Perceptive Reach

Integrated Reach Database System

(IRDS)

Developers Guide



Department of Veterans Affairs

**September 2015**

Version 1.0

Revision History

| Date | Version | Description | Author |
| --- | --- | --- | --- |
| 9/14/2015 | 1.0 | Final Review | Monica Mohler / Paul Bradley |
| 9/13/2015 | .6 | Minor formatting updates | Matthew Robinson |
| 9/11/2015 | .5 | Contract Compliance and Quality Assurance Review | Matthew Robinson / Kaitlin Reskovac / Radina Ivanova |
| 9/10/2015 | .4 | Technical updates to Deployment Process section | Robert Snelling |
| 9/9/2015 | .3 | Formatting and minor content updates | Matthew Robinson / Kaitlin Reskovac |
| 9/1/2015 | .2 | Miscellaneous updated throughout | Robert Snelling |
| 6/1/2015 | 0.1 | Initial Draft | Jaya Rao |

**Table of Contents**

[1. About the Developers Guide 5](#_Toc430025633)

[2. Introduction 5](#_Toc430025634)

[3. Developer Workstation Setup 5](#_Toc430025635)

[3.1. Initializing GitHub 6](#_Toc430025636)

[**3.1.1. Git Bash 6**](#_Toc430025637)

[**3.1.2. SourceTree 7**](#_Toc430025638)

[3.2. Branching Strategy 9](#_Toc430025639)

[**3.2.1. Main branches 9**](#_Toc430025640)

[**3.2.2. Supporting branches 9**](#_Toc430025641)

[3.3. GitHub Working with Feature Branches 10](#_Toc430025642)

[**3.3.1. Creating a feature branch 10**](#_Toc430025643)

[**3.3.2. Incorporating a finished feature on dev 10**](#_Toc430025644)

[3.4. Committing and Pushing Source Code Changes 10](#_Toc430025645)

[**3.4.1. Git Bash 10**](#_Toc430025646)

[**3.4.2. SourceTree 11**](#_Toc430025647)

[3.5. Working with Hotfix Branches 13](#_Toc430025662)

[**3.5.1. Creating a hotfix branch 13**](#_Toc430025663)

[**3.5.2. Incorporating a hotfix to stable 13**](#_Toc430025664)

[**3.5.3. Finishing a hotfix branch 13**](#_Toc430025665)

[3.6. Finishing out the Sprint 14](#_Toc430025666)

[3.7. Git Stash 14](#_Toc430025667)

[4. Code Compilation 15](#_Toc430025668)

[4.1. Grunt and Node.js 15](#_Toc430025669)

[4.2. SQL Server 2012 16](#_Toc430025670)

[4.3. Configuration Files 16](#_Toc430025671)

[**4.3.1. .env 16**](#_Toc430025672)

[5. Deployment Packaging 17](#_Toc430025673)

[**5.1.1. Process 17**](#_Toc430025674)

[6. Development Cycle 19](#_Toc430025677)

[7. Test Driven Development 20](#_Toc430025678)

[8. VA Cloud Server Deployments of Perceptive Reach 20](#_Toc430025679)

[8.1. Dashboard 20](#_Toc430025680)

[8.2. Databases 20](#_Toc430025681)

[9. Atlassian JIRA 21](#_Toc430025682)

[9.1. Epics, User Stories, and Tasks 21](#_Toc430025683)

[9.2. JIRA Guidelines 22](#_Toc430025684)

[**9.2.1. Linking epics, user stories, and tasks 22**](#_Toc430025685)

[**9.2.2. Adding blockers 23**](#_Toc430025686)

[**9.2.3. Smart Commits 24**](#_Toc430025687)

[10. Jenkins Continuous Integration 24](#_Toc430025688)

[11. Build Process 25](#_Toc430025689)

[11.1. Build Overview 25](#_Toc430025690)

[11.2. Build Manual Steps 25](#_Toc430025691)

[12. Additional Resources 25](#_Toc430025692)

[12.1. Documentation 25](#_Toc430025693)

[12.2. VA Perceptive Reach Innovation Site 26](#_Toc430025694)

[13. Acronyms & Abbreviations 26](#_Toc430025695)

# About the Developers Guide

This Developers Guide details the context and steps for development on the VA Perceptive Reach project and is a live document to be consulted in conjunction with the VA Perceptive Reach System Design Document and VA Perceptive Reach web application. In accordance with Human Centered Design (HCD) principles and Agile software methodology, the team will update this document continuously and deliver a revised version monthly as new information and key insights are gathered.

The intended audience for this document is all key stakeholders who participate in development, testing, or sustainment of the system. The primary stakeholder for this document will possess a technical background and understanding of web applications, continuous integration (CI), databases, and Agile methodology.

