TalentSavvy Software Development Process

Last Update: October 14, 2024

Introduction

This document describes the process followed at TalentSavvy to develop the software for the TalentSavvy Team Management solution. TalentSavvy uses an Agile process under the Scrum framework and uses the Trunk-based Development model for source code branching and releases.

Software Development Tools

TalentSavvy uses the following tools and systems in its software development process:

1. **Work Tracking / Demand Management**: Azure DevOps aka Azure Boards: Azure Boards (AB) is used to track work items (Features, User Stories, Bugs, Tasks), backlogs and sprints. The Team Management solution uses 2 AB Projects, *Core* and *Assessments*.
2. **Source Code Revision Control:** GitHub (Cloud): Git on GitHub (Cloud) is used as the source code revision control system. The Team Management solution uses 3 GitHub Repositories, *talentsavvy-fe*, *talentsavvy-be-core* and *talentsavvy-utility*. The code is each repository is packaged and deployed as a Kubernetes Service on the Azure cloud.
3. **Continuous Integration (CI):** GitHub Actions: GitHub Actions are used to define and execute the test and build workflows. The Team Management solution uses two workflows for each source code repository, *<repository>-main* and *<repository>-release* to build the application container images from the trunk (main branch) and release branch respectively.
4. **Continuous Deployment (CD):** Octopus: Octopus is the CD tool used to deploy the trunk and release builds in different target environments in the Azure cloud. The Team Management solution is deployed in four targets: *Dev*, *QA*, *Stage* and *Prod*.

User Story Life Cycle

Each User Story goes through the Categories and States within each Category described below.

1. **Category: Planning**
    a. **New**: The Product Owner creates the Work Item in Azure Boards, with a Title and possibly a minimal Description.
    b. **Ready For Development:** The Product Owner provides a detailed Description, UX Design (if applicable) and Acceptance Criteria in Azure Boards.
    c. **Accepted:** The Dev Team Leader or Software Architect understands and accepts the Description, UX Design (if applicable) and Acceptance Criteria, and estimates the development effort by entering the Story Points in Azure Boards.
    d. **Assigned**: The Dev Team Leader or Software Architect assigns the Work Item to a Developer in Azure Boards. If the User Story requires work in multiple source code repositories in GitHub, sub-tasks are created in Azure Boards, and each sub-task is assigned to a Developer. Sub-tasks may be created for software design and test development.
2. **Category: Development**
    a. **Development in Progress**: Each Developer creates a Feature branch off the trunk (main branch) in GitHub to make changes in the application source code, unit and regression tests and documentation. The Feature branch is named *feature/<AB#>*.
    b. **Pull Request Created:** Each Developer commits their changes in their Feature branch and creates a Pull Request in GitHub. The Developers provide a demo of the User Story and dev test results before or along with the PR. Separate PRs are created for each source code repository that is modified. Commits and PRs are tagged with *AB#<AB#>* in the commit comment. Thus, each User Story in Azure Boards is directly linked to the corresponding Pull Requests in GitHub.
    c. **Pull Request Approved**: One or more Code Reviewers review and approve each PR in GitHub. The approval may be preceded by several rounds of review comments and code changes. If the PR is not approved, it is closed, and the work item goes back to the *Development In Progress* state.
    d. **Code Committed In Trunk**: The final Code Reviewer merges the code changes included in the PR into the trunk (main branch) in GitHub.

e. **Trunk Build Created:** The Code Commit in trunk in GitHub triggers a GitHub Action that performs the following steps:
    i. Execute unit tests.
    ii. Scan the source code using SonarQube.
    iii. Build the Kubernetes pod (container) image.

    If the Trunk Build fails, the failures are fixed by further code changes committed in the Feature branch using the same PR, which needs to be re-approved and committed in trunk. Each build is assigned a unique build number in this format: 0.0.<GitHub Action Run Number>.<Build Date>.<Git revision short hash>. For example: 0.0.537-2024-08-10-0e4f829. Thus, each PR that is merged into the trunk results in an application build with a unique build number. Since a User Story in Azure Boards is directly linked to Pull Requests, the User Story can be linked to Trunk Builds by noting the build number corresponding to the Pull Requests. Alternatively, the User Story can be linked to the first Trunk Build that is successfully created after the corresponding PR is committed into trunk.
f. **Trunk Build Deployed in Dev**: When a trunk build is created, the GitHub Action triggers an Octopus Deployment Process to deploy the application in the Dev environment in the Azure cloud.

