Redmine Feedback Workflow Document

Table of Contents

1.Introduction

This document serves as a comprehensive guide to the **Feedback Workflow Management System**, detailing its setup, user roles and permission, workflow, trackers and reporting features. The system is designed to simplify feedback handling within projects, ensuring organized issue resolution and integration with Redmine, a popular project management tool.

1.1. Overview

The Feedback Workflow Management System is designed to enhance the efficiency of handling feedback within Redmine by improving how feedback is received, categorized, processed, and resolved. Developed to address inefficiencies in feedback tracking and resolution, the system integrates with Redmine, a popular open-source project management and issue-tracking tool. The NPI project incorporates a structured feedback collection system that allows users to submit feedback through three different channels: directly from the Feedback section, through the Translation Feedback , or via a Feedback Form link (named post your suggestion) available on every page. This document details the integration, user roles, workflow, and management processes related to handling feedback in Redmine.

1.2. Purpose

This document outlines the workflow for feedback processing within Redmine, ensuring a structured, efficient, and transparent system for managing user-submitted feedback. It categorizes feedback within the relevant project and defines a clear workflow, enabling admins, project managers, and project users/team members to track progress and resolve feedback-related tasks effectively.

1.3. Scope

This document covers:

- Integration of the feedback system with Redmine.
- User's roles and permissions.
- Feedback workflow management.
- Customization of Redmine features.
- Reporting and security aspects.

1.4. Audience

This section identifies the primary users and their respective responsibilities. The system cates to different roles, each with specific functions:

- **Admin:** Manage users, projects, and system configurations.

- **Project Managers:** Handle feedback categorization and assignment.
- **Project users/ Team members:** View and manage assigned feedback tickets.

1.5. Key Features

- Real-time status updates and notifications.
- Reporting and analytics dashboard.
- Customizable Redmine workflows.
- Issue categorization.
- User role-based access control for managing feedback.
- Integration with Redmine & API

2. Installation & Setup

2.1. Technology Stack

- **Redmine Version:** Latest
- **Database:** PostgreSQL
- **API:** Redmine exposes some of its data through a **REST API.** This API provides access and basic **CRUD** operations (create, read, update, delete) for the resources described below.

Resource	Status	Notes	Availability
Issues	Stable		1.0
Projects	Stable		1.0
Project Memberships	Alpha		1.4
Users	Stable		1.1
Time Entries	Stable		1.1
News	Prototype	Prototype implementation for index only	1.1
Issue Relations	Alpha		1.3
Versions	Alpha		1.3
Wiki Pages	Alpha		2.2
Queries	Alpha		1.3
Attachments	Beta	Adding attachments via the API added in 1.4	1.3
Issues Statuses	Alpha	Provides the list of all statuses	1.3

Trackers	Alpha	Provides the list of all trackers	1.3
Enumerations	Alpha	Provides the list of issue priorities and time tracking activities	2.2
Issue Categories	Alpha		1.3
Roles	Alpha		1.4
Groups	Alpha		2.1
Custom Fields	Alpha		2.4
Search	Alpha		3.3
Files	Alpha		3.4
My account	Alpha		4.1
Journals	Alpha		5.0

Status legend:

1. **Stable** - feature complete, no major changes planned.
2. **Beta** - usable for integrations with some buys or mission minor functionality.
3. **Alpha** - Major functionality is in place, but feedback from API users and integrators is needed.
4. **Prototype** - very rough implementation, possible major breaking changes mid-version. **Not recommended for integration**

2.2. Prerequisites

1. **Server Setup:**
    - **Operating system:** Typically installed on Linux (Ubuntu, Debian, CentOS).
    - **Web Server:** Apache, Nginx, or Passenger.
    - **Database:** PostgreSQL for storing Redmine data.

2. **Installation & Deployment:**
    - IA Linux system with Kubernetes (k3s) installed.
    - YAML configuration knowledge for Kubernetes.
    - Ensure the Redmine version is stable and compatible with integrations.

3. **Database Configuration:**
    - Configure **PostgreSQL** as the database for Redmine.
    - Ensure database backups are enabled for data recovery.

4. **User Roles & Permissions:**
    - Enable role-based access control to manage feedback efficiently.
    - Define user roles (Admin, Project Manager, Team Member/project user) with appropriate permissions.

5. **API & Integrations:**
    - Enable **REST API** to allow external applications to connect with Redmine.

