SET-1

Q1. Git Scenario – Project Initialization You're starting a new project called InvoiceApp. Describe how to: ● Initialize Git in the project folder ● Create a remote GitHub repo and connect it ● Add all files, commit with the message "Initial commit", and push the project Include all necessary Git commands.

✓ Step 1: Initialize Git in the Project Folder

1. **Navigate to your project folder** (or create it if it doesn't exist):

```
mkdir InvoiceApp
cd InvoiceApp
```

2. **Initialize Git**:

```
git init
```

This creates a .git folder and makes your directory a Git repository.

✓ Step 2: Create a Remote Repository on GitHub

You can do this manually on GitHub or via GitHub CLI. Here's how to do both:

Option A: Create the repo manually (GitHub website)

1. Go to https://github.com
2. Click **New repository**
3. Name it InvoiceApp, set visibility (public/private), and **do not** initialize with a README (since you already have local files).
4. Click **Create repository**
5. GitHub will show a URL like:

https://github.com/your-username/InvoiceApp.git

Option B: (Optional) Create using GitHub CLI:

```
gh repo create InvoiceApp --public --source=. --remote=origin
```

✅ Step 3: Add All Files and Commit

Assuming you've added your project files already:

1. **Stage all files**:

```
git add .
```

2. **Commit with a message**:

```
git commit -m "Initial commit"
```

✅ Step 4: Connect to the Remote Repository

(Use the URL from GitHub in step 2)

```
git remote add origin https://github.com/your-username/InvoiceApp.git
```

You can verify with:

```
git remote -v
```

✅ Step 5: Push to GitHub

1. **Push to the main branch** (if you're using main as default):

```
git branch -M main   # Renames the branch to main (if not already)
git push -u origin main
```

Q2. JQL Advanced Search – Created, Due, and Resolution Filters Write JQL queries to: a. Find all issues created in the last 10 days that are still unresolved b. Show all issues that are due in the next 3 days c. Find all bugs that were resolved in the last 5 days Explain and show the difference between resolution = EMPTY and status != Done.

🔹 a. Find all issues created in the last 10 days that are still unresolved

```
created >= -10d AND resolution = EMPTY
```

❖ b. Show all issues that are due in the next 3 days

```
due >= now() AND due <= 3d
```

❖ c. Find all bugs that were resolved in the last 5 days

```
issuetype = Bug AND resolutiondate >= -5d
```

❖ Difference:

- resolution = EMPTY: Issue is unresolved
- status != Done: Issue is not in the "Done" status (might still be resolved)

SET-2

Q1. GitHub Scenario – Team Collaboration You're working on a group project. How would you: ● Create a new GitHub repository ● Add a README.md ● Invite your team for collaboration ● Set up branch protection on main to require pull requests

✅ 1. Create a New GitHub Repository

1. Go to GitHub and log in.
2. Click the **"New"** button on your repositories page.
3. Name the repository (e.g., **TeamProject**), set it as **public** or **private**, and **do not** initialize it with a README.
4. Click **Create repository**.

✅ 2. Add a README.md

1. Inside your repository, click the **"Add file"** button.
2. Select **"Create new file"**.
3. Name the file **README.md**.
4. Add some basic information, like:

# TeamProject

This project is developed by our team.

5. Commit the changes.

✅ 3. Invite Your Team for Collaboration

1. Go to the **"Settings"** tab of your repository.
2. In the left sidebar, click on **"Collaborators"**.
3. Under **"Manage access"**, click **"Invite a collaborator"**.
4. Enter their GitHub username or email to send an invitation.

✅ 4. Set up Branch Protection on main to Require Pull Requests

1. Go to **Settings** > **Branches**.
2. Under **"Branch protection rules"**, click **"Add rule"**.
3. In the **Branch name pattern**, enter main.

4. Check **"Require pull request reviews before merging"** and other necessary settings.
5. Click **Save changes**.

Q2. JQL Search – Transitions, Assignee, Due Dates Write JQL queries for: a. Find issues where status changed from "To Do" to "In Progress" in the last 7 days b. List all issues assigned to you that are overdue c. Show all tasks that transitioned to "Done" in the last 2 days Briefly explain the use of CHANGED keyword in tracking workflow changes.

