13th September

Build server

Operational Concept Document

Komal Gujarathi (211778351)
Guided by – Jim fawcett

Table of Contents

1. Executive Summary

This document represents the Operational Concept of the Build Server. It explains the uses and design of a build server and the various critical issues one can encounter while implementing build server. The Operational Concept Document (OCD) also attempts to propose solutions to the listed critical issues. Finally, it describes the code prototype, that helps demonstrate the working of the build server.

A Build Server is a distinct concept to a Continuous Integration Server. It is an automated tool that builds test libraries. The main purpose of Build server is to take the load of the developer machines, provide a stable, reproducible environment for builds. The Build Server then builds libraries needed for each test request, and submits the request and libraries to the Test Harness, where they are executed. Each test execution, in the Test Harness, runs a library consisting of test driver and a small set of tested packages, recording pass status, and perhaps logging execution details. Test requests and code are submitted by the Repository to the Build Server.

The system will be implemented in C# using the facilities of the .Net framework class libraries and Visual Studio 2017. The system will be divided into separate modules in which each module performs the module specific tasks. The **Major Modules** of the build server are:

I. BuildManager – This package is the heart of the build server, which coordinates all the tasks of the build server, like xml parser, convertToDll, toolChainSelector, reportGenerator.
II. XMLParser – That takes in the request, parses it and finds out exactly which code files needs to be sent to the test harness for testing.
III. ConvertToDll – That converts the code files into the dynamic link libraries.
IV. ToolChainSelector – This package is mainly responsible for selecting the appropriate tool chain based on the extension of the code files.
V. ReportGenerator – This package will be used to generate the build reports based on the build reports we get by building the files.
VI. Test Executive – This is the main entry of the application. It will demonstrate that all the requirements are met.

In implementing this system, **few important Issues** are identified and their potential solutions are recommended in the sections below. Some of the critical issues are:

i. Long Builds - The more test files you add to your software, the more lines of code you maintain, and the more tests and routines you run as part of your build process- the longer the build will take to run.
ii. Security Issues - Because developers are treated as trusted entities in the build system, a malicious user could, for example, construct a build definition to run arbitrary code that is designed to take control of the server and steal data from Build Server.
iii. Inconsistent input data - Input data in the XML file might contain inconsistent values in many cases like not providing required fields or value type mismatch etc. If these issues are not well - handled application doesn't know what to do in such scenarios.
Demonstrating Requirements - How can we test all the requirements and check if build is happening correctly.

2. Introduction

The main purpose of Build Server is to take the load of the developer machines, provide a stable, reproducible environment for builds. They focus the tasks associated with builds. This includes updating the build tag, creating any distribution packaging, running automated tests, creating and distributing build reports. Automation is the key.

Consider an Example. When we are working in teams, we have different developers working in teams and they develop the code and they keep working on and changing or enhancing the code. So as soon as the developer makes the changes in the code, he will go into the system and commit the code into some shared repository. Now what happens, let's suppose through the entire day, there are many developers who are changing the code and checking in the code in the shared repository, at the end of the day, when you have the build, let's suppose there is some bug introduced in any of the code and thus the build fails, now it will be very difficult for us to identify for us what exact code failed the build and at what point this bug was introduced in the code. then you have to go back and check in all the code that was created that day and there will be a lot of confusion. This is where the build server comes into picture.

2.1. Application Obligations

The main responsibility of this application is to build the test libraries by selecting the appropriate tool chain, which can further be passed to the test harness to automate the process of testing.

Primary Obligations of the system would be -

i. Accept test requests from repository.
ii. Parse the request to see how each test is configured.
iii. Create temp directory.
iv. Accept files.
v. Parse test request to select correct tool chain, e.g. C# or C++.
vi. Attempt build.
vii. Create build logs and build report and send it to the repository.
viii. If build failed, send notification to repository.
ix. If build succeeded, send libraries to test harness.
x. Command test harness to execute tests.
xi. Remove temp directory.

2.2. Organizing Principles

The organizing principles are to perform the primary functionalities of the Build Server System usnig XMLParser, convertToDll, ToolChainSelector, ReportGenerator modules. BuildManager will coordinate all the tasks of the build server, which will be called by the testExecutive to check the functioning of the entire system.

2.3. Key Architectural Ideas

The Key Idea in Build Server System is to automate the build of test libraries. In this application, we will be using C# language with .NET framework 6.0 and Visual Studio 2017. MS Build is the build platform that enables all build activity in the visual studio world. By invoking msbuild.exe on your project or solution file, you can orchestrate and build products in environments even if Visual Studio isn't installed. The Build Server Automation thus reduces manual labor by developers, ensures builds are consistent and complete, and improves product quality.