3. Category: QA
    a. **Trunk Build Deployed in QA**: Every night at 1 am, an Octopus Deployment Process picks up the most recent application container images (a separate container image for each application service) from the Azure Container Registry and deploys them in the QA environment in the Azure cloud. Thus, the trunk build deployed in QA each night includes all the PRs merged into trunk the previous day. Sometimes, the QA manager manually deploys Trunk Builds in QA.
    b. **Automated QA Tests Passed**: Every night at 2 am, a GitHub Action runs automated Cypress tests in the QA environment in the Azure cloud. Test Failures are analyzed manually and may result in the Work Item being put back in the *Development In Progress* state or a separate Bug being created in Azure Boards.
    c. **Manual QA Tests Passed**: A QA engineer tests each User Story manually in the QA environment. Test Failures are analyzed and may result in the Work Item being put back in the *Development In Progress* state or a separate Bug being created in Azure Boards. When the Automated and Manual QA Tests pass, the Work Item is put in the Tested state in Azure Boards.
4. **Category: Release**

a. **Release Branch Created:** At the end of a Sprint or when a critical number of User Stories pass the QA Tests, the Release Manager increments the Major and/or Minor Number for the next Release and creates a Release branch named *release/<Major Number>.<Minor Number>* from trunk in GitHub. Each Release includes all the PRs (and thus the corresponding User Stories) that were committed in trunk since the previous Release. The Release Manager prepares Release Notes that list the New Features (User Stories) and Resolved Bugs in the Release.
b. **Release Build Created:** Creation of a release branch triggers a GitHub Action to build the application (Kubernetes pod image) from the release branch. This build is expected to succeed always because the code was built successfully using the same Git revision in trunk. Each build is assigned a unique build number in this format: <Major Release>.<Minor Release>.<GitHub Action Run Number>.<Build Date>.<Git revision short hash>. For example: 1.12.537-2024-08-10-0e4f829.
c. **Release Build Deployed in Stage**: When the Release Build is created, the GitHub Action triggers an Octopus Deployment Process to deploy the application in the Stage environment in the Azure cloud. The Stage Bastion VM and Kubernetes Services are stopped when Stage testing is not in progress. The deployment in Stage will fail if the Stage Bastion VM or the Stage Kubernetes Services are not running.
d. **Automated Stage Tests Passed**: The Release Manager may trigger a GitHub Action manually to run the automated Cypress tests in the Stage environment in the Azure cloud. Test Failures are analyzed manually and may result in the Work Item being put back in the *Development In Progress* state or a separate Bug being created in Azure Boards. The bug fix is cherry-picked from trunk to the release branch in GitHub which results in a new Release Build being created and deployed in Stage.
e. **Manual Stage Tests Passed**: The Release Deployment team tests each Release manually in the QA environment - typically this includes Smoke tests and User Acceptance Tests. Test Failures are analyzed and may result in one or more Work Item being put back in the *Development In Progress* state or a separate Bug being created in Azure Boards. The bug fix is cherry-picked from trunk to the release branch in GitHub which results in a new Release Build being created and deployed in Stage.

5. **Category: Production**
    a. **Release Build Deployed in Prod**: When the Stage Tests pass, the Release Manager triggers an Octopus Deployment Process to deploy the application

in the Production environment in the Azure cloud. This operation requires the Production Bastion VM to be running (the Production Bastion VM is stopped after a Release is deployed in Prod).

b. **Closed**: On successful deployment of the Release in Prod, the Release Manager marks each User Story completed in the Release as Closed in Azure Boards.

Bug Life Cycle

Each Bug goes through the Categories and States within each Category described below.

1. **Category: Planning**
    a. **New**: A QA engineer creates the Work Item in Azure Boards, with a Title and possibly a minimal Description.
    b. **Analyzed:** The QA engineer provides detailed Repro Steps for the Bug in Azure Boards.
    c. **Assigned**: The Dev Team Leader triages Bug and assigns the Work Item to a Developer in Azure Boards. If the Bug fix requires work in multiple source code repositories in GitHub, sub-tasks are created in Azure Boards, and each sub-task is assigned to a Developer.
2. **Category: Development**
    a. **Development in Progress**: Each Developer creates a Bug Fix branch off the trunk (main branch) in GitHub to make changes in the application source code, unit and regression tests and documentation. The Bug Fix branch is named *bugfix/<AB#>*.
    b. **Pull Request Created:** Each Developer commits their changes in their Bug Fix branch and creates a Pull Request in GitHub. The Developers provide a demo of the Bug fix and dev test results before or along with the PR. Separate PRs are created for each source code repository that is modified. Commits and PRs are tagged with *AB#<AB#>* in the commit comment. Thus, each Bug in Azure Boards is directly linked to the corresponding Pull Requests in GitHub.
    c. **Pull Request Approved**: One or more Code Reviewers review and approve each PR in GitHub. The approval may be preceded by several rounds of review comments and code changes. If the PR is not approved, it is closed, and the work item goes back to the *Development In Progress* state.
    d. **Code Committed In Trunk**: The final Code Reviewer merges the code changes included in the PR into the trunk (main branch) in GitHub.

e. **Trunk Build Created:** The Code Commit in trunk in GitHub triggers a GitHub Action that performs the following steps:
    i. Execute unit tests.
    ii. Scan the source code using SonarQube.
    iii. Build the Kubernetes pod (container) image.