How to Set Up Redmine in Production Using Kubernetes with postgreSQL?

Step 1: Create a Namespace (if not exists):
Before deploying any resources, check if the namespace exists. If not, create one.

```
kubectl get namespace  //verify
kubectl create namespace redmine
```

Step 2: Deploy PostgreSQL Database:
Redmine needs a PostgreSQL database to store its data. Deploy the database first.

```
root@haproxy:~# kubectl create -f {path to file}.yaml
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    postgres-operator.crunchydata.com/cluster: postgres-prod
    postgres-operator.crunchydata.com/role: pgbouncer
  name: postgres-prod-pgbouncer
  namespace: postgres-operator
spec:
  progressDeadlineSeconds: 600
  replicas: 2
  selector:
    matchLabels:
      postgres-operator.crunchydata.com/cluster: postgres-prod
      postgres-operator.crunchydata.com/role: pgbouncer
  strategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        postgres-operator.crunchydata.com/cluster: postgres-prod
```

```
        postgres-operator.crunchydata.com/role: pgbouncer
    spec:
      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - podAffinityTerm:
              labelSelector:
                matchLabels:
                  postgres-operator.crunchydata.com/cluster: postgres-prod
                  postgres-operator.crunchydata.com/role: pgbouncer
              topologyKey: kubernetes.io/hostname
            weight: 1
      automountServiceAccountToken: false
      containers:
      - command:
        - pgbouncer
        - /etc/pgbouncer/~postgres-operator.ini
        image:
registry.int.india.gov.in/quayadmin/crunchy-pgbouncer:ubi8-1.21-3
        imagePullPolicy: IfNotPresent
        name: pgbouncer
        ports:
        - containerPort: 5432
          name: pgbouncer
          protocol: TCP
        securityContext:
          allowPrivilegeEscalation: false
          capabilities:
            drop:
            - ALL
          privileged: false
          readOnlyRootFilesystem: true
          runAsNonRoot: true
        volumeMounts:
        - mountPath: /etc/pgbouncer
          name: pgbouncer-config
          readOnly: true
      volumes:
      - name: pgbouncer-config
        projected:
          sources:
          - configMap:
              items:
```

```
            - key: pgbouncer.ini
              path: ~postgres-operator.ini
            name: postgres-prod-pgbouncer
```

Step 3: PostgreSQL Database Service:
The PostgreSQL database service will help Redmine communication or will communicate with Redmine.

```
root@haproxy:~# kubectl get svc -n postgres-operator
```

```
apiVersion: v1
kind: Service
metadata:
    labels:
    postgres-operator.crunchydata.com/cluster: postgres-prod
    postgres-operator.crunchydata.com/patroni: postgres-prod-ha
  name: postgres-prod-ha
  namespace: postgres-operator
    ports:
  - name: postgres
    port: 5432
    protocol: TCP
    targetPort: postgres
  type: ClusterIP
```

Step 4: Deploy Redmine Application:
Deploying Redmine using this file.

```
root@haproxt:~# kubectl create -f {path to file/ redmine-deployment}.yaml
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app.kubernetes.io/managed-by: Helm
  name: redmine
  namespace: utils
spec:
  replicas: 1
```

```
  selector:
    matchLabels:
      app: redmine
  strategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: redmine
    spec:
      containers:
      - env:
        - name: REDMINE_DB_POSTGRES
          value: postgres-prod-ha.postgres-operator.svc
        - name: REDMINE_DB_DATABASE
          value: redmine
        - name: REDMINE_DB_USERNAME
          value: postgres
        - name: REDMINE_DB_PASSWORD
          value: postgres
        image: registry.int.india.gov.in/quayadmin/redmine:2024-08-31
        imagePullPolicy: IfNotPresent
        name: redmine
        ports:
        - containerPort: 3000
          name: http
          protocol: TCP
```

Step 5: Deploy Redmine Service:
The Redmine service helps us communicate with Istio-Ingress.

```
root@haproxt:~# kubectl create -f {path to file/ redmine-service}.yaml
```

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/managed-by: Helm
  name: redmine
  namespace: utils
spec:
```

```
  ports:
  - name: http
    port: 3000
    protocol: TCP
    targetPort: 3000
  selector:
    app: redmine
  type: ClusterIP
```

To check service details:
Kubectl get svc -n redmine

Step 6: Deploy Istio Ingress Gateway:
We are using a gateway to enable external access to Redmine.