3. Actors

Build Server is used to automate the build process. Following types of the users can interact with the first version of Build Server System -

I. Code Developers

Code developers will submit the code to be tested and test requests to the Repository, which will later be sent to the Build Server for building the code files. They will also get the Build Notifications and see the Build Logs, Reports received from the Build Server.

II. Build Engineer

Build Engineer will use the system to perform builds on the code from multiple clients and will generate the build report.

III. QA Engineer

Quality Analyst will get the code and the test requests from all the developers of the system and will build all the packages and check if there are any build issues after combining the code from multiple developers.

IV. Managers

Managers might be interested in having the build reports and the test reports generated by the application.

V. Test Harness

Test harness runs the tests for the multiple users based on the test requests and the libraries sent from the Build Server.

VI. Teaching Assistants and Instructor

Teaching assistants and instructors will also use the supplication to test and build the sample test files. They are also responsible for identifying the faults in the system, if any and report them to the developers of the system.

4. Uses

Build server was introduced to automate the build process. Today many companies are using the build server for their number of use cases. Few of them are

I. The Build Server Automation, if correctly designed, **reduces manual labor by developers**, ensures builds are **consistent and complete**, and improves product quality. Many companies have started using the build server to automate the process of building and later testing, especially in the big organizations, where the lot of code is generated each day, by lot of developers, which may run well independently, but when run in the integrated environment may cause build failures which would be easy to detect if we are using the automation build server, which would otherwise consume lot of developer's time identifying where exactly the build failed. They coordinate distributed development.

II. Once you have your build server set up, you can **establish a rhythm for your project,** where you are building regularly. When something breaks, you'll know, and quickly. A solid heartbeat from the build server leads to a confident development team.

III. Build server is configured such that it can select the appropriate tool chain. Thus, **it can be used to build any type of solution**, may it be C++ or C# or later it can also be extended to build Java Solutions.

IV. It can be **used by the test harness** to automate the testing process, by taking the code files and the test request from the build server and building appropriate test reports.

V. The **link between developers and QA is simplified**. QA personnel can go to a known location to grab latest, properly versioned builds.

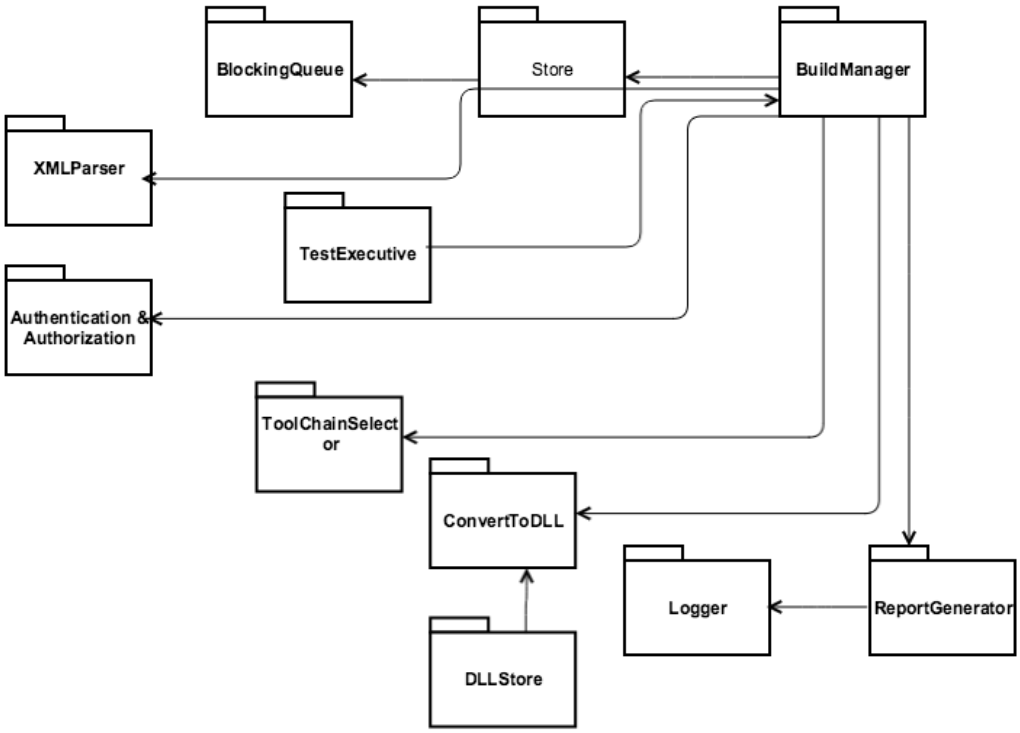
VI. **Product deployment is simplified** -- the developer or QA doesn't have to remember multiple manual steps. It can be easily automated.