The following symbols appear throughout the documentation to alert the reader to special information or conditions.

|  |  |
| --- | --- |
| **Symbol** | **Description** |
| Note | Used to inform the reader of general information and references to additional reading material, including online information. |
| Caution | **Used to caution the reader to take special notice of critical information.** |

# Introduction

The Perceptive Reach Integrated Reach Database System (IRDS) innovation project proposes to combine technology, outreach, and clinical support to realize a clinically based data-driven early intervention and treatment solution aimed at Veteran suicide prevention.

The target users for the Perceptive Reach IRDS include a diverse set of Department of Veterans Affairs (VA) staff including frontline Veteran suicide outreach and intervention providers, mental health clinical care providers, researchers, and statisticians, in addition to other VA staff in leadership or administrative positions.

# Developer Workstation Setup

The developer workstation is typically set up sequentially with starting with the code base and ensuring connectivity to GitHub. The technology used in the development environment includes the following:

* Node.js – Software platform built on JavaScript runtime (<https://nodejs.org/>)
* ExpressJS – Web framework for Node.js (<http://expressjs.com/>)
* AngularJS – JavaScript (JS) framework (<https://angularjs.org/>)
* Bootstrap – HyperText Markup Language (HTML), Cascading Style Sheets (CSS), and JS framework (<http://getbootstrap.com/>)
* Bower - Package manager for the web (<http://bower.io/>)
* Grunt – JavaScript task runner (<http://gruntjs.com/>)
* Structured Query Language (SQL) Server – Database (<http://www.microsoft.com/en-us/server-cloud/products/sql-server/>)
* JavaScript Integrated Development Environment (IDE)
  + Sublime Text 3 – (<http://www.sublimetext.com/3>)
  + Or a comparable IDE

The code base itself, however, resides on <https://github.com>.

## Initializing GitHub

The Perceptive Reach code base is housed on a public Git repository on the Github.com domain. Upon confirming minimum workstation requirements for hardware exist, proceed to install the GitHub client and SourceTree, a Graphic User Interface (GUI)-enhanced tool for interacting with GitHub. These two items can be downloaded then installed from the following sites:

* Git Bash/Git GUI <http://git-scm.com/downloads>
* SourceTree - <http://www.sourcetreeapp.com/> (Preferred Tool)

### Git Bash

After installing the above tools, clone the Perceptive Reach Github repository to the local machine or development environment by going to the GitHub Bash console, changing to your working directory using the **cd** command, then typing

git clone <https://github.com/VHAINNOVATIONS/PerceptiveReach>

A sample working directory appropriate for development could be

**/root/path/to/cloned/PerceptiveReach/**

Once the repository has successfully been cloned, change directories using the **cd** command then switch to the relevant branch to work on the project by entering

git checkout –b <branch\_name>

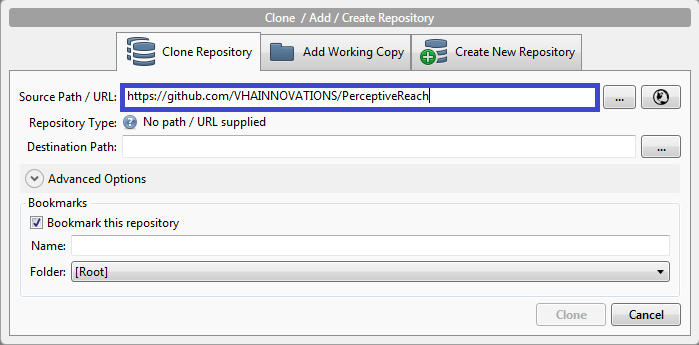
The correct branch can be determined from reading section **4.2 Branching Strategy**, but for development the command is almost always

git checkout –b dev

### SourceTree

To work with the SourceTree desktop application, the steps are the GUI-interface equivalent of the Git Bash commands, but they are issued differently as follows.

Navigate to the SourceTree application then click Clone/New and add the SourcePath URL as [**https://github.com/VHAINNOVATIONS/PerceptiveReach**](https://github.com/VHAINNOVATIONS/PerceptiveReach) as shown in Figure 1.

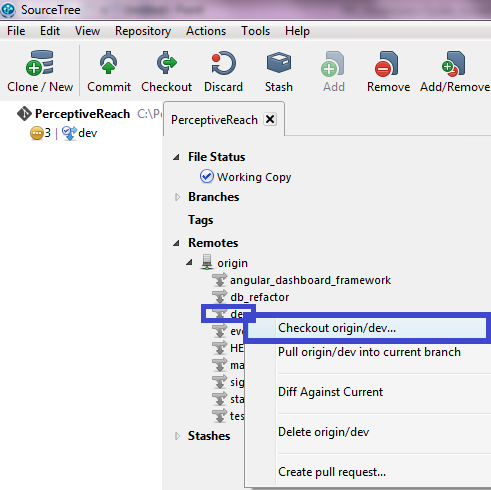


**Figure 1: SourceTree Cloning Operation**

Clone the Perceptive Reach GitHub Repository:

<https://github.com/VHAINNOVATIONS/PerceptiveReach>.