✅ **a. Find issues where status changed from "To Do" to "In Progress" in the last 7 days**

status changed from "To Do" to "In Progress" during (-7d to now())

✅ **b. List all issues assigned to you that are overdue**

assignee = currentUser() AND due < now() AND resolution = EMPTY

✅ **c. Show all tasks that transitioned to "Done" in the last 2 days**

issuetype = Task AND status changed to "Done" during (-2d to now())

CHANGED Keyword in Tracking Workflow Changes

- The **CHANGED** keyword allows you to track any changes in the **status**, **assignee**, or other fields in the workflow.
- It's useful for tracking transitions or status updates over a period (e.g., tracking when issues move from one status to another).

Example usage: status changed from "To Do" to "In Progress" helps track specific status changes.

SET- 3

Q1. Git Scenario – Branching & PR You're developing a new search feature. What steps do you take to: ● Create a new branch search-feature ● Make changes and push the branch ● Open a pull request and assign a reviewer ● Merge the changes after review

✅ 1. Create a New Branch search-feature

1. **Check out the main branch** (or whichever default branch you're using):

```
git checkout main
```

2. **Pull the latest changes**:

```
git pull origin main
```

3. **Create a new branch search-feature**:

```
git checkout -b search-feature
```

✅ 2. Make Changes and Push the Branch

1. **Make your code changes** (e.g., implement the search feature).
2. **Stage and commit the changes**:

```
git add .
git commit -m "Add search feature"
```

3. **Push the search-feature branch to GitHub**:

```
git push origin search-feature
```

✅ 3. Open a Pull Request and Assign a Reviewer

1. Go to your repository on GitHub.
2. You'll see an option to **Create Pull Request** for the newly pushed branch (search-feature).
3. **Title the PR** and provide a description of the changes.
4. Under **Reviewers**, select the team member(s) you want to assign as reviewers.
5. Click **Create Pull Request**.

✅ **4. Merge the Changes After Review**

1. Once the pull request is approved by the reviewer(s), click **Merge pull request**.
2. Choose to either **"Merge"** or **"Squash and merge"** depending on your workflow preferences.
3. After merging, **delete the branch** if no longer needed:

```
git branch -d search-feature
git push origin --delete search-feature
```

Q2. JQL – Created/Updated/Transitioned Filter Queries Write queries for: a. Issues created between March 1 and March 10, 2025 b. Issues updated within the last 3 days c. Issues that transitioned from “In Progress” to “Testing” after April 1, 2025 Also describe when to use updated >= -3d vs created >= -3d.

✅ **a. Issues created between March 1 and March 10, 2025**

```
created >= "2025-03-01" AND created <= "2025-03-10"
```

✅ **b. Issues updated within the last 3 days**

```
updated >= -3d
```

✅ **c. Issues that transitioned from “In Progress” to “Testing” after April 1, 2025**

```
status changed from "In Progress" to "Testing" after "2025-04-01"
```

When to Use updated >= -3d vs created >= -3d

- **updated >= -3d**: Finds issues that have been **updated** in the last 3 days (e.g., status change, comments added, or other modifications).
- **created >= -3d**: Finds issues that were **created** within the last 3 days.

Use updated >= -3d when you want to track issues that have been **worked on** or **modified** recently, while created >= -3d helps you track **newly created** issues within a given timeframe.

SET-4

Q1. Git Scenario – Resolve Merge Conflict You and a teammate modified the same file in different branches. On merging, a conflict occurs. Explain: ● How to view the conflict ● How to resolve it locally ● How to commit and complete the merge

✅ 1. How to View the Conflict

1. **Attempt to merge the branches** (e.g., merging your feature branch into main):

```
git checkout main

git pull origin main

git merge feature-branch
```

2. **Git will alert you if there's a conflict**, and the conflicting files will be marked with a CONFLICT status.

3. **List conflicted files**:

```
git status
```

Conflicting files will appear under **"Unmerged paths"**.

✅ 2. How to Resolve It Locally

1. **Open the conflicted file**. Git will mark conflicts in the file like this:

```
<<<<<<< HEAD

// Your changes on the main branch

=======

// Changes from the feature branch

>>>>>>> feature-branch
```

2. **Manually resolve the conflict** by editing the file, choosing which changes to keep (yours, the other branch's, or a combination).