```
root@haproxy:~# kubectl create -f {path to file/ redmine-gateway}.yaml
```

```
apiVersion: networking.istio.io/v1
kind: Gateway
metadata:
  name: utils-gateway
  namespace: utils
spec:
  selector:
    istio: utilsingressgateway
  servers:
  - hosts:
    - '*'
    port:
      name: http
      number: 80
      protocol: HTTP
```

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: istio-ingressgateway
    app.kubernetes.io/managed-by: Helm
```

```
  name: istio-443-service-utils
  namespace: utils
spec:
  ports:
  - name: https
    port: 443
    protocol: TCP
    targetPort: 8080
  selector:
    app: istio-ingressgateway
    istio: utilsingressgateway
  type: LoadBalancer
```

Step 7: Istio Ingress Gateway Service in Kubernetes cluster:
Redmine is accessed through this service as an Ingress, which can also be referred to as Nginx.

```
root@haproxy:~# kubectl get svc istio-ingressgateway -n istio-system
```

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: istio-ingressgateway
    app.kubernetes.io/managed-by: Helm
  name: istio-ingressgateway
  namespace: utils
spec:
  ports:
  - name: status-port
    port: 15021
    protocol: TCP
    targetPort: 15021
  - name: http2
    port: 80
    protocol: TCP
    targetPort: 8080
  selector:
    app: istio-ingressgateway
    istio: utilsingressgateway
  type: LoadBalancer
```

Step 8: Deploy Virtual Service for Redmine:
The Virtual Service provides the hostname to Redmine, and the gateway is also defined within it.

```
root@haproxy:~# kubectl create -f {path to file/
redmine-virtualservice}.yaml
```

```
apiVersion: networking.istio.io/v1
kind: VirtualService
metadata:
  name: redmine-prod
  namespace: utils
 spec:
  gateways:
  - utils-gateway
  hosts:
  - support.india.gov.in
  http:
  - route:
    - destination:
        host: redmine.utils.svc.cluster.local
        port:
          number: 3000
```

Note: Create a YAML file.
Command:

```
kubectl apply -f {file name} -n {namespace}
```

This command will work for all because the **kind** will be defined inside the file.

2.3. Redmine Integration Steps

Steps for integration from of NPI portal with redmine by API:

Step 1: Log in to Redmine by the admin user.
Example for NPI Portal,
Username: admin
Password: keenable@123

Step 2: If a project is already created, skip this step; otherwise, create a new project.

Go to **Project** > Click on **Create New Project**, then fill in the details according to the requirements..

Step 3: Create **Issue statuses** for the project. If they are already created, you can skip this step.

Go to **Administration** > click on **Issues statuses** > click on **New status**

Step 4: Create a Tracker for Issue tracking.

Go to **Administration** > click on **Trackers** > click on **New Tracker** to create a new issue tracker.

Step 5: Create **Issue priorities** options for assigning to issues.

Go to **Administration** > go to the **Enumerations** options > click on **New value** under Issue priorities to create a new priority option.

Step 6: Create Custom fields as per requirement of the form fields.

Go to **Administration** > click on **Custom fields** > create **New custom field** > select **Issues,**then click the **Next** button.

Step 7: Enabling API Integration for the project.

To integrate the API with Redmine, you need to activate web services. Follow these steps:

Go to **Administration** > click on **Settings** > go to the **API** > check the box for **Enable REST Web Services** (marked with a blue tick)**.**

Step 8: To access APIs or integrate them, use **two** methods for authentication:

1. **Basic Authentication —** Use a username and password for authentication.
2. **API access Key —** to obtain this key, go to **My account** and click on **Show** to view the API access key.

Step 9: Get project ID from Redmine.

To get the **project ID**, you will need to query the project's endpoint.

Step 10: Use the API to get custom fields IDs.

Once you have the project ID, you need to get the **IDs of custom fields** that are required for the issue creation.

Step 11: Create Payload for Redmine Issue.

Now, with the project ID and custom fields IDs in hand, you need to prepare a **payload** for creating a new issue in Redmine.

Step 12: Send the Payload to Redmine.

Now, use the payload you created to send a **POST request** to Redmine's issues endpoint to create the issue.

2.4. Configuration Settings

The Redmine Settings provides configuration options to customize Redmine's behaviour:

General Settings:

- Customize application title, welcome text, and text formatting.
- Define host URL and protocol for email notifications and API access.
- Configure per-page items for issue lists and reports.

Display Settings:

- Set default theme, default language, time zone and date format.
- Enable or disable **gravatar icons** for users.
- Set the maximum file size for attachments.

Authentication:

