**Technical Project Report 2**

**AUTOMATED SOFTWARE TESTING**

**1. A Discussion about the Case:**

Star Fleet is a web design company. This company primarily works on designing web pages for other companies. All the web pages are stored on the local GitLab server. There is one Git repository per website. In addition, they have one web server per site which is running on their own Open Stack Cloud. [1]

To modify the webpage for the customer, the existing workflow is as follows, in Fig 1:

1. To check the latest version from the GitLab server.
2. Make the required changes to the HTML pages.
3. Use Git Tag feature to add a new tag for the new version when all the changes are made. They don’t use any typical semantic versioning mechanism but they use V1, V2,V3 etc
4. Commit the changes and push it to the GitLab server.
5. Log in to the particular web-server which runs the site in production and go to the folder serving the HTML files. Pull the latest tag.

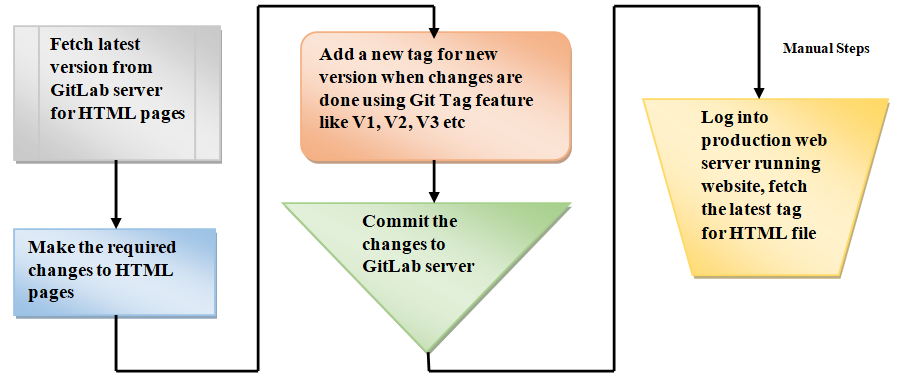


Figure 1 Current Process flow in StarFleet design company

The above method of performing the changes to the web site is best suitable only for the developers. Since they have the local copy of the versions of the site on their laptops, they can login from anywhere, be it from home or even on the road. [1]

The real problem comes when the testing for web site changes needs to be performed. That is due to multiple situations in the past recently where there were issues with the HTML code of web site and therefore the website was not displaying the correct information. Due to these mistakes, the customers are getting irritated.

The current workflow of making the changes to the website is a bit manual. This problem case describes the common problems and challenges that the developers face in the traditional software development life cycle while developing, testing and delivering the software product to the end customers.

In a typical software development process cycle, each developer receives the copy of the code from the central repository. They work on the latest stable version of the code which is the starting point to work on the new features to be developed or to resolve the bug issues. The developer progresses on the work individually or in a team. They add or modify the exiting functionalities in the code in order to meet the user requirements and complete the respective tasks. At the same time, the other developers and the teams also work on their assigned tasks by making the necessary changes. [4]

If we try to take a look back, the immediate observation is that all the developers are trying to perform the changes on the code changing the context, which in turn forms the basis for the other developers to further continue working on the source code. Eventually, when the teams complete their tasks, they copy their code to the central repository. There are two possibilities in this platform which can take place.[4]

1. *The code in the central repository is not changed.*

The source code in the central repository is the same as the initial code copy. In this case system is unchanged and things are easy. The expectations about what we think of the system to behave in a certain way would still hold true. This would happen if a person is the only developer working on the code changes and has finished the work before others in the team. This is an ideal scenario and things would look alright. The system which was designed, developed and tested can be delivered to the users without requiring additional changes.[4]

1. *The code in the central repository has changed.*

Another possible scenario which can take place is that the system on which the developer was working on has got changed, and the developer finds out about this when trying to copy the code to the central repository. The changes may or may not be in conflict with the ones the developer has made. If there are conflicts with the changes then those needs to be resolved and fixed before delivering the code to the users. In such situations, the things can get complicated and tricky to handle. [4]

There are various conflicts that can occur when integrating the code. For instance, the implementation details could have changed, the API’s could have been changed, or complete change system or application could behave in a different way. Along with these there could also be sometimes issue with versions of frameworks, libraries, databases which could lead to potential conflicts. After fixing the conflicts, one also needs to perform all the testing once the code is complied to ensure that the changes done are successful.[4]

Thus, the amount of work and time could increase to solve the problem compared to the initial estimations. There are other factors which also contribute to the complex integrations that too many changes in the system which also directly depends on the number of members in the project, the more the members, likelihood of large changes in the system.[4]

The solution to the above problem is to integrate continuously with automated testing. The developers should integrate the partially completed work back into the central repository often in short periods of time. Also, it should be ensured that the code should not break the system once it is integrated into the main repository. This would mean that developers would spend most of their time in testing the code so that it doesn’t fail when integrated in the main repository. Therefore it is also essential to also have automated testing for continuous integration. [4]

Automated Testing removes the work of doing manual which is repetitive in nature from the developers. This ensures the test process to be much faster compared to the human effort. The automated testing should occur once there any changed are encountered in the source code. This also ensures that there is continuous integration, continuous delivery and continuous deployments of the software changes.

Jenkins is one of the many Test automation tools which can monitor and keep track of any changes done to the source code repository and can run the tests on changes. It is an open source and free continuous integration server which can instrument a series of actions continuous integration process in an automated way. It can monitor and continuously test the builds of the project and display the errors early on in the development phase. This saves a lot of effort and time, and allows to take appropriate corrective measures in the initial stages of the development phase. Through Jenkins, software development process can be faster as Jenkins can automate the build and test process quickly. [3]

**2. A technical design**

Docker, Git, scripting along with Jenkins automation pipeline would be used to design the solution to the problem case discussed in the earlier section 1.

The Jenkins automation architecture integrated with Docker, Git and Micorosft Teams is shown in Fig 2 below. Once the changes are pushed to GitLab, the Jenkins pipeline would automatically get the code through web hook, it would create a docker container to build, and then perform tests. If the testing is successful then Jenkins will build and deploy the code to the production server. It would send the docker image to the docker hub registry. It would send the status of whole process through the Microsoft Teams channel.