VII. While it's a good practice for developers to test everything they check in, sometimes they just don't. Then it's good to have the build server there to catch test errors and let the team know the product is broken.

VIII. You **simplify the developers' workflow** and reduce the chance of mistakes. Your build server can take care of multiple steps such as checking out latest code, having required software installed, etc. There's no chance of a developer having some stray DLLs on their machine that can cause the build to pass or fail seemingly at random.

5. Modular Structure

Build Server is Divided into different packages which will perform different tasks assigned to them. Below is the package diagram of the BuildServer System.



Package Diagram is used here to represent the different layers of the system and how they are dependent on other packages. All the packages are explained below.

I. TestExecutive

TestExecutive package is the entry point of the system. Main method of this system resides in this package. This package will mostly be used by the TAs to check if the build server is functioning properly and if it meets all the requirements. It makes the call to the BuildManager, which handles all the tasks of the system. This package will take all the files and the test request file as an input from the repository and pass it to the BuildManager which will further coordinate all the further tasks of the build server.

II. BuildManager

BuildManager package is the heart of the Build Server System. Its task is to coordinate all the actions that needs to be performed by the Build Server System. It accepts the files and the test requests from the TestExecutive and stores them in the store. The package makes the calls to the below important packages to execute the required tasks.

I. XML Parser – to parse the XML TestRequest that comes from the repository.
II. Authentication&Authorization – to verify if the author is a valid user.
III. ToolChainSelector – to select he appropriate tool chain to build files.
IV. ReportGenerator – to generate the build report after build completes

III. Store

This package holds all the code files and the test request files along with the userAuthentication File, which will store all the information of the users, like username, password, last time he logged in the system, which will further be used for authentication of the user. BuildManager calls this package by sending all the code files and the test request files, one at a time. Store accepts all the Code Files and test request files and stores it in the Store, creates a temporary directory, and stores them all for further use of build manager.

IV. BlockingQueue

Multiple users might want to send the files to the build server for building, the main purpose of the blocking queue is that it queues up all the requests from all the users so that none of the request is remains unattained. This package is internally used by Store to accept the requests from BuildManager, and queue them.

V. XMLParser

This package is responsible to parse the XML Test Request that has all the information like author, password, time, date and all the tests that needs to be performed. XML Parser will parse all this information to find out the information about the author of the test requester, test cases, test driver, etc. Following is the sample xml test request document. Sample Request on next page.

```
<testRequest>
        <author>KomalGujarathi</author>
        <password>MyPwd</password>
         <date>13 Sept 2017</date>
         <time>12.00 pm</time>
        <test>
            <driver>TestDriverExpressions.cs</driver>
            <tested>ArithmaticExpressions.cs</tested>
            <tested>BooleanExpressions.cs</tested>
            <tested>BinaryExpressions.cs</tested>
        </test>
        <test>
            <driver>mytestdriver2.cs</driver>
            <tested>mytestcase1.cs</tested>
            <tested>mytestcase2.cs</tested>
        </test>
</testRequest>
```

VI. Authentication & Authorization

This package will handle verification of users of the system. This package is introduced to ensure more security to the system. Store will have UserAuthentication File, which will have all the information of the users registered with the system, so whenever some client sends the file for testing, the user will be verified for his identity and if the user is a valid user, then his code files will be built, else not. This is to protect the system from malicious users. UserAuthentication file will have the format as below –

```
<users>
    <user>
   <author>Komal Gujarathi </author>
          <password>MyPwd</password>
    <last_used>1.00 am 12 th september</last_used>
    </user>
    <user>
     <author>User 2 </author>
     <password>Pwd2123</password>
   <last_used>1.00 am 15 th september</last_used>
    </user>
</users>
```

Whenever test request arrives, the login credentials of the user will be checked with the credentials in this file, if no match, will show error message on console.

VII. ToolChainSelector

This package is responsible to select the appropriate tool chain to build the files based on the extension of the test files. E.g. There will be an dictionary of key value pairs in this package, which maps the extension with the appropriate compiler to select to compile and build files. E.g. .cs maps to csc, .cpp maps to C++ compiler and so on.

VIII. ConvertToDLL

This package owns the logic to convert the test files into libraries. BuildManager will get the information about the tool chain from the ToolChainSelector Package, and it will pass this information to ConvertToDLLPackage, which will create the libraries, using Process Class provided by the .net framework. This can be well understood from the build server prototype code.

IX. DllStore

It is the container for all the library files, which will later be sent to test harness for testing. ConvertToDll Package uses this package to store all the libraries.