3. **Remove the conflict markers** (<<<<<<<, =======, >>>>>>>) after resolving the conflict.

✅ 3. How to Commit and Complete the Merge

1. **Stage the resolved file**:

```
git add <filename>
```

2. **Commit the merge**:

```
git commit -m "Resolved merge conflict between main and feature-branch"
```

3. **Complete the merge** by pushing the changes:

```
git push origin main
```

Q2. JQL – Component, Labels, Sprint Filters Write queries for: a. Find all issues in the Frontend component that are unresolved b. List issues labeled as urgent or production-fix c. Show all issues in the current sprint and assigned to your team When is it better to use labels vs components in organizing issues?

✅ **a. Find all issues in the Frontend component that are unresolved**

```
component = "Frontend" AND resolution = EMPTY
```

✅ **b. List issues labeled as urgent or production-fix**

```
labels in ("urgent", "production-fix")
```

✅ **c. Show all issues in the current sprint and assigned to your team**

```
sprint in openSprints() AND assignee in (team-member1, team-member2, team-member3)
```

When is it Better to Use Labels vs Components in Organizing Issues?

- **Labels**: Use for **ad-hoc or flexible categorization** of issues. Labels are good for indicating priorities, features, or other tags (e.g., "urgent", "bug", "security"). They can be applied freely without predefined structure.
- **Components**: Use for **structured categorization** based on your project's architecture or functional areas (e.g., "Frontend", "Backend", "Database"). Components are often tied to specific teams or areas of responsibility and are more rigid than labels.

Summary:

- Use **labels** for flexible, user-defined tags.
- Use **components** for a more formal, structured organization of project areas or technical responsibilities.

SET-5

Q1. GitHub Scenario – Team Collaboration You're working on a group project. How would you: ● Create a new GitHub repository ● Add a README.md ● Invite your team for collaboration Set up branch protection on main to require pull requests

✅ 1. Create a New GitHub Repository

1. Go to GitHub and log in.
2. Click the **"New"** button on the repositories page.
3. Name the repository (e.g., **GroupProject**), choose visibility (public or private), and **do not initialize** with a README (if you want to add one later).
4. Click **Create repository**.

✅ 2. Add a README.md

1. Inside your newly created repository, click on **"Add file"** and select **"Create new file"**.
2. Name the file **README.md**.
3. Add basic information like:

# GroupProject

This project is developed by our team.

4. Click **Commit new file** to save it.

✅ 3. Invite Your Team for Collaboration

1. Go to your repository's **"Settings"** tab.
2. On the left sidebar, click **"Manage access"**.
3. Click **"Invite a collaborator"**, and enter their GitHub username or email to send an invitation.

✅ 4. Set up Branch Protection on main to Require Pull Requests

1. Go to the **"Settings"** tab of your repository.
2. On the left sidebar, click **"Branches"**.
3. Under **"Branch protection rules"**, click **"Add rule"**.

4. In the **"Branch name pattern"** field, enter main.
5. Enable the rule **"Require pull request reviews before merging"** and select other protection options as needed (e.g., require status checks, restrict who can push).
6. Click **Save changes**.

Q2. JQL – Release Readiness Queries Write queries to: a. Find all issues targeted for fixVersion = "v2.0" b. List bugs in v2.0 that are still unresolved c. Find tasks in v2.0 that were resolved in the last 7 days Explain why fixVersion is useful for managing product releases.

✅ **a. Find all issues targeted for fixVersion = "v2.0"**

fixVersion = "v2.0"

✅ **b. List bugs in v2.0 that are still unresolved**

fixVersion = "v2.0" AND issuetype = Bug AND resolution = EMPTY

✅ **c. Find tasks in v2.0 that were resolved in the last 7 days**

fixVersion = "v2.0" AND issuetype = Task AND resolutiondate >= -7d

---

Why is fixVersion Useful for Managing Product Releases?