For this discussion case, there is a need to automate the testing and deployment process once the changes are done and pushed back to GitLab. Through Jenkins, a Continuous Integration Pipeline would be established which can monitor, build, test and deploy the changes automatically.

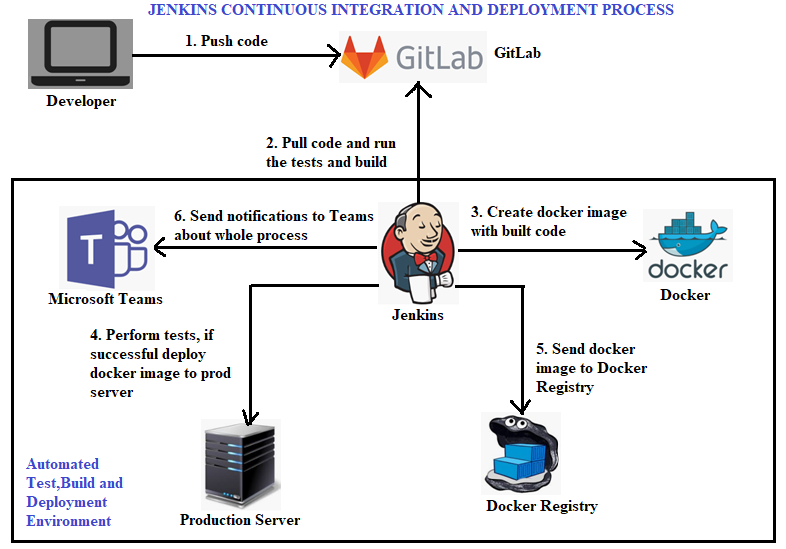


Figure 2 Jenkins automation: Continuous Integration and Deployment Process, Source [5]

A GitLab source code repository would be maintained. This would contain the index.html file for the website changes, a html\_test.sh script file to test the HTML page and a Dockerfile with the docker commands. An integration would be done between GitLab and Jenkins via webhook configuration. Also Jenkins Pipeline project would be created which would contain the Jenkins Pipeline script. The script would contain stages for Git pull, create BuildTest docker container, test the website changes and deploy to production webserver/clear the container

The Work Flow of the technical design would be as follows as illustrated in Fig 3:

1. Firstly, the developer would commit and push the changes for the HTML page to the source code repository i.e. Gitlab server.
2. The Jenkins server would check the repository continuously for any source code changes at regular time intervals. Once the commit and push occurs to the GitLab, the Jenkins server detects the changes in the source code repository and Jenkins Pipeline job is triggered.
3. Jenkins pipeline would pull those changes and would start preparing for a new build.
4. The Jenkins server would create a docker container for build-test, and would deploy the code changes in the test server.
5. An external script would be executed which would run the tests on the new HTML code changes.
6. If the testing is successful only then another container would be created for production webserver, the changes would be deployed to this production server. The build-test docker container would be removed
7. If the testing is not successful then the changes would not be deployed to the production server. The build-test docker container would be removed.
8. The notifications about the success as well as failure would be sent to Microsoft Teams channel. The build status can also be sent through Emails and Slack.

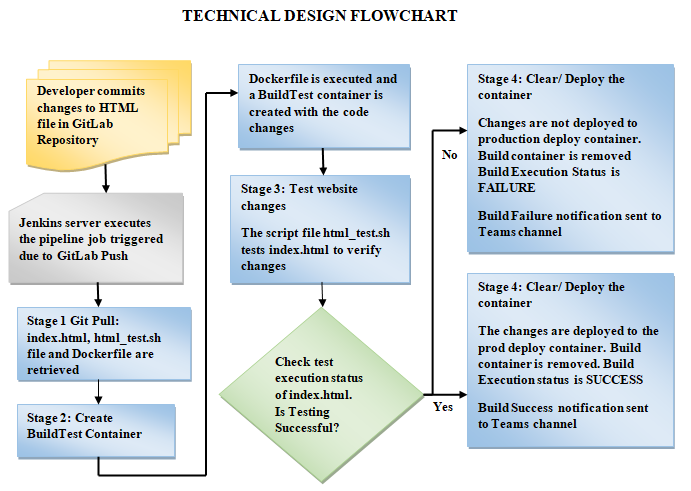


Figure 3 Technical Design Flowchart for Automated Software Testing

**3. Pilot Implementation**

The implementation for this problem case assumes that the code changes are maintained in the GitLab server repository. A GitLab repository is created with name StarFleet\_Automated\_Software\_Testing and contains the index.html file, Dockerfile and html\_test.sh script file for test execution. A Jenkins pipeline project is created with name StarFleet Automated Test Pipeline in Jenkins server. The Jenkins server is accessible via link http://localhost:9000 or http://IP address: 9000

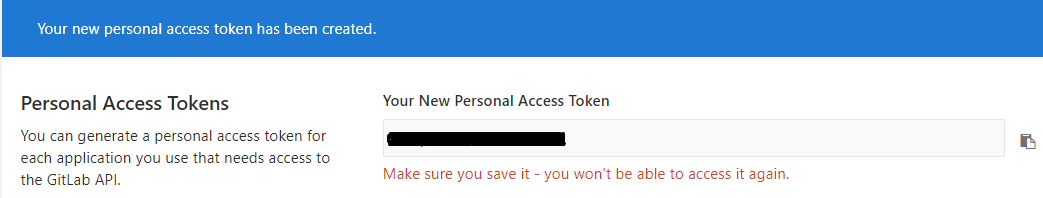
Below are the steps for implementation of an automated testing pipeline through Jenkins, Docker, GitLab and Microsoft Teams, based on reference [2]

***Step 1: Establishing a Jenkins web hook for GitLab***

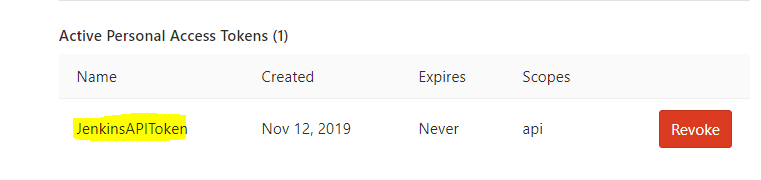
The initial step for Jenkins is that it should automatically trigger a build when there are any commit or push changes made to the Git Lab source code repository. For this, a web hook for GitLab needs to be established.