X. Logger

Logger package handles all the functionality of logging. Logging gives you information about what your code does. It catches every action of your application with timestamp and their current and before states which will be stored in one file. This information is useful while debugging of the system and regression testing. Logging makes it much easier and smoother process. It will store all the logs as displayed in the screenshot of the prototype.

XI. ReportGenerator

This package is to generate the build report. This package internally uses the Logger package to parse all the necessary information regarding builds from the logs it generates, analyzes it and store it in the file or display on console as per the user requirements. This report will be sent to the Repository, which can be further viewed by clients. Success and Fail denote the number of test case files successfully built and how many didn't.

Sample Report -

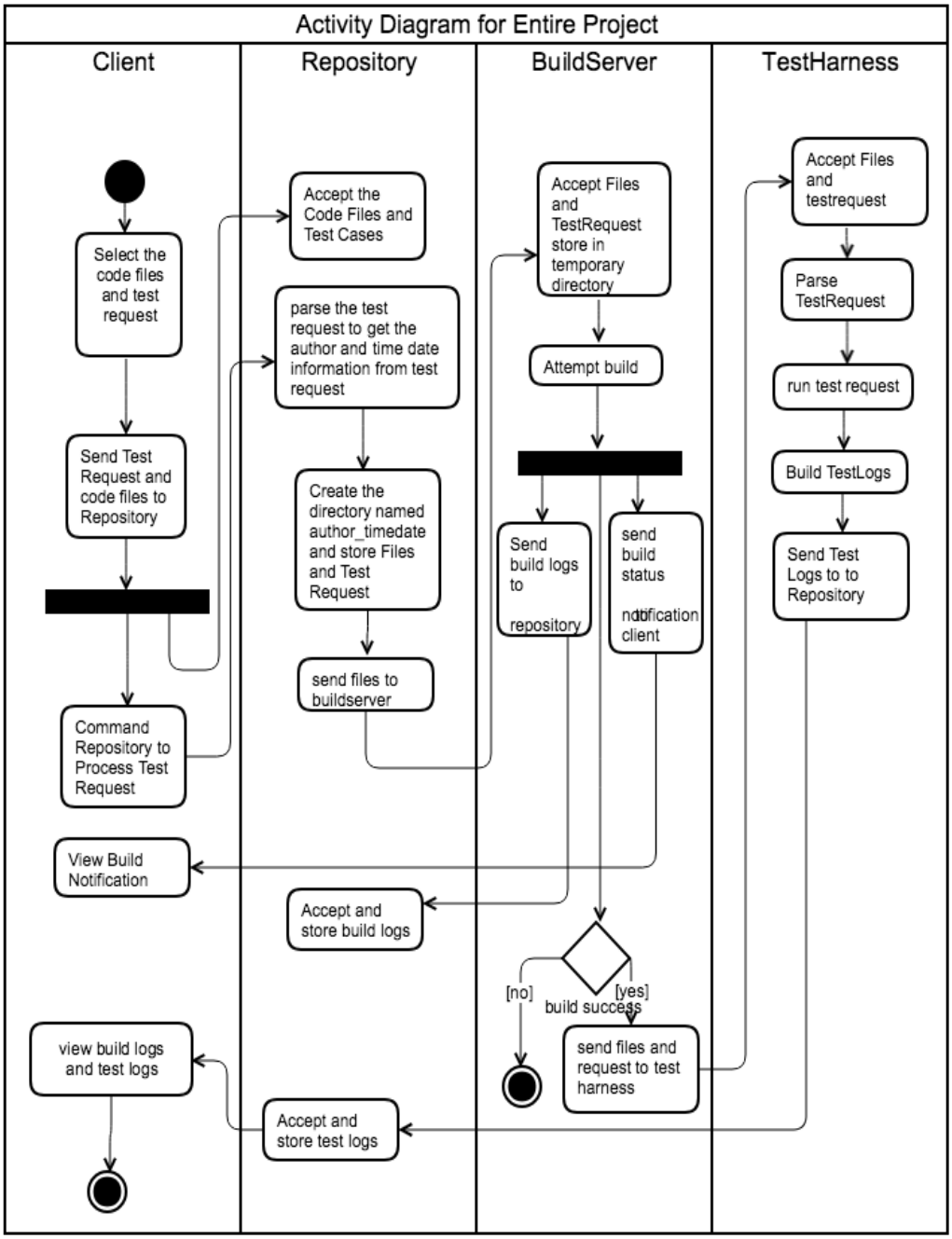
Author	**DateTime**	**Success**	**Fail**	**BuildScore**
Xyz	12 nov 2016 12.00 am	3	1	75%
Xyz	12 nov 2016 12.00 am	2	0	100%
Xyz	12 nov 2016 12.00 am	4	0	100%

In this report, suppose the testRequest is written by Author Xyz, at 12.00 am on 12th November. He writes 3 tests, so there are 3 rows in above table denote 3 tests. In the first test, there are 4 test case files, of which 3 files are successfully built and 1 file fails to build. Thus, build score for first test is ¾ i.e. 75%. Similarly for other tests.

