different tools will be used to cover different types of security tests. The following tasks must be performed:

1. Construct a threat/vulnerability matrix based on the template presented in class. Start by listing the high-level features of the application. Then identify threats of each feature and possible mitigation solutions and describe how they can be verified. Finally build the table listing the feature, threat name, brief description, mitigation solutions, verification cases and status.
2. Analyze the **SonarQube** reported issues in order to identify potential vulnerabilities or weaknesses in the application's source code. Audit the scan result by analyzing every vulnerability and security hotspot issue (bugs and code smells can be ignored) reported by the tool considering the threat model (i.e., whether they are relevant or applicable) and/or by trying to exploit them.
    a. First, analyze whether each issue is either a false alarm or a real vulnerability and mark it accordingly – note that exploitable Security Hotspot issues will be turned into vulnerabilities when set to "Detect". To help analysis, click on the ellipsis icon to the right of the issue's title to bring the details panel. Read the information there thoroughly to learn more about the specificity of each issue type. This is especially important when dealing with possible security issues since analyzing them is a complex task.
    b. Then, for the real issues, update each issue type and severity according to the Impact/Likelihood matrix. This is important because the tool might misclassify some issues.
    c. Add the appropriate commentary to each issue to describe the reasoning behind each action performed.

3. Use the **Zed Attack Proxy (ZAP)** tool to pentest the application and uncover vulnerabilities not previously identified. An adequate pentest consists of automated scans interwoven with proxied explorations and directed attacks.
4. Fix all high severity (that is, high, critical and blocker) vulnerabilities identified in steps 2 and 3. Lower severity vulnerabilities do not need to be fixed. Do not forget to follow the same protocol used in Part 3 (GitHub flow). Note once again that some issues might be misclassified by the tools, so classify every issue according to its impact and likelihood.
5. After fixing the high severity issue, re-scan the updated version of the application with ZAP to make sure the vulnerabilities were properly fixed, and that no other vulnerability was introduced.
6. After merging the fixes to the master branch, analyze the updated SonarQube report to be sure every reported issue was fixed and that no new critical issue was reported. Mark issues as "fixed" if required.
7. Based on the above testing activities/outcome, update the Threat/Vulnerability matrix, by updating the test coverage/status.
8. Before delivery, release a new version of the application as done in Part 3.

The deliverable of this phase is:

1. The release (tag) available on the group's repository containing the updated code.
2. The actions and commentary available at the group's SonarQube project.
3. A zip file comprising:
    i. The ZAP session(s) containing all the scans performed.
    ii. A report containing:
        1. Screenshots of some reports generated by SonarQube showing changes in the number, type and/or severity of issues through time.
        2. A summary of the pentest process with screenshots and descriptions of scans and tests performed.
        3. A description of the highest severity vulnerabilities found by ZAP.
        4. A short comparison of vulnerabilities found by the two tools and an explanation of why some of these are different and what role each has, if any, in a thorough security evaluation of an application.
        5. The Threat/Vulnerabilities matrix.
        6. A small paragraph commenting on what other tools and/or techniques could be used to further test the security of the application.