*GitLab: Generate Personal Access Token for Jenkins*

In GitLab, under the User Profile settings, in Personal Access Tokens, a token is created for Jenkins Application by giving the name of the token and clicking on the “Create personal access token” button. The token content is copied and saved in a text file as it is displayed only once.

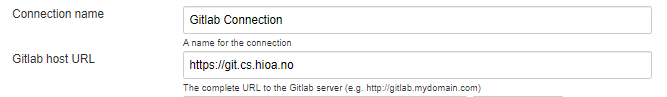


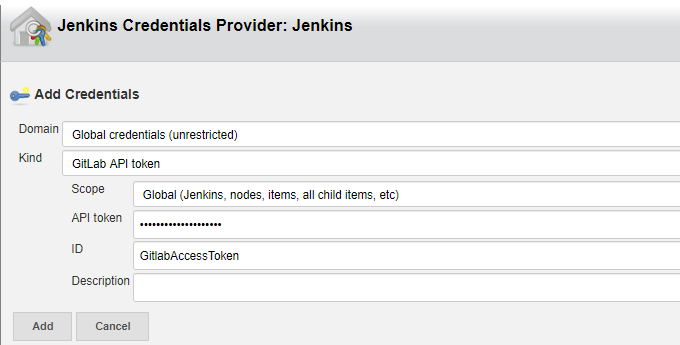
After successful creation of the Personal Access Token, it is displayed under the Active Personal Access Tokens as below



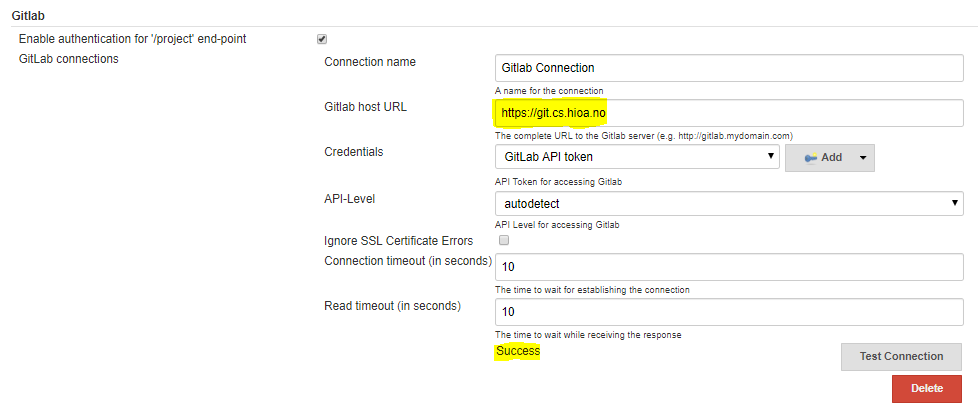
*Manage Jenkins Configuration: Configure GitLab options*

In Jenkins, go to Jenkins>Manage Jenkins>Configuration, go to the GitLab section, enable the check box for Gitlab connects. Provide the Connection name and Gitlab host URL, in this case it would be <https://git.cs.hioa.no>.



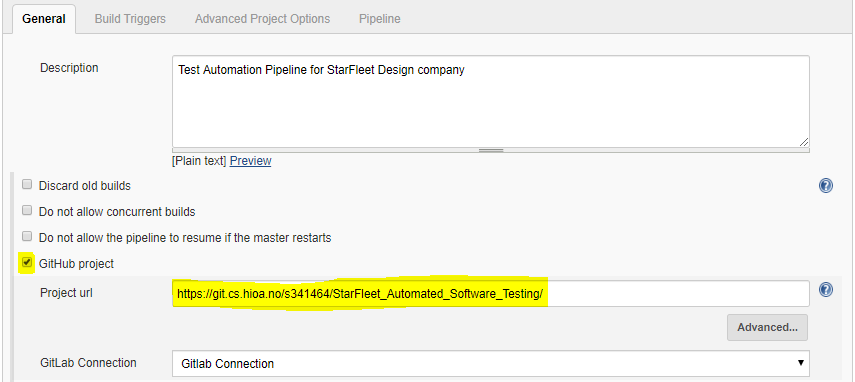
Add the credentials as in the below screenshot by providing the API access token obtained from the above step.

After adding the API access token for accessing GitLab in the Credentials field, the Test Connection button is clicked and to verify that Success message is displayed as shown in the screenshot below.

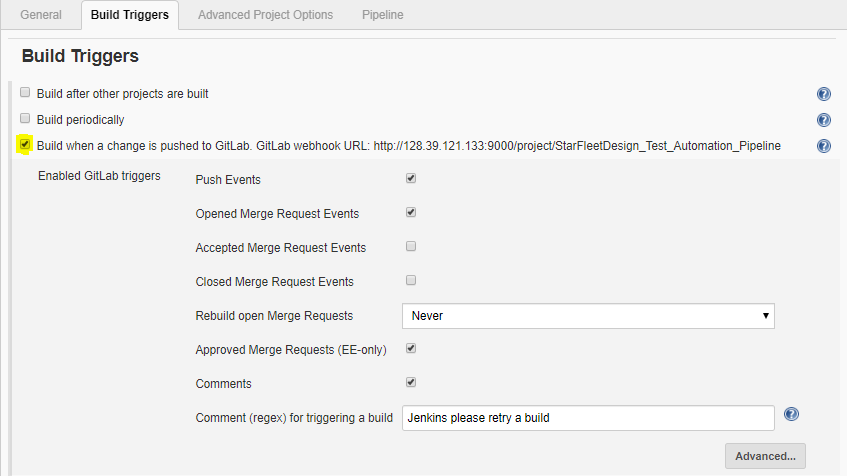


*Jenkins Pipeline Project: Build Trigger settings and Secret Token Generator*

In Jenkins Pipeline project, go to the Configure option > General Tab, select the check box for GitHub project as below and provide the URL of GitLab source code repository project folder.

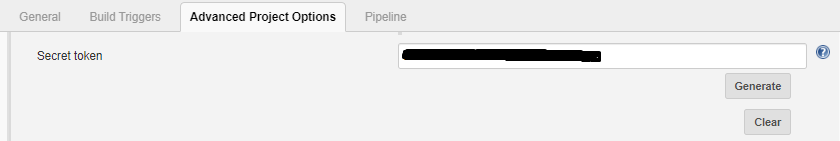


Then, under Configure > Build Triggers section, select the tick box option for “Build when a change is pushed to GitLab. GitLab webhook URL …” as in the below screenshot.