- Enable /Disable self-registration for new users.
- Define password policies (complexity, expiry rules).
- Support OpenID authentication for secure login.

API:

- Enable REST API for external system integration.
- Enable JSONP support.

Projects:

- Set default trackers and issue statuses for new projects.
- Allow public project access or restrict it to certain roles.
- Enable/disable project creation for specific roles.

Issue Tracking:

- Configure default priority. Status, and tracking behavior.
- Enable sub-projects and issue relations.

Time Tracking:

- Enable time tracking for task completion estimation.
- Set required permissions for logging time entries.

Files:

- Set max upload size for files and allowed extensions.
- Enable/disable file versioning for document management.

Email Notifications:

- Enable/disable email alerts for issues, projects and user activities.
- Choose SMTP settings for mail delivery.

Incoming Emails:

- Generate an API key, you can use for the issue creation, or comments via email feature.
- Enable/disable WS for incoming emails.

Repositories:

- Enable/disable SCM, systems Redmine should provide to the individual projects. You only support several SCM-systems (e.g. only Git or only SVN).
- Generate an API key for repository management WS.

3. User Roles & Permissions

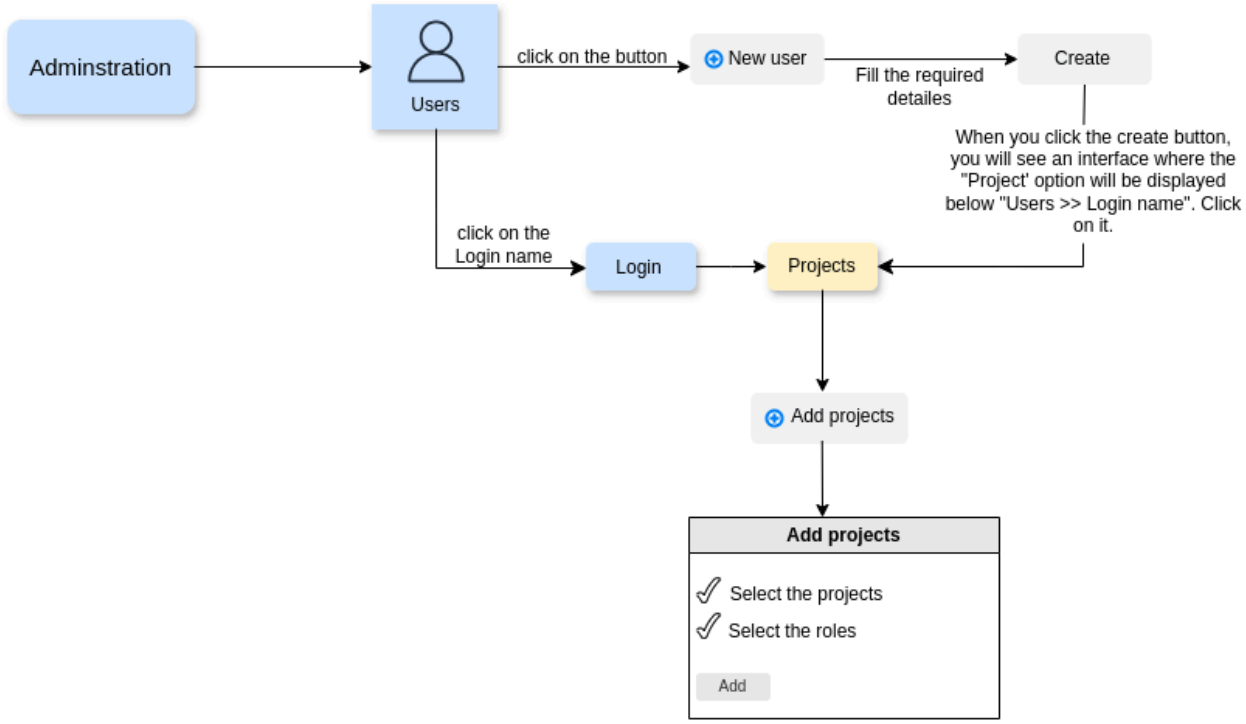
3.1. Types of User

There are 3 types of users: Admin, Project Manager, and Project User

3.1.1. Admin

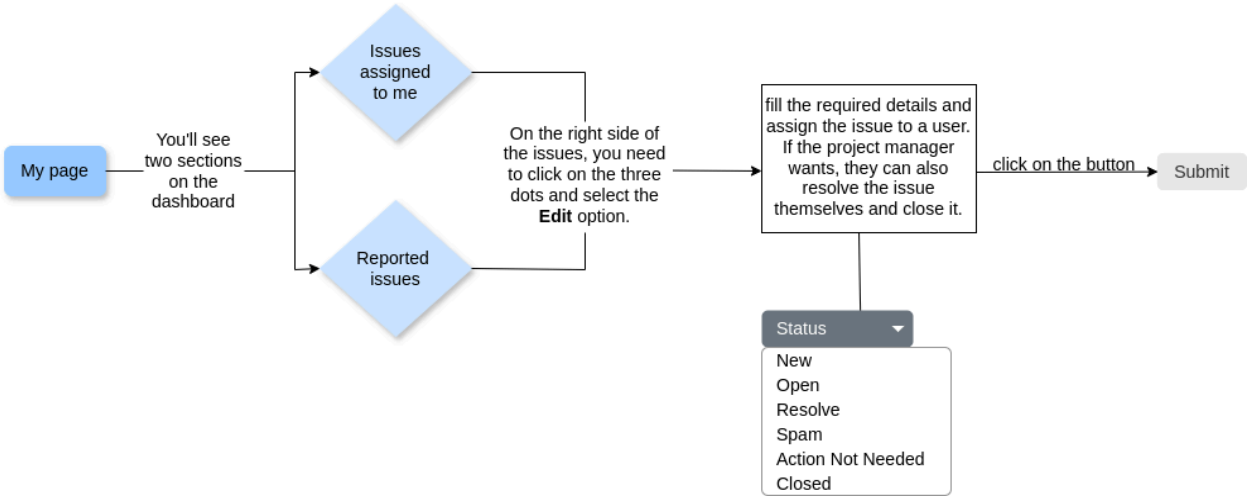
Admin have full control over the Redmine, including: create projects, create users and assign roles, etc.

How to create users and assign roles and projects:



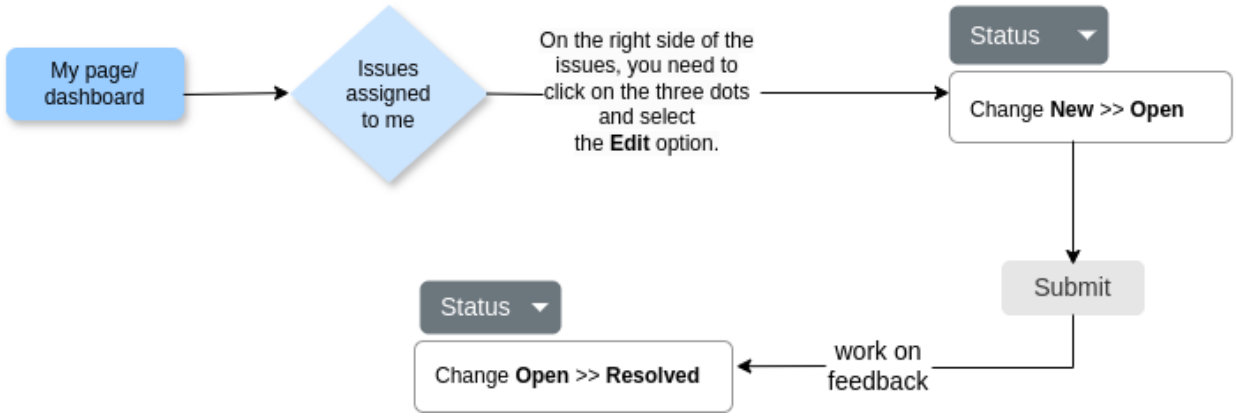
3.1.2. Project Manager

Project managers have permissions to assign issues to project users, validate and update feedback statuses, and oversee project workflows.

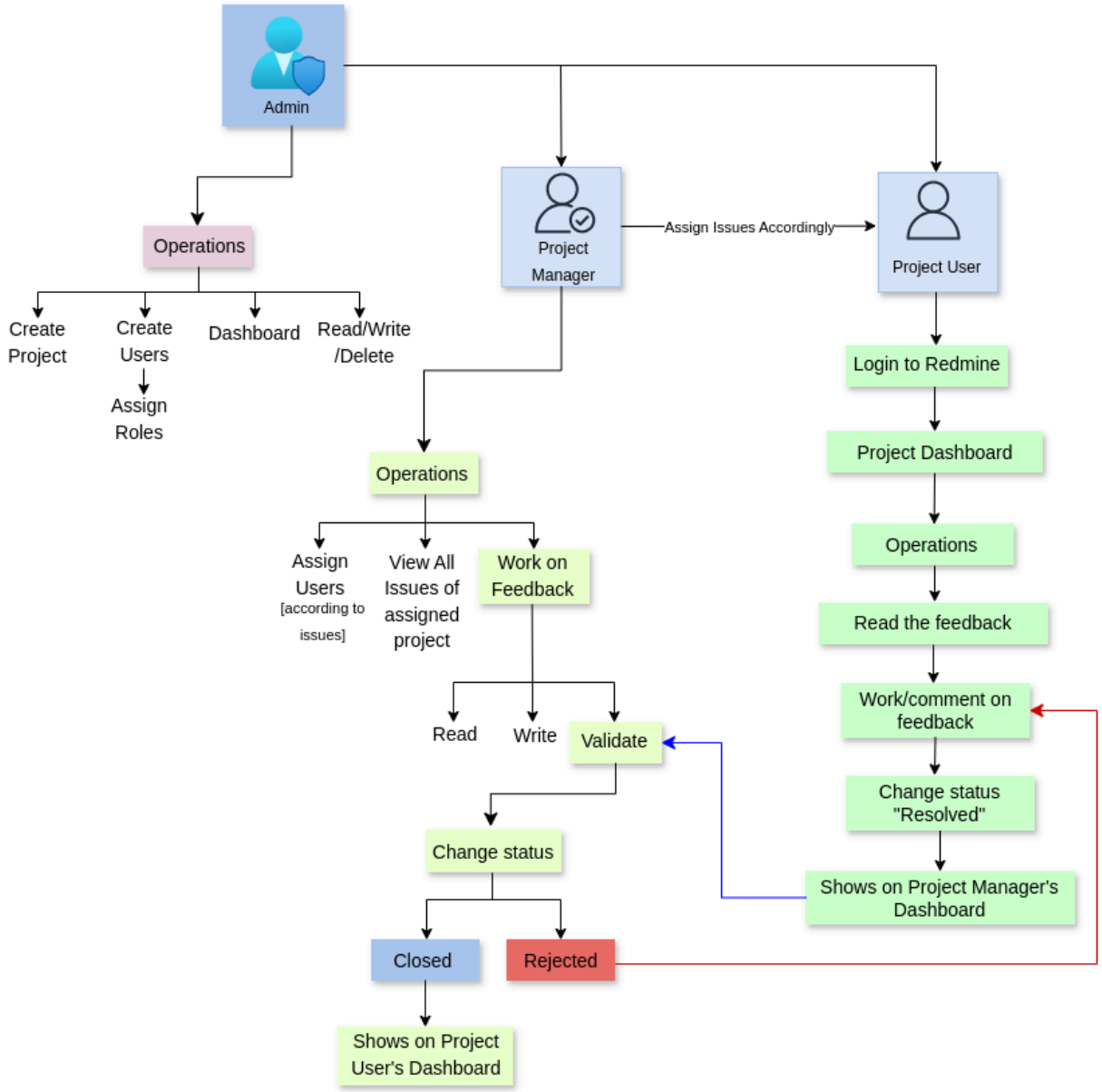


3.1.3. Project User

Project users with limited permission who works on assigned feedback issues, updates the status, and ensures issues are addressed.



How does the Redmine feedback workflow work with all users?



Redmine Workflow Diagram

3.2. Users Roles:

	Admin	Project Manager	Project User
Issue Visibility	–	All issues	Issues created by or assigned to the users

Users Visibility	–	All active users	Members of visible projects
Default spent time activity	–	none	none

3.3. Users Permissions

Projects:

	Admin	Project Manager	Project User
Create project	☑	☐	☐
Close/ reopen the project	☑	☐	☐
Set project public or private	☑	☐	☐
Manage members	☑	☐	☐
Create subprojects	☑	☐	☐
Save queries	☑	☑	☑
Edit project	☑	☐	☐
Delete the project	☑	☐	☐
Select project modules	☑	☐	☐
Manage public queries	☑	☐	☐

Calendar:

	Admin	Project Manager	Project User
View calendar	☑	☑	☑

Gantt:

	Admin	Project Manager	Project User
View gantt files	☑	☑	☑

Issue tracking:

	Admin	Project Manager	Project User

View issues	☑	☑	☑
Edit issues	☑	☑	☑
Copy issues	☑	☑	☑
Manage subtasks	☑	☑	☑
Set own issues public or private	☑	☑	☑