6. Activity Diagram

Activity diagrams are used to illustrate activities of the functionalities of business system. Below are the activity diagrams for whole system.

6.1. High Level Activity Diagram



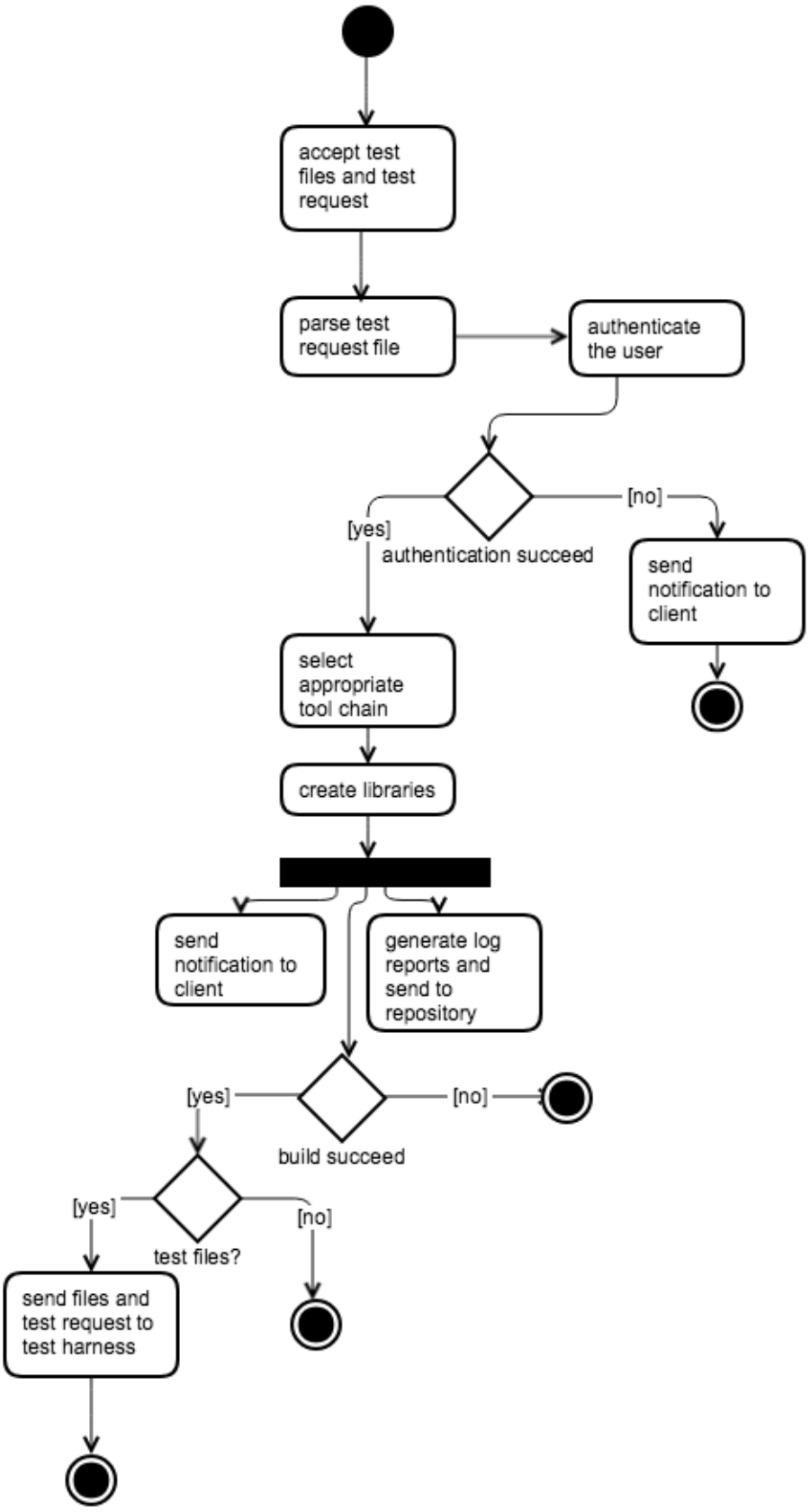
The above activity diagram shows all the functionality of the system, considering **all the important modules of the federation**.