Then, navigate to advanced setting by clicking on the Advanced… button in Build Triggers.

In the section Secret token, click on the Generate button to generate a secret token as below.

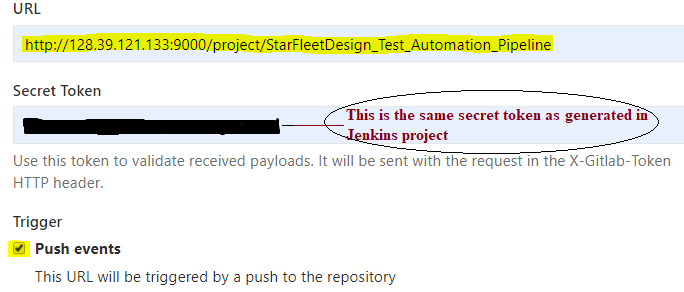


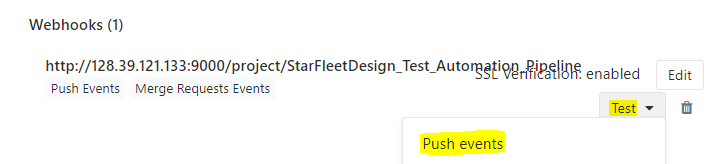
*GitLab Project Integrations: Add and Test Web hook*

In the GitLab, navigate to the Project folder and go to Project>Settings>Integrations.

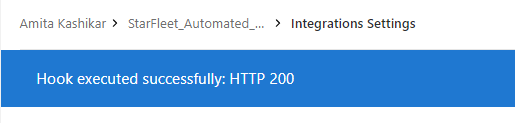
Provide the Jenkins pipeline project URL in the URL field as in the below screenshot.

Also, the secret token is given in the Secret Token field which is the same secret token generated in the previous step in Jenkins project configure Advanced options. Enable the Push events and Merge Request events check boxes and click on Add webhook button.



Once the web hook is added, click on the Test >Push events 

If the Web hook is configured successfully, the message “Hook executed successfully: HTTP 200” gets displayed as in the below screenshot.



***Step 2: Jenkins Pipeline Script***

Once there are any changes pushed or merged in the GitLab source code repository, the Jenkins Pipeline script would automatically trigger.

A Jenkins pipeline script is written under the Pipeline section of the Jenkins Pipeline project to automate the build, test and deploy stages of the web site changes for the company. To support the syntax and pipeline script code generation, there is an option Pipeline Syntax which helps to effectively Generate Pipeline Script. The Snippet Generator was used to generate the Jenkins Pipeline Script. For demonstration, the port numbers 32800 would be used (as a test server) for build test container for testing the website changes and the port number 32798 would be used (as a production webserver) to deploy the website changes. The port numbers are opened in the Open Stack instance in the Manage Security Rules in Open Stack instance

The Jenkins pipeline script is illustrated and explained in parts as below.

1. *Jenkins Pipeline Script: Stage ‘****git Pull****’*

**node{**

**stage(‘git Pull’)**

**{**

**git ‘https://git.cs.hioa.no/s341464/StarFleet\_Automated\_Software\_Testing’**

**}**

In the beginning of the Pipeline script as in the above screenshot, a node is initiated and within it a stage named **‘**git Pull**’** is defined. The command git ‘https://git.cs.hioa.no/s341464/StarFleet\_Automated\_Software\_Testing’ pulls the entire git repository link mentioned and the contents are made available in the Jenkins Project Working Space.

1. *Jenkins Pipeline Script: Stage ‘****Create BuildTest Docker Container****’*

**stage('Create BuildTest Docker container')**

**{**

**sh label: '', script: ''' docker build -t="buildtest\_container" .**

**docker run -d --name buildtest\_container -p 32800:80 acit4410web:v1 '''**

**}**

In this stage ‘Create BuildTest Docker container’, shell command for docker build is executed present in the root directory. This executes the Dockerfile which is now contained in the Work Space of the Jenkins pipeline project.

The below set of command in the Dockerfile get executed

**FROM ubuntu:18.04**

**MAINTAINER s341464@oslomet.no**

**RUN apt-get update**

**RUN apt-get install -y apache2**

**RUN apt-get install -y git**

**RUN git clone https://git.cs.hioa.no/s341464/StarFleet\_Automated\_Software\_Testing**

**COPY index.html /var/www/html/index.html**

**EXPOSE 80**

**CMD /usr/sbin/apache2ctl -D FOREGROUND**

The second command in the pipeline script runs a container with name “buildtest\_container” on port number 32797

1. *Jenkins Pipeline Script: Stage ‘****Test the website changes****’*

**stage('Test the website changes')**

**{ sh label: '', script: '''pwd**

**chmod +x html\_test.sh '''**

**sh "ls -la ${pwd()}"**

**def rc= sh script: '/bin/bash html\_test.sh', returnStatus: true**

**}**

In this stage ‘Test the website changes’, the shell script named “html\_test.sh” gets executed and the based on the check conditions in the shell script to determine the correctness of the website html code, the shell script returns with a execution status code. The html\_test.sh file contents are as follows:

**1 #! /bin/bash**

**2 data=$(curl -s http://128.39.121.133:32800/index.html | grep -w -wc "Welcome to the website")**

**3 if [ "$data" -eq 0 ]; then**

**4 echo "The test execution was not successful"**

**5 exit 1**

**6 else**

**7 echo "The test execution was successful"**

**8 exit 0**

**9 fi**

In the above shell script, the line number 2, gets the index.html file content which is serving on the port number 32800. With the grep –w –wc “Welcome to the website”, it tries to match the exact string in the index.html file and returns number of lines which matches the string text in “data” variable.

At line 3, it checks the value in “data” variable, if it is 0, that means it did not find the exact string, it prints a failure message at line 4 and exits with return value as 1 at line 5. Else at line 7 it prints a success message and line 8 exits the script with return value as 0

1. *Jenkins Pipeline Script: Stage ‘Clear container / Deploy code’*

In this Pipeline Script stage ‘Create Build Test Docker Container’.

The stage is divided into two parts based on the return value obtained from the execution of html\_test.sh script for testing the website changes.

