Continuous Integration

The core goal of Continuous Integration is to get fast feedback. Continuously ensure that your code passed **AND** you have not broken something else:

Essentials Practices:

- Don't check In on a broken build: The cardinal sin of continuous integration is checking on a broken link. If the build breaks, the **developers** are **responsible** for fixing it. They identify the cause of the breakage as soon as possible and fix it.
- Always run all commit tests locally before committing, or get your CI server to do it for you: Performing a local sanity check before committing in the main branch. It is a way to ensure that what we believe to work actually does. Before committing, **refre sh the local copy** by updating from version control system.
- Wait for commit tests to pass before moving on: As a developer you are responsible for the build. Before moving to something else (new dev, lunch, meeting) paid attention to the build. If the build fails, **fix it immediately** or reverse your changes.
- Never go home on a broken build: if it is 5:30 pm on a Friday and have a broken build you 3 choices: Admit you will stay late tonight and fix the build. Revert your change and you can leave the office within 5 minutes, or leave the office now and leave the build broken. Be sure to understand: Monday your name will be **MUD**.
- Always be prepared to revert to the previous release: We are all humans and make mistakes and we expect everyone will break the build from time to time. If for any reason you cannot quickly fix the build now, you should **revert** to the previous stage in the version control system.
- Time-box fixing before reverting: establish a team rule: when the build breaks on check-in, try to fix it for the next 10 minutes. If, after 10 minutes you have not fixed the build, REVERT back to previous working version.
- Don't comment out failing tests: this push you on a **slippery road**. When a test that has been passing for a while begin to fail, it can be hard to work out why. Is it really a regression? Or one of the assumption of the test is no longer valid. Whatever the reasons, you have to either fix the code (in case of regression found), modify the test (if one assumption has changed) or delete the test (if the functionality under test no longer exists).
- Take responsibility for all breakages that result from your changes: if you commit a change and the tests you wrote pass, but others break, the build is still **BROKEN** and it is **YOUR** resp onsibility to fix it.
- Integration test - changes to repos that span ONAP like integration, demo, oom - a full heat/k8s deployment and minimum set of robot/rest calls should be run to verify that the deployment under change has not regressed.
- Practice Test-Driven Development: the only way to get excellent unit test coverage. Idea is to **first create a test** that is an executable specification of the expected behavior of the code to be written and then only write the code (this practice helps to drive the application design, the test serves for regression testing)

Info

- **NEVER** embed jar, war, tar, gz, gzip, zip in Gerrit

Tips

1. If there is 1 word to remember "Commit, commit, commit" multiple times a day
2. CI Practice must be in place around release planning review
3. Build Time: it should not take hours to generate a build. If it takes hours, then something is wrong and has to be fixed. A practice is to get a build in less than 3 minutes.
4. Watch the dependency diagram. You don't need to rebuild all the code for each build.
5. To get started on Jenkins, go to Getting Involved, Configuring Jenkins

References

- The content presented in this page is a very high level summary of "Continuous Delivery" book by Jez Humble and David Fairley
- To submit code into Gerrit on an unfinished feature, multiple times to bring transparency and get early feedback , refer to "Submitting a Draft Feature"
- Auto Continuous Deployment via Jenkins and Kibana