Then the correct branch can be checked out by selecting the correct branch from the **Remotes** menu in SourceTree then right-clicking the branch\_name and selecting **Checkout <branch\_name>**.



**Figure 2: Checking out dev from SourceTree**

All source code changes with happen in one of three branches, *dev* branch, a feature branch, or a hotfix branch. It is imperative for a developer to confirm what branch is active/checked-out before committing source code changes to GitHub. This can be done by reviewing the branch that is **bolded** under the “Branches” tree in the SourceTree Client or by running the following command in Git Bash: “git status”.

## Branching Strategy

The Perceptive Reach Repository incorporates a branching strategy in order to provide a structure to the repository for successful development, testing, and promotion of source code.

### Main branches

The main branches used in GitHub for VA Perceptive Reach are detailed below:

***master branch***

The *master* branch is the project deliverable branch and as such no development should occur against this branch. Source code is only merged from other branches, primarily *stable*.

***stable branch***

The *stable* branch will act as an integration/staging branch. It will contain production ready code that has passed all the continuous integrating testing scenarios. Source code will be manually merged from the *test* branch to the *stable* branch by the testing team. At the end of a Sprint, the *stable* branch will be merged into the *master* for delivery to the client.

***test branch***

The *test* branch will consist of all source code that has passed the developer unit test and is ready for continuous integration testing. Jenkins will provide an auto merge from the *dev* branch to the *test* branch. If the code passes, it will be merged into the *stable* branch, however if it fails, the code will be rolled-back to the *dev* branch.

***dev branch***

The *dev* branch is where the ongoing software development takes place. With the exception of hot fixes, all software code changes should be done against the *dev* branch. This means a developer will spend most of their time checked out against the *dev* branch.

### Supporting branches

Supporting branches are used to aid parallel development between team members and to assist in quickly fixing production problems. Unlike the main branches, these branches always have a limited lifetime, since they will always be removed eventually.

***Feature branches***

Feature branches are used to develop new features that are not ready for inclusion to the *dev* branch. This would be appropriated for development that is related to one specific feature that spans more than one Sprint. The essence of a feature branch is that it exists as long as the feature is in development, but will eventually be merged back into the *dev* branch or possibly discarded. Feature branches may branch off from *dev* and must be merge back into *dev*.

***Hotfix branches***

Hotfix branches are very much like feature branches in that they are short lived but are meant to fix a production release bug. When a critical bug in a production version must be resolved immediately, a hotfix branch may be branched off from the corresponding tag on the master branch that marks the production version. The essence is that work of team members (on the *dev* branch) can continue, while another person is preparing a quick production fix. Hotfix branches may branch off from the stable branch and must be merge back into dev, test, and stable. The naming convention should be hotfix-issue#, where issue# is the GitHub issue task number.

## GitHub Working with Feature Branches

### Creating a feature branch

When starting work on a new feature, branch off from the develop branch using the following commands.

$ git checkout -b myfeature *dev*

*Switched to a new branch "myfeature"*

A “Feature Branch” will be built against the Dev Branch. The “Feature Branch” will be created for one feature at a time. The “Feature Branch” will be tested and merged after completion.

### Incorporating a finished feature on dev

Finished features may be merged into the *dev* branch using the following commands.

$ git checkout *dev*

*Switched to branch 'develop'*

$ git merge --no-ff myfeature

*Updating… (Summary of changes)*

$ git branch -d myfeature

*Deleted branch myfeature*

$ git push origin *dev*

## Committing and Pushing Source Code Changes

Much like the GitHub initialization steps, source code change steps can be committed to GitHub with either Git Bash or SourceTree.

### Git Bash

To commit changes using the Git Bash console, navigate to the directory where Perceptive Reach has been cloned during initialization using the **cd** command to change directory.

For example, if the repository is in root\path\to\cloned\PerceptiveReach\directory, the developer would type

cd /root/path/to/cloned/PerceptiveReach/directory

Then after the correct directory path is achieved, type the git command to commit the changes.

git commit -a -m ' PR-XXXX Put your work comment/description of the changes'

 This commit command is specific to VA Perceptive Reach because it has been integrated with JIRA to update the developers’ tickets denoted by the ‘PR-XXXX’ with the developers’ GitHub commits. The specific format of the letters ‘PR’ followed by a hyphen and four numbers corresponding to the JIRA ticket number is imperative or the code and ticket will not connect to one another when the ticket is pushed.