I. Client generates the test request, and uploads the test request files and the code files to the repository.
II. Client also commands the repository to process the test request files.
III. Repository receives the files from the client
IV. Repository parses the test request xml file to get information about the request, and stores the request in the relevant directory or creating new directory if it does not already exist.
V. Repository then sends the files and the xml file to the build server.
VI. Build server accepts the test files and the test request file and stores it in the temporary directory.
VII. Build server attempts to build the files and if succeed, sends the files to test harness for testing and sends the build notification to client. Now if the client just wants to build the files and not test them for correctness, then client must include this information in the test request file, and based on the information, the server will either further pass the files to test harness or stop.
VIII. Build server also sends the build logs to the repository, which can be further used by the client.
IX. Test harness will further accept the files.
X. Test harness will also parse the test request files to know which file is the testdriver file and the test case files.
XI. It will then test the files for their correctness and generate the test logs.
XII. Notification will be sent to the client that the processing has completed, and sends the test logs to the repository.

6.2. Activity Diagram for BuildServer

The below activity diagram shows all the functionality of the **build server**.

I. Build Server first gets the files and the test request file from the repository.
II. Then it parses the test request file and extracts the information like author, password, time, date, test file names, test driver file names.
III. It will then authenticate the user by authenticating the user by trying to match the user credentials with the use authentication file, which stores the information about all the users of the system.
IV. The first decision box is to check if the user is a valid user, if he is a authentic user, appropriate tool chain is selected for the further processing, else the user is notified that the authentication failed.
V. Next, build the test files into libraries and store it for the further sending files to the test harness for testing.
VI. Now, even if the build process fails, or succeeds, display the log files to the repository.
VII. Generate reports as described in section 5 and send it to the repository.
VIII. Send Build Notification to client.


accept test files and test request
parse test request file
authenticate the user
[yes]
authentication succeed
[no]
send notification to client
select appropriate tool chain
create libraries
send notification to client
generate log reports and send to repository
[yes]
build succeed
[no]
[yes]
test files?
[no]
send files and test request to test harness

7. Critical Issues and Analysis

Critical Issues which are experienced by developers and build managers that have critical impact on software quality and release times are

I. **Security Issues** – Because developers are treated as trusted entities in the build system, a malicious user could, for example, construct a build definition to run arbitrary code that is designed to take control of the server and steal data from Build Server.

Solution – Solution to this issue is handled by the Authentication & Authorization package of the system. It contains UserAuthentication file of the system which will declare all the permitted users and their privileges. Every individual who wants to build and test his code files, have to go through the password authentication protocol, thereby protecting the system from malicious users. Before building the files from the user, first he will be authenticated for his username and password (which can be gained from the test request file), against the user id and password stored in the UserAuthentication File that resides in build server. If the user is a verified user, then only his build request will be processed, else system will throw the failure to authenticate message and stop.

II. **Long Builds** - The more components you add to your software, the more lines of code you maintain, and the more tests and routines you run as part of your build process- the longer the build will take to run. This, in turn increases developer build time, context switch, product quality, build bottleneck. If you're a **build engineer** responsible for running 500 builds, you'll feel the pain even if each of them is 10 minutes long (=~83 hours build time without parallelization). While builds may be short, cumulatively they take very long to run. But if you're a **developer or QA engineer** using such as build system, and your build takes only 10 minutes, why should you care?

Solution – The quick fix for long build times can be achieved by running builds parallel on one machine in the multi-threaded environment. Also, other fix can be running only those parts of the build that can be changed. Build speed can also be improved by moving the entire build operation to RAM to eliminate disk I/O. Also, we can precompile the shared dependencies that change less frequently.

III. **Reliability And Work Load** – If you are running large numbers of builds, you'll find you have limited access to build servers during specific time windows, or the servers are often overloaded and builds will take much longer. When you rely on running builds often to get fast feedback and fix bugs, you'll notice that having to "wait in line" for your builds to

run hurts your productivity (particularly if you're practicing Agile), and that you can't move development fast enough because your waiting for a build. Also, there are the chances of losing some of the build requests, because of the system overload.

Solution – We may want to run multiple instances of the test harness on multiple servers. In this case a team of developers can use one test harness for development and a QA team may use another instance of test harness on another system. Having multiple instances also helps in case we have a downtime on one of the servers running the test harness. The users can direct their test cases to alternate instances of test harness in case one server is down. Also, blocking queue can be used to queue up all the requests to be processed, so that none of the requests remains un attained.

IV. **Inconsistent Input Data** - Input data in the XML file might contain inconsistent values in many cases like not providing required fields or value type mismatch etc. If these issues are not well -handled application doesn't know what to do in such scenarios.

Solution - We will design an application specific schema to provide input in particular format. In that case application will easily find which all inputs are inconsistent and write those back to user.