Edit notes	☑	☑	☑
View private notes	☑	☑	☑
Delete issues	☑	☐	☐
Add watchers	☑	☑	☑
Import issues	☑	☑	☑
Add issues	☑	☑	☑
Edit own issues	☑	☑	☑
Manage issue relations	☑	☑	☑
Set issues public or private	☑	☑	☑
Add notes	☑	☑	☑
Set notes as private	☑	☑	☑
View watchers list	☑	☑	☑
Delete watchers	☑	☐	☐
Manage issue categories	☑	☐	☐

Issue tracking for all users:

Tracker	View issues	Add issues	Edit issues	Add notes
All trackers	☑	☑	☑	☑

3.4. User Access Control

- **Admin** has full access to handle system setup, user creation, and role assignments.
- **Project Manager** assigns issues, monitors progress, and validates resolutions.
- **Project User** works on feedback, updates the status, and ensures issues are addressed.

4. Feedback Workflow Management

4.1. Feedback Submission Process

- Users submit feedback through one of the three available forms.
    - Feedback Section.
    - Translation Feedback.
    - Page-Level Feedback (named post your suggestions).
- Feedback is automatically categorized and logged in Redmine.

4.2. Issue Categorization & Tagging

4.2.1. Issue Categorization

Categorization refers to grouping issues into predefined types based on their nature and relevance. In Redmine, feedback is categorized into:

- **Valid Feedback:** Issue that requires action and needs to be assigned to a Project User.
- **Spam:** If any feedback is irrelevant, fake or unwanted feedback that should be marked and closed.
- **Not Relevant (No Action Needed):** Feedback that is duplicate, unclear, or does not contribute to improvements.

Example:
If a user submits unclear feedback like “Make it better,” the Project Manager may categorize it as **Not Relevant** and close it. However, if the feedback is specific, like “Fix the broken login button,” it will be categorized as **Valid Feedback** and assigned to a Project User responsible for resolving it. The assigned project user will see the issue on their dashboard and take appropriate action.

4.2.2. Issue Tagging

Tagging helps in further refining and filtering categorized issues by adding labels or tags based on specific attributes like priority, department, or type of issue.

Common Tags:

- **High-priority:** Critical issues that need immediate attention.

- **Frontend:** Issues related to UI/UX.
- **Backend:** Issues related to the server-side logic.
- **Database:** Issues involving database queries or structure.

4.3. Status Tracking

Status tracking helps in managing feedback efficiently by defining different status for issues in Redmine.

4.3.1. Status Options for Admin/ Project Managers

Admin/ Project Managers have access to multiple status options and can update feedback based on its relevance and resolution progress. The status options include:

- **New:** Feedback is newly received and has not been reviewed yet.
- **Open:** Feedback is assigned to a project user and is under review.
- **Resolved:** The issue has been addressed and resolved by the assigned user.
