# Continuous Integration with TestShell

Continuous Integration (CI) is a software engineering practice which tracks changes immediately when they are added to a larger code base. The goal of CI is to provide rapid feedback so that if a bug is introduced into the code base, it can be identified and corrected as soon as possible

The CI methodology uses build servers machines, which automatically run the unit tests periodically or even after every commit and report the results to the developers.

TestShell platform provides an advanced testing mechanism as well as scheduling & queuing platform which could be integrate as part of any Continuous Integration process.

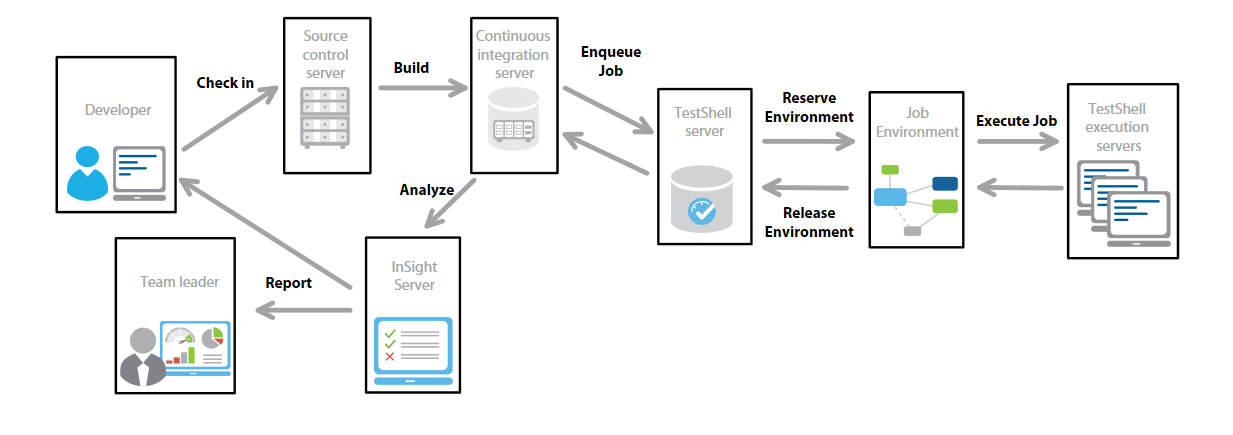
This white paper document present the principles of CI methodology as the testing section managed by TestShell suite.

**Continuous Integration ‘Classical’ Use case**

Any development *organization* which manage product development must protect the product branch from any potential bugs and inspire to locate any bugs as soon it speeded into the branch. This is the main goal of the CI process. The ‘classic’ CI flow built from several main events.

The following diagram presents the ‘classic’ use case of CI while using TestShell as the testing section orchestrator:

1. The developer commits ‘check in’ into the organization source control server.
2. The source control server recognizes the ‘check in’ and notify the CI server - the CI server preform ‘check out’ (get latest).
3. The CI server starts with the build process.
4. Once the application build process finish – the CI server queues new job in TestShell server.
5. The TestShell server reserve all the Job resources (Environment).
6. Run the job tests on the relevant execution server.
7. Once all the tests finished – the TestShell server release the reserved Environment.
8. The Continuous Integration server notifies the Insight server which update the build reports with the latest build status.
9. The InSight server notifies the developer\team that the build process has finished and what are the TestShell test results.
10. Based on the build result - the developer could validate if his ‘check in’ was legal.



**Supported systems**

There are several of CI systems available today in the market such Team-city, Jenkins, Clear Case, etc.

TestShell CI API implemented in REST API – meaning there is no limitation to the CI server machine OS.

For a given CI platform, we need to implement script which will connect TestShell server via REST API and will queue the job\suite.

**Implementation**

There are many systems of CI platform in the market and TestShell inspire to integrate with them all as part of the build process. The implementation may be different between the different platforms - but all the implantations methods should basically work like this:

Once the build process finishes deploy and build the application (the test section should initiate - it could be implemented by any language/script which support REST API:

1. The first step would be to set REST session with TestShell server machine.
2. The next step is to run the ‘Enqueue Custom Job’ - this function would reserve the requested environment and would run the Job tests. This function return ‘Job-ID’.
3. The scripts needs to run the ‘Get job details’ function and retrieve the job status, this function should be implemented using ‘polling’ methodology – sample the TestShell server every X minutes and find out if the job finishes its execution.
4. Once the job finish its execution TestShell server would end the reservation (release resources) and end the REST session.
5. The CI server can get all the tests results and analyze if the build process passed or failed.



**Best practice configuration**

1. Since TestShell implement QBA capability- it’s hard to know when each job will finish its execution\* - therefore know the execution servers in the domain and select the relevant execution server during the queuing time - the build process won’t finish until the job will finish. Selecting a dedicate execution server would provide custom control on the machine which execute the job.
2. Set different CI levels – since some of the job duration could last for a ‘long’ period – running a ‘long’ job could prevent other jobs from executed – the best practice methodology is to build set of CI level as each level runs a different set of tests (jobs). For example set of 4 different CI levels:
   1. **Daily CI operation** – triggered manually - each ‘check in’ action start build operation and run set of smoke tests, these tests should not last for a long time. Triggered by each ‘check in’
   2. **Nightly** - occurs every night on fixed time - The purpose of this build is to collect all the daily ‘check ins’ and analyze the current state of the application for a given day – the build process should run all the sanity tests which would verify the build quality. This build process could take ‘long’ period. Triggered by schedule.
   3. **Weekly** – run each weekend – this build enjoys resources usage and could run high performance testing and provide detailed build quality status. Triggered by schedule.
   4. **Release** – Before each release – run the same tests as the Weekly, but these builds save on different report, which present release build status. Triggered manually.

**TestShell server as CI server**

In case there is no CI server exist in the *organization* and the *organization* would like to implement CI methodology – it is possible to set TestShell server as the *organization* CI server.

In this use-case Each CI flow will be TestShell Job, the Job would be consisted of 2 main parts:

1. Build test – this job would commit ‘check out’ and build the application.
2. Validation tests – this job run set of validation tests which would validate the build quality.

It is recommended to use the following configurations:

1. Use single Execution Server so the 2 jobs will run synchronously and not parallel.
2. If the application is not developed on windows environment – it is possible to use TestShell custom Execution Server as the build machine, the custom Execution server could be installed on any OS environment so TestShell server could run the Build Job on the relevant environment.