V. **Demonstrating Requirements** - How can we test all the requirements and check if build is happening correctly?

Solution - TestExecutive package will be used to test all the requirements, to check if everything works perfectly. After the build process, whether or not the build is successful, the ReportGenerator package will generate the report in the specified format, which will be stored in Repository, where we can get all the details about the builds.

Impact on Design - It will be effective to provide appropriate test logs and build logs for each test request on the console, or even logs can be stored in the files for further review.

8. Conclusion

In conclusion, BuildServer can be used by every huge organization to automate the build process. It reduces the developers machine load, provides efficiency working in the integrated environment. This OCD explains who are the actors of the system and what is their responsibility in the system. Also, many uses of the system are discussed which gives the importance of the use of the system. OCD also tries to discuss about the Critical Issues user might face while building the system and the solutions to resolve the issue. The system is divided into cohesive packages, which can interact with each other to perform all the tasks defined in the requirements. The entire system activity diagram shown in the section 6 of this report gives us the big picture of what all modules are involved in the system, and how they interact. While, the package diagram and the activity diagram of the build server gives the information of the packages and the responsibility each package executes and the activity flow of the build server.

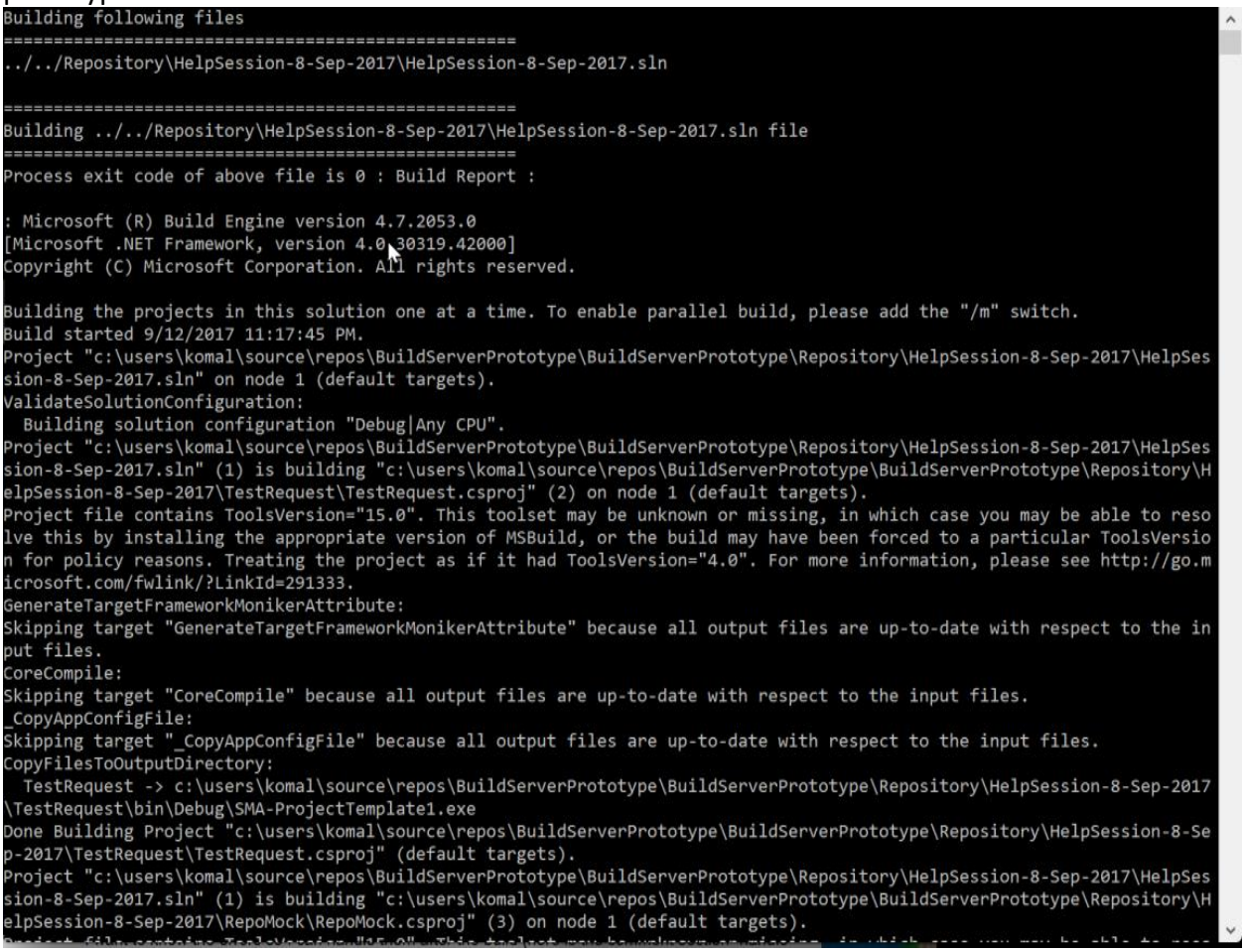
9. Appendix

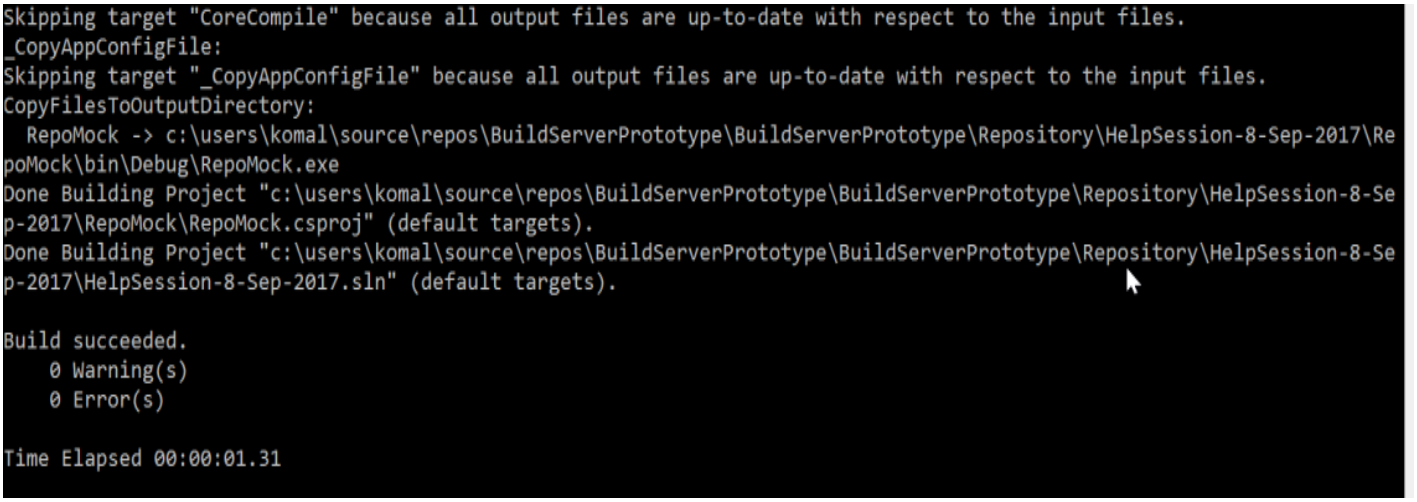
9.1. Prototype Of BuildServer

The build server prototype is used to programmatically attempt to build the solution.