- **fixVersion** helps track which issues (bugs, features, tasks) are being worked on or are resolved for a specific version of the product. It’s crucial for release planning as it:
    - Clearly associates issues with specific versions/releases.
    - Helps prioritize work for upcoming releases.
    - Provides visibility into the status of a release, helping ensure that the version is ready for deployment.
    - Simplifies tracking of progress across multiple issues tied to a release.

SET-6

Q1. Git Scenario – Tag and Release

✅ **1. Tag the Release as v1.0**

1. **Make sure you are on the correct branch** (typically main or master):

```
git checkout main
```

2. **Tag the release**:

```
git tag -a v1.0 -m "Release version 1.0"
```

- -a v1.0: Creates a new tag named v1.0.
- -m "Release version 1.0": Adds a message for the tag.

✅ **2. Push the Tag to GitHub**

1. **Push the tag to GitHub**:

```
git push origin v1.0
```

This pushes the v1.0 tag to the remote repository on GitHub.

✅ **3. Create a Release on GitHub from the Tag**

1. Go to your repository on GitHub.
2. Click on **"Releases"** in the navigation menu.
3. Click the **"Draft a new release"** button.
4. Select the **v1.0** tag from the dropdown.
5. Fill in release notes, and click **"Publish release"**.

This will create a GitHub release associated with your v1.0 tag.

Q2. JQL – Overdue & SLA Queries

✅ **a. Show all issues that are overdue by more than 2 days**

```
due < now() AND due < -2d
```

✅ **b. List all issues that must be resolved within 48 hours (use SLA label or custom field if available)**

Assuming you have a custom field or label like SLA that tracks the SLA:

```
labels = SLA AND due <= 48h
```

Alternatively, if using a custom field like SLA Due Date:

```
"SLA Due Date" <= now() + 48h
```

✅ **c. Find issues with a due date within this week**

```
due >= startOfWeek() AND due <= endOfWeek()
```

How JQL Helps with SLA Enforcement in Service-Based Teams

JQL enables service-based teams to easily track and manage their SLA (Service Level Agreement) compliance by filtering issues based on custom SLA fields or labels (e.g., SLA Due Date). With queries like due <= 48h or filtering based on due dates, teams can monitor issues that need immediate attention or are overdue, ensuring timely resolution and customer satisfaction.

JQL also allows for the automation of reports that highlight SLA breaches or approaching deadlines, which helps ensure adherence to agreed-upon service timelines. This improves efficiency and accountability within service teams, particularly in environments like IT support, customer service, and project management.

SET-7

Q1. Jira Scenario – Scrum Project Setup You're setting up a new Scrum project. Explain how to: ● Create the project ● Set up a board and sprints ● Add backlog items ● Start and manage a sprint

✅ **1. Create the Project**

1. **Go to Jira** and click on the **"Create Project"** button.
2. Select **"Scrum"** as the project type.
3. Choose a template (e.g., **Scrum Software Development**).
4. Give the project a name (e.g., **"New Scrum Project"**).
5. Select **"Create"** to finalize the setup.

✅ **2. Set up a Board and Sprints**

1. After creating the project, Jira will automatically create a Scrum board.
2. **To create a board manually**:
   - Go to **Boards** > **Create board** > Select **Scrum board**.
   - Choose the project and configure board settings.
3. To create a sprint:
   - Go to the **Backlog** view.
   - In the **Backlog** section, click **Create Sprint** to add a new sprint.

✅ **3. Add Backlog Items**

1. In the **Backlog** view of your project, click on **Create Issue**.
2. Fill in the details for each backlog item (e.g., **Story**, **Task**, **Bug**).
3. As you create items, they'll appear in your backlog.

✅ **4. Start and Manage a Sprint**

1. In the **Backlog** view, select the issues you want to include in the sprint.
2. **Drag the issues** into the **Sprint** section.
3. To **start the sprint**, click **Start Sprint**.

   - Set the **Sprint Name**, **Duration**, and **Start Date**.
   - Click **Start**.
4. As the sprint progresses, manage it through the **Active Sprint** board, where you can move issues between columns (e.g., To Do, In Progress, Done).

Q2. JQL – Team Assignment and Status Filters Write queries to: a. Find all unresolved tasks assigned to Team-A b. Find issues currently in "Blocked" status for more than 2 days c. List all bugs assigned to your name and in "In Review" Why is it valuable to include team-level JQL filters on sprint dashboards?