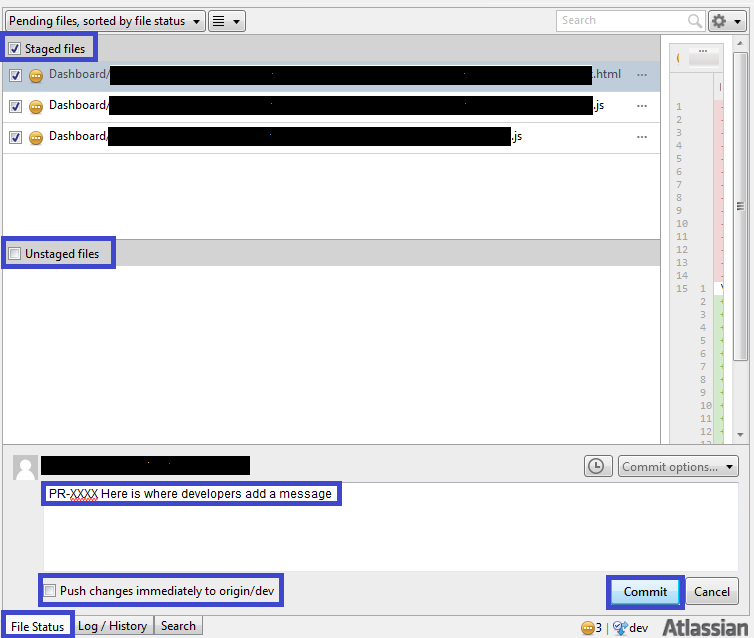
After the commit step has been completed, if a developer wishes to push changes to GitHub, he/she would issue the Git command to push, which is

git push

### SourceTree

The SourceTree client will recognize any files that have not been committed in the **Unstaged Files** area and allow you to select then commit them. The process is as follows:

* + Navigate to the **File Status** tab.
  + Select all files to be committed to GitHub. This part of the process stages the files.
  + Provide a Commit Message – “PR-XXXX Was able to finish the proof of concept….” This step is where the “Smart Commit” connecting JIRA and GitHub takes place. If Smart Commit is being used, an optional time spent working on the task can be entered such as “PR-XXXX #2h” to reflect the developer spent 2 hours working on the task. Time can be populated manually as well directly in JIRA, so it is not imperative that the time tag is added. However, it is imperative that the task number is added.
  + Click “Commit”. This step results in the commit being added to a Jira task, and if the changes checked to immediately push the changed, then the changes also push during this step, which is unadvised unless the developer pulled the latest code just before committing.



**Figure 3: Commit files from SourceTree to GitHub**

From this point if a developer wishes to **Push** the committed changes to GitHub, he or she would look at the **Pull** icon at the top of the application to see if any other developers checked in any new changes. If there are no changes to pull, the developer can click **Push** to push the changes to the dev branch.



**Figure 4: Pull and push buttons in SourceTree**

## Working with Hotfix Branches

### Creating a hotfix branch

Hotfix branches are created from the master branch in order to fix a production problem. The branch can be checked out using the following commands.

$ git checkout -b hotfix-PR-XXX X *stable*

*Switched to a new branch "hotfix- PR-XXXX "*

A “Hot Fix” will be developed against the Stable branch. Only one branch will be created per “Hot Fix” branch. After successful testing the “Hot Fix” branch will be merged with the Stable, Test, and Dev branches to close out the process.

### Incorporating a hotfix to stable

Once the bug is fixed, commit the fix in one or more separate commits as shown below.

$ git commit -m "Put your comment/description of the changes along with the GitHub Task Number”

*[hotfix-PR-XXX] Fixed severe production problem*

*5 files changed, 32 insertions(+), 17 deletions(-)*

### Finishing a hotfix branch

When finished, the bugfix needs to be merged back into *stable*, but also needs to be merged back into *dev* and *test* in order to safeguard that the bugfix is included in the next release as well.

First, update *stable* and tag the release.

$ git checkout *stable*

*Switched to branch 'stable'*

$ git merge --no-ff hotfix-PR-XXX

*Merge made by recursive.*

*(Summary of changes)*

Next, include the bugfix in *dev* and *test*, too (Repeat for each branch):

$ git checkout *dev*

*Switched to branch 'dev'*

$ git merge --no-ff hotfix-PR-XXX

*Merge made by recursive.*

*(Summary of changes)*

Finally, remove the temporary branch:

$ git branch -d hotfix-PR-XXX

*Deleted branch hotfix- PR-XXX*

## Finishing out the Sprint

Developers provide manual merges from the test branch to the stable branch and from the stable branch to the master branch at the end of a Sprint. The following commands can be used to accomplish this task as follows:

$ git checkout *stable* (or *master*)

*Switched to branch 'stable'*

$ git merge --no-ff *test* (or *stable*)

*Updating…(Summary of changes)*

$ git push origin *stable* (or *master*)

## Git Stash

If a developer would like to switch branches but does not want to commit what he or she has been working on yet, the following steps can be used to mitigate the situation.

To push a new stash onto the local stack, which will stage the uncommitted files until the developer is ready to work with them again in *dev*, he or she can use the git stash command as shown.