- **Spam:** The feedback is irrelevant, fake, or unnecessary and marked as spam.
- **No Action Needed:** The feedback is either duplicate or not relevant, requiring no further action.
- **Closed:** The issue has been verified and finalized.

4.3.2. Status Options for Project Users

Project Users have a streamlined workflow and can only update statuses sequentially as they work on feedback. Status progression includes: **New >> Open > > Resolved**

- **New:** When feedback is first received.
- **Open:** Once the user starts working on the issue.
- **Resolved:** When the user has fixed or addressed the feedback.

Project Users cannot mark feedback as Spam, No Action Needed, or Closed. These actions are reserved for Admin/ Project Managers.

4.4. Notifications & Alerts

Users will receive the feedback updates on their Email .

5. Redmine Features Customization

5.1. Custom Fields

Supported Data Types:

Redmine supports multiple data types for custom fields:

- **Boolean:** checkbox (Yes/No, True/False).
- **Date:** Calendar date selection

- **Float:** Floating point number (decimal numbers).
- **Integer:** Positive or negative numbers allowed (whole numbers).
- **Link:** URL.
- **List:** Predefined selectable options in a dropdown.
- **Long Text:** Resizeable multiple lines of text (supports rich text formatting as of 2.50).
- **Text:** Multiple lines text input (supports rich text formatting as of 2.50).
- **User:** Custom field format that can be used to reference a project member.
- **Version:** Custom field format that can be used to reference versions.

Custom Fields can be Added to:

- **Issues** (Bugs, Task, Feature Request, etc.).
- **Projects** (Additional metadata for projects).
- **Users** (Extra user details).
- **Time Entries** (Custom tracking information)
- **Groups & Roles** (Specific attributes for teams).

Custom Field Settings & Options:

- **Required Field:** Field which is required in order to create/save a user.
- **Visible:** Field which is displayed in the user profile.
- **Editable:** Field which is editable by the Redmine user owning the user account.

5.2. Custom Filters

Custom filters in Redmine allow users to sort, search, and categorize issues, projects, or other entities based on specific criteria. These filters help in quickly finding relevant records.

How to apply custom filters?

- Go to the **Issues** section in Redmine.
- Click on **Add Filter** in the sidebar.
- Select the **filter criteria** (for example, Status, Assignee, Date, Priority).
- Set conditions (for example, “is”, “is not”, “has been”, “any” etc.).
- Click **Apply** to see the filtered results.

Common filtering options:

- **By Issue Status:** New, Open, Resolved, Closed, Spam.
- **By Assignee:** Filter issues assigned to specific users.
- **By Priority:** High, Normal, Low.
- **By Updated:** Show last updated time and date.
- **By Project:** filter by project name.
- **By Subject:** Service Experience, Language feedback form, Portal feedback, Global feedback form etc.

5.3. Exporting Data