**stage ('Clear container/ Deploy code')**

**{ def rd= sh script: '/bin/bash html\_test.sh', returnStatus: true**

**if (rd!=0)**

**{ sh label: '', script: '''**

**docker stop buildtest\_container**

**docker rm buildtest\_container '''**

**currentBuild.result = 'FAILURE’**

**}**

The script checks the return value of the html\_test.sh execution.

If the return value is not equal to 0 that means the test execution of the website was not successful and there were issues encountered. In this case, the buildtest\_container services are stopped and removed.

Also with the statement currentBuild.result = ‘FAILURE’, the build execution status of the Jenkins Pipeline is set to FAILURE.

**else**

**{ sh label: '', script: '''**

**docker stop buildtest\_container**

**docker rm buildtest\_container**

**docker stop deploy\_container**

**docker rm deploy\_container**

**docker build -t="deploy\_container" .**

**docker run -d --name deploy\_container -p 32798:80 acit4410web:latest '''**

**} } }**

In the else part, if the return value of the html\_test.sh is equal to 0 , it means that the test execution of the website changes were successful and in this case, stops and removes the buildtest\_container, also stops and removes any previous deploy\_container.

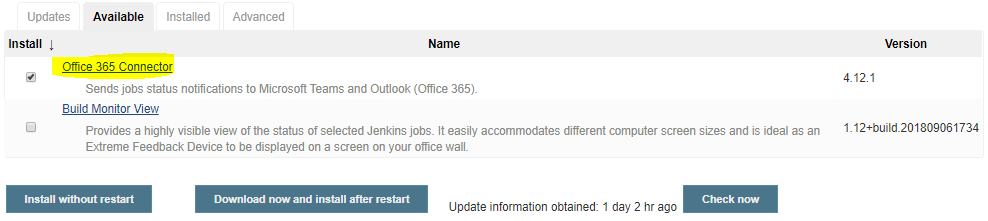
The docker build and runs the deploy\_container on port 32798 for with the latest successful website changes available in the index.html file.

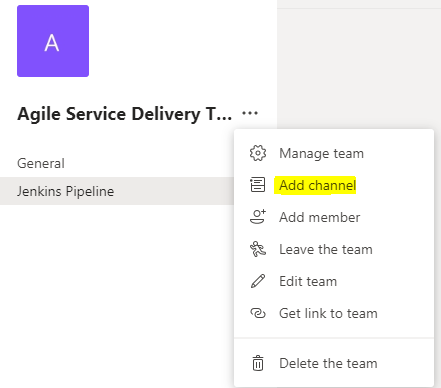
***Step 3: Setting up Microsoft Teams Channel for sending notifications regarding the Build status***

For sending the notifications regarding the Success or failure status of the Jenkins Automated pipeline build, there are multiples of ways of communicating such as through Slack, Emails or Microsoft Teams.

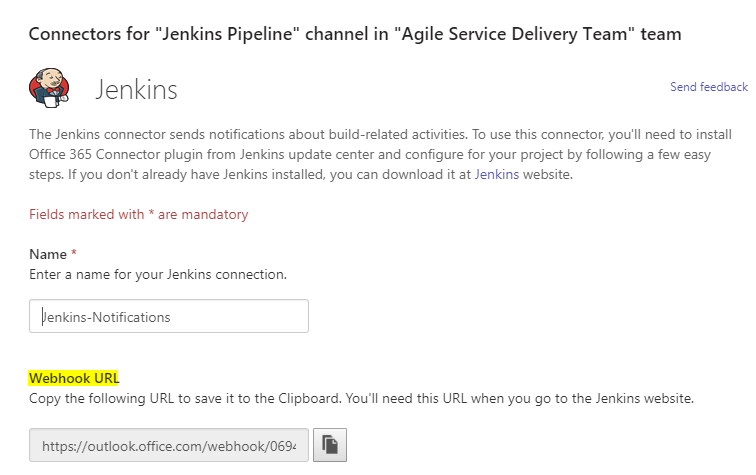
Emails can be sent to the developers who committed the push changes to the GitLab notifying about the SUCCESS or FAILURE status. The body of email can contain details such as the Build number, Build time, error messages (if any), whether or not the changes are deployed to the production server. The Jenkins Plugin “Email Extension” can be used and configured for setting up the alert via emails.

For sending the notifications automatically to Microsoft Teams channel, a Jenkins Plugin called “Office 365 Connector” is downloaded from the Manage Jenkins> Manage Plugin as in the below screenshot.

****

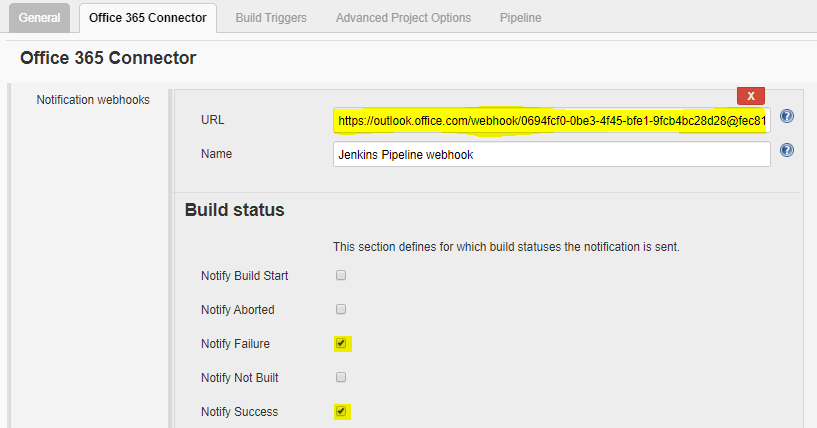
In Microsoft Teams, a Team named “Agile Service Delivery Teams” is created. In the Teams, a channel is added by clicking on the Add Channel button as in the below screenshot.****

A connector is added in the Channel, and Jenkins option is selected. A connector for Jenkins is created with the name “Jenkins-Notifications” and clicking the Create button.

****

A webhook URL is generated which is saved in the clipboard.

Then, the pipeline project in Jenkins is opened, navigating to the Office 365 connector. In the Notification webhooks, the webhook URL generated from the above step is given in the URL field as in the below screenshot.