$ git stash

The working directory will now be clean. To verify use the following command:

$ git status

*# On branch master*

*nothing to commit, working directory clean*

At this point, any developer can easily switch branches and do work elsewhere. Changes are stored on the local stack. To see which stashes have been stored, use the git stash list command.

The following commands are used in the context of an example.

$ git stash list

*stash@{0}: WIP on master: 049d078 added the index file*

*stash@{1}: WIP on master: c264051 Revert "added file\_size"*

*stash@{2}: WIP on master: 21d80a5 added number to log*

In the above example, two stashes were created previously, so the developer has access to three different stashed works. Developers can reapply the code recently stashed by using the following commands.

$ git stash apply

*Applies the most recent stash*

$ git stash apply stash@{2}

Applies a specific one if there are more

# Code Compilation

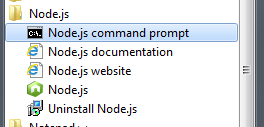
A few configurable settings and commands must be run on a freshly cloned code base to make it compile and run successfully. Once the application has run once successfully from a clean pull of the code base, these commands do not need to be issued to run the application unless the entire code base is wiped away replaced with a fresh Git clone. The configuration steps need to take place before the application is run.

## Grunt and Node.js

The Dashboard Application, client and server, build process is managed by Grunt, The JavaScript Task Runner. This enables the build process to be consistent and automates tasks like minification, compilation, unit testing, linting, etc. Grunt and Grunt plugins are installed and managed via npm, the Node.js package manager. Therefore a few steps need to take place to install Grunt and Node.js components before running the application.

* Download and install Grunt
* Download and install Node.js

The compilation process utilizes the Node.js command prompt which can be found on the start menu under the Node.js folder.



Compile the code by first going to the PerceptiveReach\Dashboard\PRDash-ADF directory then running the following in sequential order at the Node.js command prompt:

cd \PerceptiveReach\Dashboard\PRDash-ADF

npm install –g grunt-cli

npm install –g bower

npm install

bower install

Once they have successfully run, the developer can run the program by typing in **grunt serve.**

This command will deploy the project locally on port 9000 (<http://localhost:9000>)

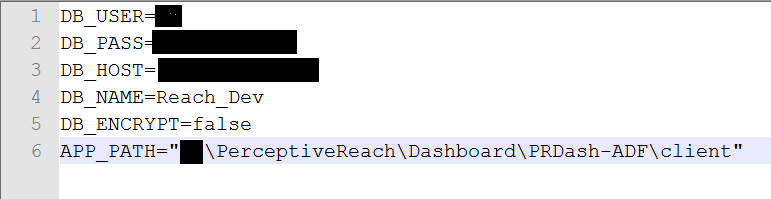
## SQL Server 2012

The Perceptive Reach web application points to a SQL Server database backend. For developers to run the application locally on their own machine or within a Virtual Machine (VM)/development environment, they must have a functional copy of SQL Server 2012 running and configured properly.

## Configuration Files

### .env

The .env file located under PerceptiveReach\Dashboard\PRDash-ADF\ must have the correct values for the Perceptive Reach database the developer would like to use. To configure the database settings, open .env in the development environment, and add the appropriate values as shown. The actual values for the db user, pass, and host will need to come from the development manager as the values may be dynamic in nature. The APP\_PATH variable should reflect the absolute path of the PRDash-ADF\client folder in the development environment.



**Figure 5: Database .env file configuration**

# Deployment Packaging

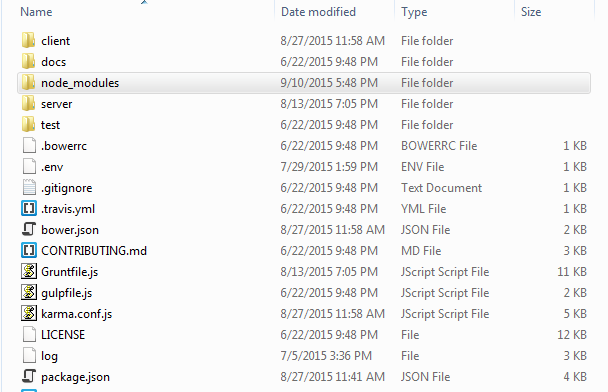
The deployment of the Perceptive Reach Dashboard and Direct applications requires the same process for packaging. The deployment packages do not need to contain all of the development dependency files but rather only what is necessary for a production deployment. Once the deployment file is completed then it can be transferred to the production server for installation.