Design - The prototype is implemented as an executable which accepts the absolute location of the repository, which has all the files. All the sln files are found out by diving into all the directories recursively and finding out if there exists any .sln file in the folder. It finds out all the sln files, and builds them one by one using the process library provided by the .Net framework. It displays all the build logs on the console, thus demonstrating the small prototype of what build server is intended to do. We have used System.IO namespace, which has all the necessary functions to scan through directories and get the files with particular patterns. And system.Diagnostics namespace, which has all the commands to build the test files.

I have attached the screen shots of the build log file that gets generated on running this prototype.




Skipping target "CoreCompile" because all output files are up-to-date with respect to the input files.
_CopyAppConfigFile:
Skipping target "_CopyAppConfigFile" because all output files are up-to-date with respect to the input files.
CopyFilesToOutputDirectory:
RepoMock -> c:\users\komal\source\repos\BuildServerPrototype\BuildServerPrototype\Repository\HelpSession-8-Sep-2017\RepoMock\bin\Debug\RepoMock.exe
Done Building Project "c:\users\komal\source\repos\BuildServerPrototype\BuildServerPrototype\Repository\HelpSession-8-Sep-2017\RepoMock\RepoMock.csproj" (default targets).
Done Building Project "c:\users\komal\source\repos\BuildServerPrototype\BuildServerPrototype\Repository\HelpSession-8-Sep-2017\HelpSession-8-Sep-2017.sln" (default targets).
Build succeeded.
0 Warning(s)
0 Error(s)
Time Elapsed 00:00:01.31

10. References

1. https://msdn.microsoft.com/en-us/library/ms171470.aspx
2. https://en.wikipedia.org/wiki/Continuous_integration
3. https://stackoverflow.com/
4. https://ecs.syr.edu/faculty/fawcett/handouts/CSE681/Lectures/Project1-F2017.htm#top