****

The options Notify Failure and Notify Success are checked under the Build Status section and the settings are applied and saved. Once the Jenkins Pipeline project build is completed, the notifications would be received in the Microsoft Teams Channel.

**4. An evaluation**

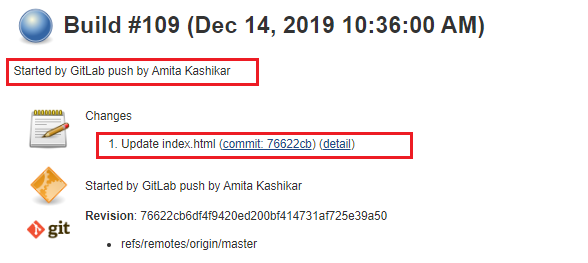
For evaluation of the pilot implementation, the below features were tested to validate whether the implementation of the prototype design is successful.

*Use Case Scenario 1 Push and Commit the changes in the index.html file to the GitLab repository with the correct website content and no errors in the html file.*

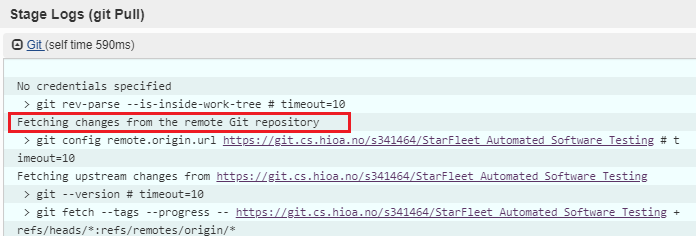
The index.html contents are as follows



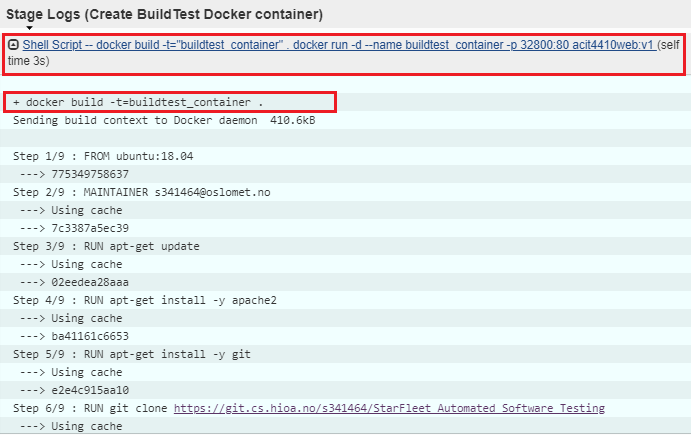
The index.html file changes are updated and committed to GitLab. Once the changes are pushed, the Jenkins Pipeline job automatically triggers the Build as in the below screenshot which is viewed in the Console Output.

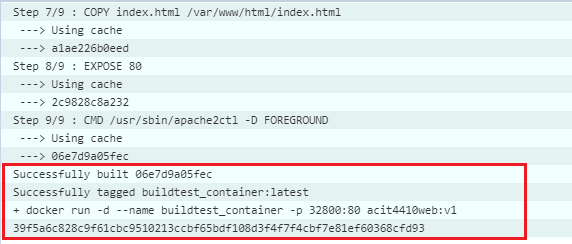


Stage 1 git Pull Logs details

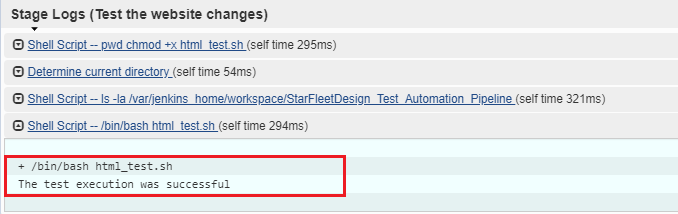


As highlighted, the pipeline executed pulls the changes from the Git repository successfully.

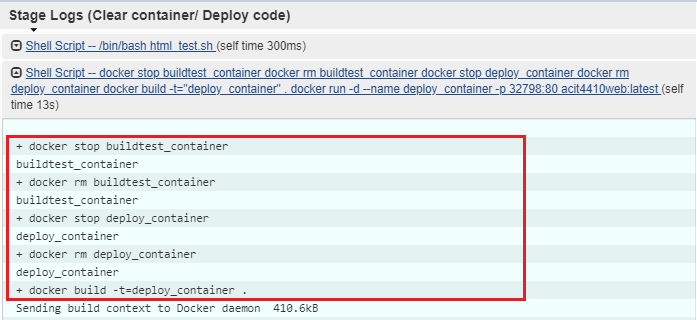


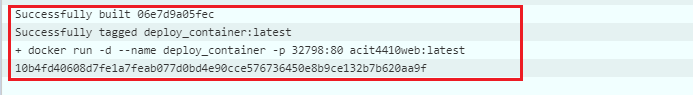


In the stage 2 Create BuildTest container, a docker container for build and test is created successfully with the index.html deployed on port 32800



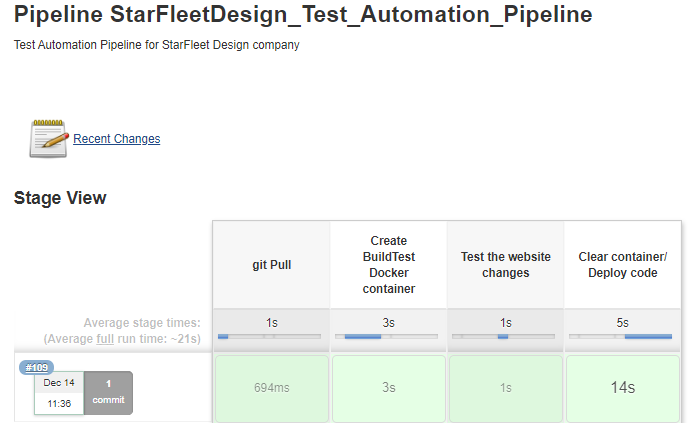
The test execution is successful after script performed the test on index.html



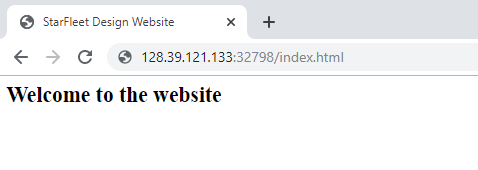


The changes are deployed to the production server through docker container created on port number 32798. The build\_test container services are stopped and removed.