### Process

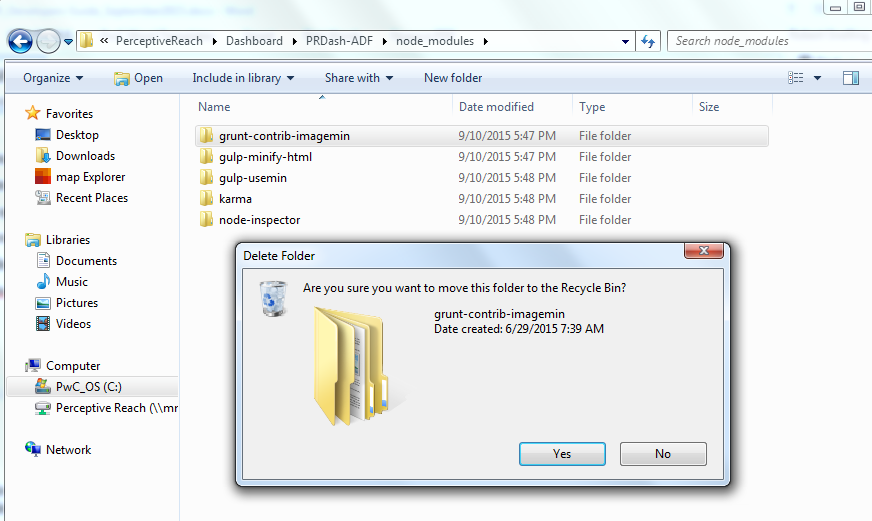
The packaging process will utilize the Node.js command prompt and an archiving tool, such as Winzip or Windows Send-to Compressed Folder.

Note: All deployments are to be built from stable and certified code.

1. Checkout the “Stable” Branch from GitHub (see 3.1.2 for examples).
2. Navigate to the root of the application:
   1. Cd C:\PerceptiveReach\Dashboard\PRDash-ADF
3. If the “node\_modules” folder exists, then delete it with the following command:
   1. rmdir node\_modules /S /Q



* 1. On occasion, Windows will prevent some of the contents of the node\_modules folder from being deleted, when this occurs delete the folder and all of its contents manually.



**Figure 6: Delete Folder Manually**

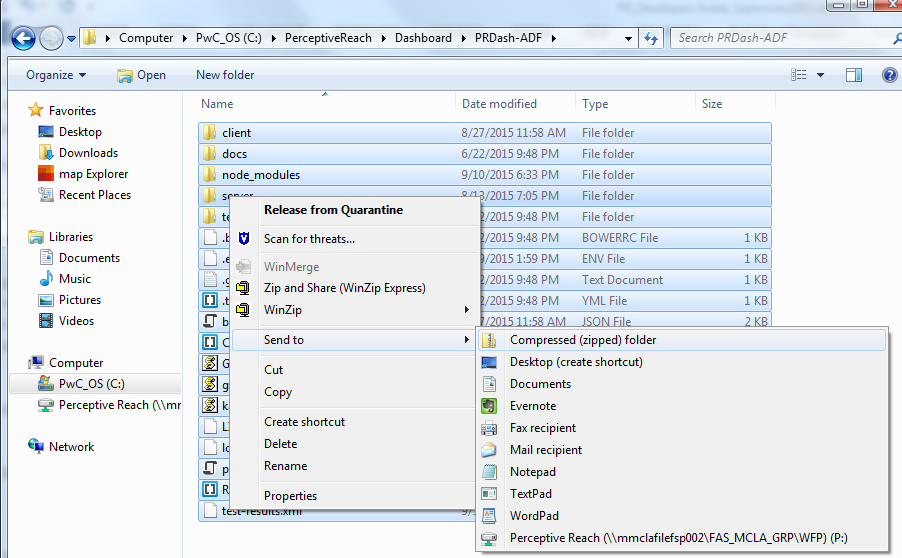
1. Run the following commands in succession:

npm install ––production (2 “–“ dashes)

npm dedupe

bower install

1. Create a compressed archive (Zip) of the contents of the PRDash-ADF folder. Do this by highlighting all of the folders and files, then right click and select “Send to”, “Compressed (zipped) folder”.



**Figure 7: Zip the PRDash-ADF Folder**

1. Rename the Zip file to match the release deployment, ie. PR\_Dashboard\_Release 2.zip.

# Development Cycle

Each development cycle typically takes on a standard pattern of events as follows.

* Developers use the **dev** branch to make changes, enhancements, and updates
* Developers create unit tests for recently developed code and verify they pass when run
* Developer commits code to GitHub **dev** branch which is tested via Jenkins
* Jenkins compiles, deploys, and tests the **dev** branch
* The **dev** branch build process successfully completes then triggers the **test** branch to update
* The **test** branch is merged with the dev branch and triggers a build process
* The **test** branch is built and tested via Jenkins
* Upon failure the CI pipeline sends emails out to prompt action
* Upon completion of Sprint and testing succeeding the code is promoted to the **stable** branch.
* The **stable** code is pushed to the **master** branch

# Test Driven Development

Test-driven development relies on the repetition of a very short development cycle. The developers usually write a test case and then write the minimum amount of code to illicit an expected test outcome. The typical steps for this sort of development form a pattern, and for Perceptive Reach they take on the following steps.