- Export feedback data for further analysis.
- Exports feedback reports in CSV, PDF formats.

6. Appendix

6.1. Glossary of Terms

- **API (Application Programming Interface):** A set of rules that allows different software applications to communicate with each other. Redmine provides a REST API for external integrations.
- **Attachments:** Files that can be uploaded and associated with issues, projects, or comments in Redmine.
- **CRUD Operations:** Stands for Create, Read, Update, and Delete, which are basic functions of a database or API.
- **ClusterIP:** A Kubernetes service type that allows communication between pods within the same cluster but does not expose services externally.
- **Dashboard:** A central interface where users can view and manage feedback, issues, and assigned tasks in Redmine.
- **Database (PostgreSQL):** A structured storage system where Redmine saves all data, including feedback, users, projects, and permissions.
- **Deployment (Kubernetes):** The process of running applications in a Kubernetes cluster using YAML configuration files.
- **Feedback Workflow:** The process of collecting, categorizing, assigning, tracking, and resolving user feedback in a structured manner.
- **Gateway (Ingress Gateway):** A Kubernetes component that routes external traffic to Redmine services running inside the cluster.
- **Ingress (Kubernetes):** A set of rules that allows external users to access Redmine inside a Kubernetes cluster through an Ingress Gateway.
- **Kubernetes:** An open-source container orchestration platform used to deploy and manage Redmine services.
- **Redmine:** An open-source project management and issue-tracking tool used to manage feedback and tasks.
- **Namespace (Kubernetes):** A logical grouping of resources in Kubernetes to organize Redmine and its dependencies.

- **REST API:** An API that allows Redmine to integrate with other applications for data exchange and automation.
- **Trackers:** Predefined categories in Redmine that help classify feedback based on the type of issue.
- **Integration:** the process of connecting Redmine with the NPI feedback system to automatically receive and manage feedback.
- **Enumerations:** Predefined lists in Redmine that store issue priorities, time tracking activities, and categories.
- **CSV (Comma-Separated Values):** A file format used for exporting feedback reports from Redmine for further analysis.
- **Exporting Data:** The process of extracting feedback reports from Redmine in different formats like CSV or PDF.
- **Permissions:** The level of access and control granted to different Redmine users (Admin, Project Manager, Project users/ Team Members) based on their roles.
- **Virtual Service (Kubernetes):** A Kubernetes object that defines how Redmine receives and processes network traffic.

6.2. References & Useful Links

- https://v2.india.gov.in/
- https://www.redmine.org/projects/redmine/wiki/Rest_api
- https://www.redmine.org/projects/redmine/wiki/RedmineBackupRestore