The stage view for the Jenkins Pipeline project is displayed as below with an overview of execution of all the stages.

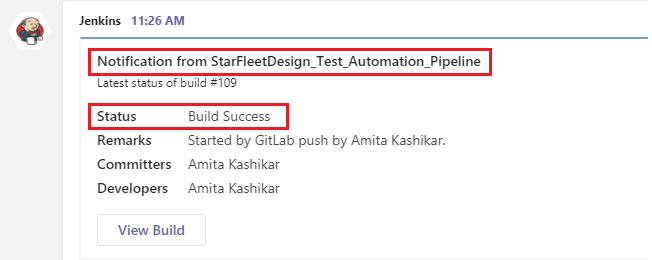


Website Changes



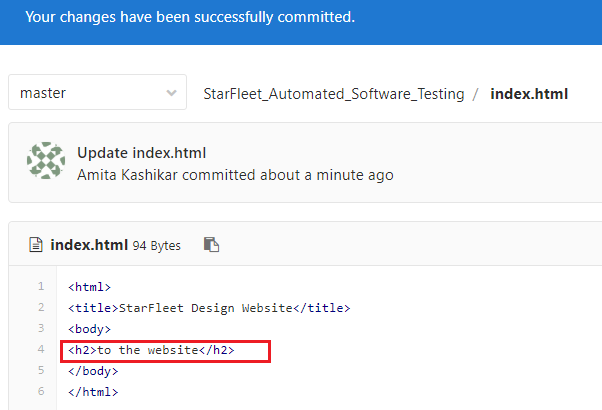
The changes are successfully deployed on the prod web server as in the above screenshot.

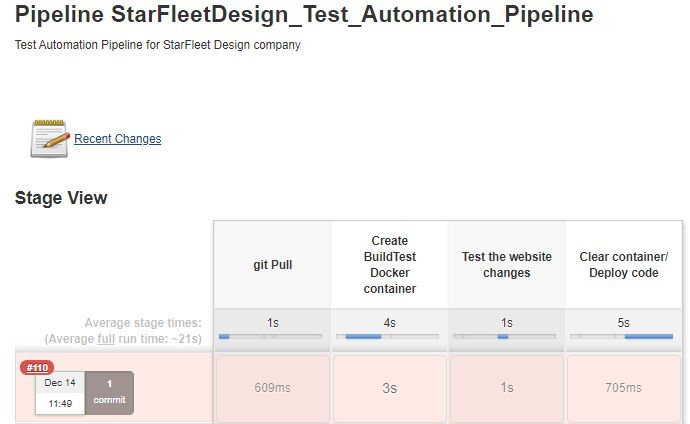
A Notification message received in the Microsoft Teams Channel with build Success.

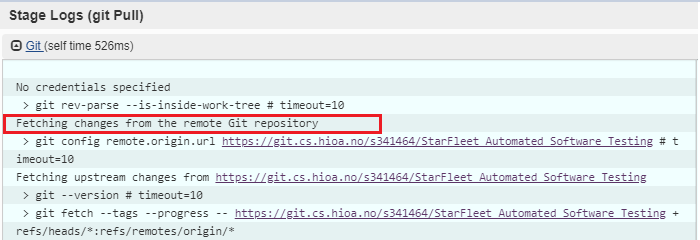


Jenkins Pipeline successfully performed build, automated tests and deployed the code to prod webserver without human intervention.

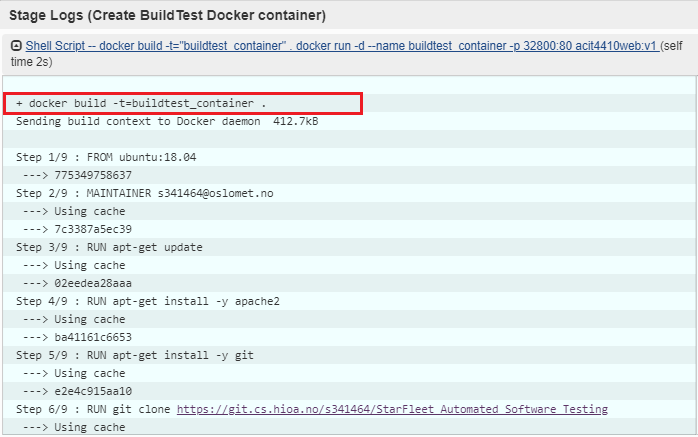
*Use Case Scenario 2 Push and Commit the changes in the index.html file to the GitLab repository planting some wrong content in the html file.*

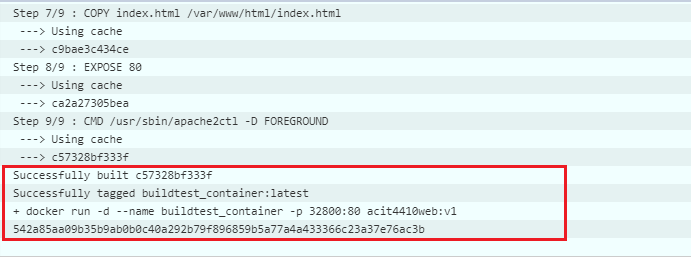




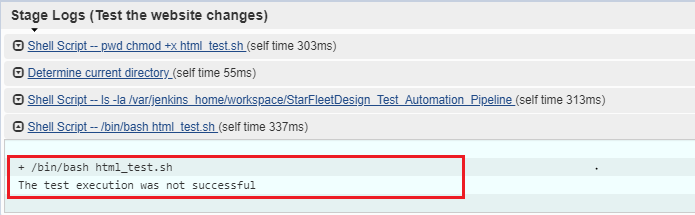


The changes are pulled from the Git repository

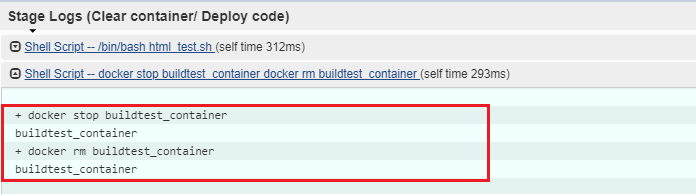




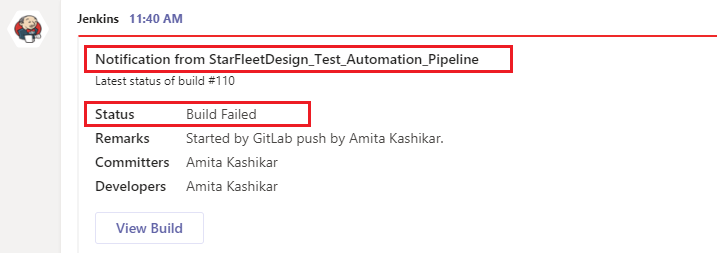
A build\_test docker container is created on port number 32800 with the index.html changes



The test execution was not successful in this case.