* Requirements become a user story
* The user story becomes test plan task, executed by the test team in Cucumber
* The entire team engages to refine a requirement as needed
* The development team works in Jasmine to turn the test plan into a unit test then subsequently a function or feature
* The entire team engages again to refine the requirement as needed
* The development and test team work through continuous integration
* Unit testing succeeds
* Functional/Usability testing succeeds
* The build succeeds and is promoted to the **stable** environment

# VA Cloud Server Deployments of Perceptive Reach

## Dashboard

The Perceptive Reach Dashboard is running on the VA Cloud Server running via Internet Information Services (IIS). There are three environments running from three different ports, differentiating dev from test and stable versions of the dashboard.

|  |  |
| --- | --- |
| VA Perceptive Reach Dashboard Environments | |
| Development, Dev DB | http://irdsdev.vaftl.us:7001 |
| Test, Test DB | http://irdsdev.vaftl.us:7003 |
| Stable, Stable DB | http://irdsdev.vaftl.us:7005 |

**Table 1: VA Perceptive Reach dashboard environments**

## Databases

There are currently three separate Perceptive Reach databases. The developers use Reach\_Dev for all development activity, and testers follow suit using Reach\_Test.

Reach is the production database and runs stable code. It is backed up after each Sprint, and major changes or enhancements are all stored in scripted format for use in future release guides.

|  |  |
| --- | --- |
| VA Perceptive Reach Databases | |
| Reach\_Dev | Development database built ad-hoc to produce scripts and guides for future use |
| Reach\_Test | Test database built using scripts and deployment guides |
| Reach (master database, stable) | Master database with stable code |

**Table 2: VA Perceptive Reach databases**

# Atlassian JIRA

Jira is the Sprint management tool being used for Perceptive Reach. It can be downloaded from the internet at <https://opensourceehr.atlassian.net>.

JIRA makes use of Epics, User Stories, and Tasks to track code base development and issue resolution.

## Epics, User Stories, and Tasks

**EPICS**

* Captures a large body of work
* It is essentially a large [user story](https://confluence.atlassian.com/display/AGILE/Story) that can be broken down into a number of smaller stories
* It may take several [Sprints](https://confluence.atlassian.com/display/AGILE/Sprint) to complete an epic.
* *Example: As a member of VA leadership, VA Center of Excellence for Suicide Prevention staff, VA Mental Health leaders, and VA Suicide Prevention Coordinators (hereby indicated as a "Suicide Prevention Staff Member") I want to view a surveillance dashboard with results produced from the continuous monitoring and processing of linked data sources so I can monitor and understand Suicide Outreach outcomes.*

**USER STORIES**

* A *story* or *user story* is a software system requirement that is expressed in a few short sentences, ideally using non-technical language.
* User oriented language
* *Example: As a Dashboard User, I want to view newly identified at-risk Veterans during a specific time frame.*

**TASKS**

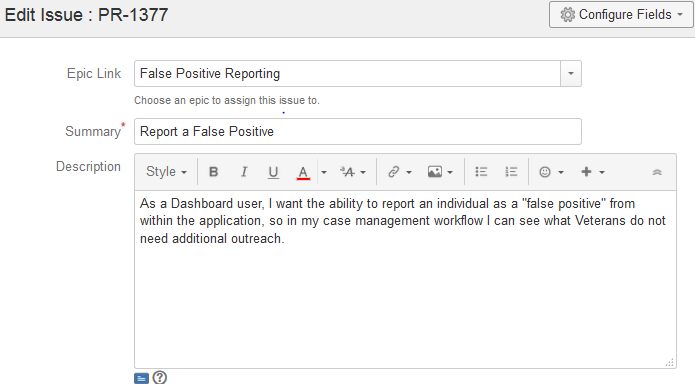
* A *task* is a unit of work contained within a [story](https://confluence.atlassian.com/display/AGILE/Story).
* *Example: Create Database/API Components for Clinical Practice Guidelines (PR-996)*
* When writing a Task it HAS to contain an action word such as: create, implement, research, enable, etc.

## JIRA Guidelines

There are several best practices in place to ensure the Agile methodology is optimized for use on this project. These guidelines govern the way the development team behaves with each other as well as the code base.

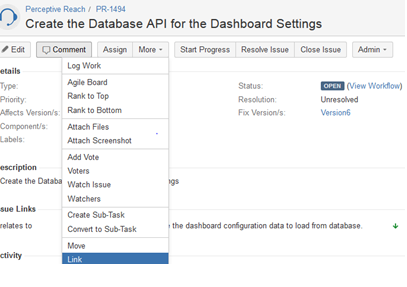
### Linking epics, user stories, and tasks

When using JIRA to document work, each epic has to be linked to one or more user stories in JIRA. When a ticket is being created, the user can link it to an epic through use of the interface.