✅ **a. Find all unresolved tasks assigned to Team-A**

assignee in (Team-A) AND resolution = EMPTY AND issuetype = Task

✅ **b. Find issues currently in "Blocked" status for more than 2 days**

status = Blocked AND updated <= -2d

✅ **c. List all bugs assigned to your name and in "In Review"**

assignee = currentUser() AND issuetype = Bug AND status = "In Review"

Why is it Valuable to Include Team-Level JQL Filters on Sprint Dashboards?

Including team-level JQL filters on sprint dashboards is valuable because it:

1. **Provides Visibility**: Allows team members and stakeholders to track progress on the issues specific to their team, helping to avoid confusion with other teams' work.
2. **Improves Focus**: Ensures that each team can quickly access and prioritize the issues that are directly relevant to them during a sprint.
3. **Facilitates Performance Monitoring**: Helps track the team's performance in real-time, identifying bottlenecks and unassigned tasks for quick resolution.
4. **Enhances Accountability**: By filtering issues assigned to specific teams or team members, it promotes accountability and encourages teams to manage their workload effectively.

Overall, team-level filters help teams stay organized and aligned within the sprint, ensuring a smoother workflow and faster resolution of blockers.

SET-8

1. GitHub Scenario – Fork and Contribute You want to contribute to an open-source project. Describe how to: ● Fork the repo ● Clone your fork ● Create a branch, make changes, and push ● Open a pull request to the original repository

✅ 1. Fork the Repo

1. Go to the **original open-source repository** on GitHub.
2. Click on the **"Fork"** button at the top-right corner of the page. This will create a copy of the repository under your own GitHub account.

✅ 2. Clone Your Fork

1. Go to your **forked repository** on GitHub.
2. Click on the **"Code"** button and copy the repository URL (either HTTPS or SSH).
3. Clone the repository to your local machine:

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

4. Navigate into the cloned directory:

```
cd repository-name
```

✅ 3. Create a Branch, Make Changes, and Push

1. **Create a new branch** for your changes:

```
git checkout -b feature-branch
```

2. **Make the necessary changes** to the files.
3. **Stage and commit your changes**:

```
git add .
git commit -m "Add feature or fix issue"
```

4. **Push the branch** to your forked repository:

```
git push origin feature-branch
```

✅ 4. Open a Pull Request to the Original Repository

1. Go to the **original repository** on GitHub.

2. You should see a **"Compare & pull request"** button. Click it.
3. Review your changes, add a description, and make sure you're comparing your feature-branch to the correct branch (usually main or master) of the original repository.
4. Click **"Create pull request"** to submit your changes for review.

Q2. JQL – Time-Sensitive Filters & Workload Tracking Write JQL queries to: a. Find issues where created >= startOfWeek() b. Find all issues resolved in the previous quarter c. Show unresolved issues due within the next 7 days and assigned to your team Explain the use of startOfWeek(), startOfQuarter(-1), and <= 7d in time-sensitive reporting.

✅ **a. Find issues where created >= startOfWeek()**

created >= startOfWeek()

This query finds all issues created from the **start of the current week**.

✅ **b. Find all issues resolved in the previous quarter**

resolutiondate >= startOfQuarter(-1) AND resolutiondate <= endOfQuarter(-1)

This query finds all issues that were resolved **during the previous quarter**. The -1 indicates the **previous quarter**.

✅ **c. Show unresolved issues due within the next 7 days and assigned to your team**

due <= 7d AND resolution = EMPTY AND assignee in (team-member1, team-member2, team-member3)

This query shows unresolved issues with a **due date** within the next **7 days**, and assigned to your team.

Explanation of Time-Sensitive JQL Functions

- **startOfWeek()**: Represents the beginning of the current week (typically Monday). It is useful for filtering issues created or updated from the beginning of the week to track weekly progress.
- **startOfQuarter(-1)**: Represents the start of the **previous quarter**. This function helps in filtering issues that were resolved or created in the last quarter for historical reporting.
- **<= 7d**: Filters issues with **due dates within the next 7 days**. It is useful for setting up time-sensitive reminders or reports to track upcoming deadlines and manage workload.