The changes are not deployed to prod server. It sets the build status to FAILURE and removes the buildtest\_container.



A notification message regarding the Build Failure is received in Microsoft Teams channel as in the above screenshot.In this case, Jenkins pipeline performed the build and test automatically, and since the test was not successful it did not deploy the changes to the production webserver.

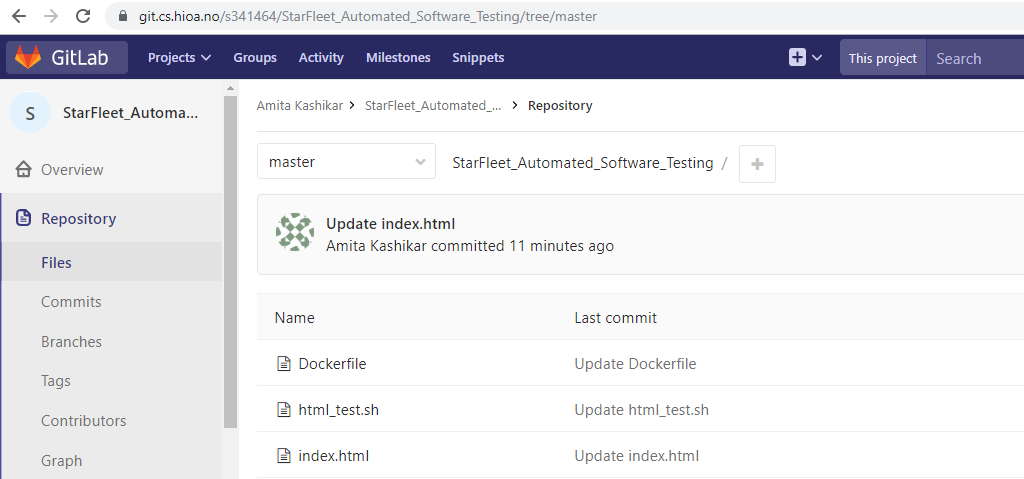
The prototype is implemented successfully through Jenkins. The Jenkins Pipeline sets up a continuous automated test platform reducing the manual steps, required in order to build, test and deploy the changes to the Production. With this approach, the effort and time are saved and also there are less chances of errors as the testing is automated. The notifications via Teams channel are also sent regarding the success and failure of the Build status.

The company can implement the continuous automated testing process through Jenkins Pipeline with the above implementation steps. The design can be enhanced further by implementing the logic for testing multiple webpages, performing tests and deploying them on production. A more complex webpage can also be tested by adding more conditions in the test script. The notifications can also be sent through various other communication modes such as Emails, Slacks etc. The company should consider implementing the website changes build, test and deployment through Jenkins with additional features as per requirement. This would lead to significant reduction of manual activities in website designing.

**Appendix**

**Configuration Files:**

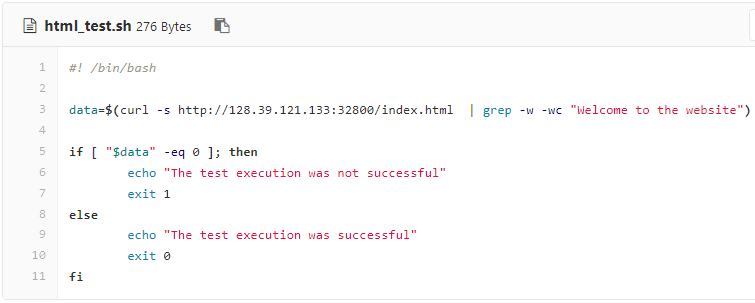
GitLab Source code repository to push and commit the website changes in index.html file



**Dockerfile**



**html\_test.sh file - Test execution script to test HTML file**



**index.html file**



**Jenkins Pipeline Script**

**node**

**{**

**stage('git Pull')**

**{**

**git 'https://git.cs.hioa.no/s341464/StarFleet\_Automated\_Software\_Testing'**

**}**

**stage('Create BuildTest Docker container')**

**{**

**sh label: '', script: ''' docker build -t="buildtest\_container" .**

**docker run -d --name buildtest\_container -p 32800:80 acit4410web:v1 '''**

**}**

**stage('Test the website changes')**

**{**

**sh label: '', script: '''pwd**

**chmod +x html\_test.sh '''**

**sh "ls -la ${pwd()}"**

**def rc= sh script: '/bin/bash html\_test.sh', returnStatus: true**

**}**

**stage ('Clear container/ Deploy code')**

**{**

**def rd= sh script: '/bin/bash html\_test.sh', returnStatus: true**

**if (rd!=0)**

**{**

**sh label: '', script: '''**

**docker stop buildtest\_container**

**docker rm buildtest\_container '''**

**currentBuild.result= 'FAILURE'**

**}**

**else**

**{**

**sh label: '', script: '''**

**docker stop buildtest\_container**

**docker rm buildtest\_container**

**docker stop deploy\_container**

**docker rm deploy\_container**

**docker build -t="deploy\_container" .**

**docker run -d --name deploy\_container -p 32798:80 acit4410web:latest '''**

**} } }**

**References to Sources:**

[1] Automated Software Testing <https://oslomet.instructure.com/courses/16478/pages/automated-software-testing?module_item_id=119336>

## [2] ACIT4410 CI with Jenkins\_Lect.pdf <https://oslomet.instructure.com/courses/16478/files/763333?module_item_id=119123>

[3] <https://www.guru99.com/jenkin-continuous-integration.html>

[4] <https://semaphoreci.com/community/tutorials/continuous-integration>

[5] <https://medium.com/@ahmetatalay/continous-deployment-via-gitlab-jenkins-docker-and-slack-5d08836d01e0>