    If the Trunk Build fails, the failures are fixed by further code changes committed in the Bug Fix branch using the same PR, which needs to be re-approved and committed in trunk. Each build is assigned a unique build number in this format: 0.0.<GitHub Action Run Number>.<Build Date>.<Git revision short hash>. For example: 0.0.537-2024-08-10-0e4f829. Thus, each PR that is merged into the trunk results in an application build with a unique build number. Since a Bug in Azure Boards is directly linked to Pull Requests, the Bug fix can be linked to Trunk Builds by noting the build number corresponding to the Pull Requests. Alternatively, the Bug can be linked to the first Trunk Build that is successfully created after the corresponding PR is committed into trunk.
f. **Trunk Build Deployed in Dev**: When a trunk build is created, the GitHub Action triggers an Octopus Deployment Process to deploy the application in the Dev environment in the Azure cloud.

3. Category: QA
    a. **Trunk Build Deployed in QA**: Every night at 1 am, an Octopus Deployment Process picks up the most recent application container images (a separate container image for each application service) from the Azure Container Registry and deploys them in the QA environment in the Azure cloud. Thus, the trunk build deployed in QA each night includes all the PRs merged into trunk the previous day. Sometimes, the QA Manager manually deploys the Trunk Build in QA.
    b. **Automated QA Tests Passed**: Every night at 2 am, a GitHub Action runs automated Cypress tests in the QA environment in the Azure cloud. Test Failures are analyzed manually and may result in the Work Item being put back in the *Development In Progress* state or a separate Bug being created in Azure Boards.
    c. **Manual QA Tests Passed**: A QA engineer tests each Bug Fix manually in the QA environment. Test Failures are analyzed and may result in the Work Item being put back in the *Development In Progress* state or a separate Bug being created in Azure Boards. When the Automated and Manual Tests pass, the Bug is put in the Tested state in Azure Boards.
4. **Category: Release**

a. **Cherry-picked**: If the Bug is classified as a High Severity bug, the Release Manager may cherry-pick the bug fix from trunk to the current release branch – before the next Release branch is created as described in the following step. The Bug then goes through the states starting with *Release Build Created* up to *Release Build Deployed in Prod* for the current release, independently from the same state transitions for the next release.
b. **Release Branch Created:** At the end of a Sprint or when a critical number of User Stories pass the QA Tests, the Release Manager increments the Major and/or Minor Number for the next Release and creates a Release branch named *release/<Major Number>.<Minor Number>* from trunk in GitHub. Each Release includes all the PRs (and thus the corresponding Bug fixes) that were committed in trunk since the previous Release. The Release Manager prepares Release Notes that list the New Features (User Stories) and Resolved Bugs in the Release.
c. **Release Build Created:** Creation of a release branch triggers a GitHub Action to build the application (Kubernetes pod image) from the release branch. This build is expected to succeed always because the code was built successfully using the same Git revision in trunk. Each build is assigned a unique build number in this format: <Major Release>.<Minor Release>.<GitHub Action Run Number>.<Build Date>.<Git revision short hash>. For example: 1.12.537-2024-08-10-0e4f829.
d. **Release Build Deployed in Stage**: When the Release Build is created, the GitHub Action triggers an Octopus Deployment Process to deploy the application in the Stage environment in the Azure cloud. The Stage Bastion VM and Kubernetes Services are stopped when Stage testing is not in progress. The deployment in Stage will fail if the Stage Bastion VM or the Stage Kubernetes Services are not running.
e. **Automated Stage Tests Passed**: The Release Manager may trigger a GitHub Action manually to run the automated Cypress tests in the Stage environment in the Azure cloud. Test Failures are analyzed manually and may result in the Work Item being put back in the *Development In Progress* state. The bug fix is cherry-picked from trunk to the release branch in GitHub which results in a new Release Build being created and deployed in Stage.
f. **Manual Stage Tests Passed**: The Release Deployment team tests each Release manually in the QA environment - typically this includes Smoke tests and User Acceptance Tests. Test Failures are analyzed and may result in one or more Work Item being put back in the *Development In Progress*

state. The bug fix is cherry-picked from trunk to the release branch in GitHub which results in a new Release Build being created and deployed in Stage.

5. **Category: Production**
    a. **Release Build Deployed in Prod**: When the Stage Tests pass, the Release Manager triggers an Octopus Deployment Process to deploy the application in the Production environment in the Azure cloud. This operation requires the Production Bastion VM to be running (the Production Bastion VM is stopped after a Release is deployed in Prod).
    b. **Closed**: On successful deployment of the Release in Prod, the Release Manager marks each Bug fixed in the Release as Closed in Azure Boards.

Note: The *State* field for a Work Item in Azure Boards may be the same as or different than its State as described above. For example, the *State* of a User Story in Azure Boards may be set to *New* when it is in the New, Ready for Development, Accepted and Assigned states. The *State* in Azure Boards may be set to *Active* when it is in the Development in Progress, PR Created and PR Approved states.

Filtering Capabilities

We would like to have the following filters to be able to view the work items in aggregate:

- Work Item type (User Story or Bug)
- Work Item aggregates (Epic or Feature)
- Release
- Time range
- Area path (tbd)

Exceptions

Any work item that has not followed the above-described process should be marked appropriately. Examples of exceptions include, but are not limited to:

- Pull requests with no associated work item (orphan pull requests)
- Pull requests with one or no reviews
- User stories that are cherry picked into production