Version Control Procedure for Node-RED Project

Repository Setup

- **GitHub Repository**: All project code will be stored in a shared GitHub repository.
- **Clone the Repository**: Before starting any work, each team member must clone the repository onto their local machine.

  Command:

  ```
  git clone https://github.com/your-repository.git
  ```

Development Workflow

Step 1: Pull the Latest Code

Before starting any new work, **always make sure you have the most recent code** from the `main` branch.

1. **Pull the Latest Updates**
   Run the following command to fetch the latest changes from the `main` branch:

   Command:

   ```
   git pull origin main
   ```

2. **Check for Local Changes**
   If you have any local changes that haven't been committed, either commit or stash them to prevent conflicts.

Step 2: Make Changes Locally

Once the latest code is pulled, you can begin working on your task, whether it's editing Node-RED flows, updating configuration, or adding new features.

Step 3: Test Locally

Ensure that your code changes do not break existing functionality. Run any tests locally before proceeding.

Step 4: Commit Changes

When you've completed your task, you should commit your changes.

1. **Stage Your Changes**
   First, stage all changes using:

   Command:

   ```
   git add .
   ```

2. **Commit with a Meaningful Message**
   Commit the changes with a clear, concise message that describes the change made.

   Command:

   ```
   git commit -m "Added new search API endpoint"
   ```

   - **Commit message guidelines**:
     - Keep it short and descriptive.
     - Use present tense (e.g., "Fix bug" instead of "Fixed bug").
     - Example:
       - Good - "Fixed issue with catalog API pagination"
       - Bad - "update files"

Step 5: Push Changes

Once you've committed your changes, **push them to the shared repository**.

1. **Push to the `main` Branch**
   Push your changes to the `main` branch.

   Command:

   ```
   git push origin main
   ```

2. **Handling Push Conflicts**
   If you encounter a conflict during the push (meaning someone else has pushed changes in the meantime), you'll need to pull the latest changes first, **resolve the conflicts**, and then push again.

   Command:

   ```
   git pull origin main
   ```

   If conflicts arise, Git will mark the conflicting files. Resolve the conflicts, then:

   ```
   git add .
   git commit -m "Resolved merge conflict"
   git push origin main
   ```

Branching Strategy

Currently, the project is working on the **`main` branch**, with no branches created for new features or tasks. However, this will change when the development becomes more complex.

In the future (when feature branches are needed):

- **Create a Branch** for each new feature or bug fix.

  Command:

  ```
  git checkout -b feature/feature-name
  ```

- **Push to the feature branch** (before merging with `main`).

  Command:

  ```
  git push origin feature/feature-name
  ```

Handling Merge Conflicts

If two developers are working on the same file or area of code, **Git might raise a conflict** when trying to merge their changes.

1. **Identifying Conflicts**:
   Git will mark the files with conflicts and prompt you to resolve them.
2. **Resolving Conflicts**:
   Open the files marked with conflicts, choose which changes to keep, or combine them manually.
3. **Commit and Push After Resolving Conflicts**:
   After resolving the conflicts, commit the changes and push again.

Visibility of Changes

- All changes are visible in **GitHub's commit history**.
- You can view the commits by visiting the **repository page** in GitHub and navigating to the **"Commits"** tab.

Release Process

We are currently using the `main` branch for stable code. As the project progresses, we will introduce branches for new features and larger changes.

1. **Feature Branches**: When we start adding larger features or breaking changes, we will create branches like `feature/xyz` or `bugfix/abc`. These will allow separate development without affecting the stable `main` branch.
2. **Releases**: Once a feature is finished, it will be merged back into the `main` branch.
3. **Version Tags**: At the end of each stable release, we will tag the version using a versioning format like `v1.0.0`.

Commit Best Practices

1. **Commit Often, But In Small Chunks**
   - Commit your work frequently, but in small, logical chunks. Each commit should represent one unit of work.
2. **Meaningful Commit Messages**
   - Always write a descriptive commit message to explain the reason for the change.
3. **Don’t Push Until It’s Tested**
   - Test your code before pushing to ensure it doesn’t break existing functionality.