**Figure 8: Linking an epic to a user story in JIRA**

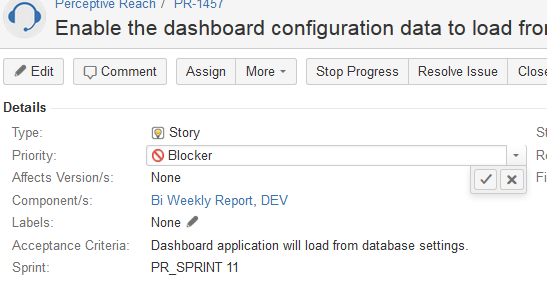
Additionally, each task must be linked to its user story.



**Figure 9: Linking a user story to a task in JIRA**

### Adding blockers

A blocker in the context of JIRA is any story, task, or issue that prevents a developer from moving forward with others. To add a blocker, change the priority of the story, task, or issue as shown.



**Figure 10: Adding a blocker in JIRA**

### Smart Commits

Smart Commits link GitHub submissions to JIRA tasks and tickets to enhance traceability.

Each git commit must be accompanied by relevant Smart Commit tags for task or issue number, time spent, and comments. Three of these four tags can be used as follows in the commit message of a Git code submission

Task number: PR-XXXX

Time spent: #time Xh Ym Zs

Comments: #comment ‘here is a comment’

Transition: **do not use**

# Jenkins Continuous Integration

Jenkins Continuous Integration tool is used to build and deploy code and third party source to multiple environments in a continuous fashion. Changes flow from development to test to stable based on gateway criteria for each environment. To honor the quality benchmarks set for the code base, the Perceptive Reach cycles code through the following environments:

|  |  |
| --- | --- |
| **Environment** | **Details** |
| **PR\_DEV** | Development branch of Perceptive Reach. Once build occurs successfully, this environment script triggers a push to PR\_DEV\_TEST. |
| **PR\_DEV\_TEST** | This environment runs development unit tests. Upon successful completion it triggers a push to PR\_TEST. |
| **PR\_TEST** | This environment is test for Perceptive Reach. Upon successful build and test it merges the dev branch with the test branch then triggers a push to PR\_TEST\_TEST. |
| **PR\_TEST\_TEST** | This environment is where Cucumber functional tests are run, and its results represent coverage of the functional requirements. |
| **PR\_STABLE** | This environment is a merge of test after the successful build and test of PR\_TEST\_TEST. It represents the completion of the Sprint, and it is the demo environment. |

**Table 3: Jenkins Continuous Integration Environments**

# Build Process

## Build Overview

## Build Manual Steps

To prepare for the build process the following tasks must be completed.

* Install Node.js (v) + IISNode (v) + IIS redirect addon
* Compile the code by first going to the PerceptiveReach\Dashboard\PRDash-ADF directory then running the following in sequential order:
  + npm install –g grunt-cli
  + npm install –g bower
  + npm install
  + bower install
* Configure: IIS
* Create a new site then point site to the directory where the compiled code.
* A web.config file must be present with a redirect to the correct file.js.
* Database: SQL Server 2012
* CI: Jenkins (v)
* Configure: install add-ons for Git, NodeJS
* Testing: Cucumber (v)
* Configure Cucumber using the following commands:
  + gem install cucumber
  + gem install selenium-webdriver
  + gem install gherkin
  + gem install capybara
  + gem install rspec

# Additional Resources

Additional resources for the VA Perceptive Reach innovation project are available in the form of companion documents to this Developers Guide and online.

## Documentation

Relevant documentation for the VA Perceptive Reach Integrated Reach Database System includes:

* *Perceptive Reach Installation Guide*: Provides detailed instructions for setting up, installing, and configuring Perceptive Reach on VA servers. Its intended audience includes server administrators, Information Rights Management IT specialists, and AngularJS application developers.
* *Perceptive Reach System Design Document*: Contains detailed information on the design considerations as well as high and low level architecture for VA Perceptive Reach.
* *Perceptive Reach Developer Guide* (this document):Contains detailed information about workstation setup, authentication, continuous integration, JIRA, SQL Server, and code base details.

## VA Perceptive Reach Innovation Site

Additional web-based information about the VA Perceptive Reach project is available on the VA Cloud at <http://vacloud.us/groups/558/>.

# Acronyms & Abbreviations

|  |  |
| --- | --- |
| **Acronym** | **Term** |
| CI | Continuous Integration |
| CSS | Cascading Style Sheets |
| GUI | Graphic User Interface |
| HCD | Human Centered Design |
| HTML | HyperText Markup Language |
| IDE | Integrated Development Environment |
| IIS | Internet Information Services |
| IRDS | Integrated Reach Database System |
| JS | JavaScript |
| SQL | Structured Query Language |
| VA | Department of Veterans Affairs |
| VM | Virtual Machine |

**Table 4: Acronyms and Definitions**