Link Jira tasks with Confluence to streamline task tracking and progress monitoring for the Library Management System development. • Create a new page in Confluence titled "Library Management System Project Overview." • Embed at least 5 Jira issues related to the development of the Library Management System (e.g., tasks from the sprint like "Develop book search functionality," "Create user login page," etc.). • Use the Jira macro to display issues with status (e.g., "To Do," "In Progress," "Done"). • Add a progress bar in the Confluence page to visually track the completion of each embedded Jira task (e.g., percentage of tasks completed in the sprint). • Submit a screenshot of the Confluence page showing the embedded Jira tasks and the progress bar.

Jira and confluence….  
  
**Aim**

To integrate Jira and Confluence for seamless task tracking, progress monitoring, and collaboration during the development of the **Library Management System**.

**Software Used**

1. **Jira**: For creating and managing development tasks, tracking progress, and monitoring project status.
2. **Confluence**: For documentation and providing a centralized project overview page.
3. **Browser/Tool Add-ons**: Jira-Confluence Integration Plugin (if not natively enabled).

**Procedure**

**- Linking Jira with Confluence**

- Creating a Confluence Page

**-Embedding Jira Tasks in Confluence**

-embed at least 5 Jira tasks related to the development sprint, such as:

* Develop book search functionality.
* Create user login page.
* Design librarian dashboard.
* Implement overdue book notifications.
* Integrate QR code scanning.

**-Documentation and Submission**

**Output**

1. A **Confluence page** titled **"Library Management System Project Overview"**, containing:
   * Embedded Jira tasks related to development.
   * Real-time task statuses displayed (e.g., "To Do," "In Progress," "Done").
   * A progress bar visually tracking sprint completion.
2. A screenshot of the Confluence page with the above elements for submission.

**Result**

1. Enhanced team collaboration with centralized task tracking and progress visibility.
2. Streamlined project monitoring, enabling quick identification of bottlenecks and task statuses.
3. Improved transparency for stakeholders with a clear view of development progress.
4. A well-documented overview page that bridges task management and documentation seamlessly.

Create a Static Website and Containerize, Build & Serve it using Docker. Tasks: 1. Create a Simple Static Website (index.html file) with basic HTML content. 2. Write/create a Dockerfile to serve the website using Nginx. 3. Build the Docker Image 4. Run the container: 5. Access the Website using a Browser

Docker

**Aim**

To create a simple static website, containerize it using Docker with Nginx, and serve it for browser access.

**Software Used**

1. **Text Editor**: Visual Studio Code, Sublime Text, or Notepad++.
2. **Docker**: For containerization.
3. **Web Server**: Nginx for serving the static website.
4. **Browser**: Any web browser (Chrome, Firefox, etc.) to access the website.

**Procedure**

1. Create a Simple Static Website

2. Write a Dockerfile

FROM nginx:latest

COPY index.html /usr/share/nginx/html/

EXPOSE 80

1. Build the Docker Image

Run - --- docker build -t static-website .

1. Run the Docker Container

Run --- docker run -d -p 8080:80 static-website

5. Access the Website Using a Browser

**Output**

1. A static website with the message from the html file
2. A running Docker container serving the website on <http://localhost:8080>

**Result**

1. Successfully created a static website and containerized it using Docker.
2. Demonstrated the ability to serve static content using Nginx within a Docker container.
3. Verified the website's accessibility through a web browser.

Create a Simple Python Flask API, Containerize the Application, Build & Push the Image using Docker and Deploy the Application using Kubernetes. Tasks: 1. Create a Simple Flask API by writing a Python file (app.py) with basic endpoints. 2. Containerize the Flask App using Dockerfile. 3. Build the Image using Docker. 4. Push the Image to Docker Hub. 5. Create Kubernetes manifests (Deployment YAML & Service YAML) to deploy the application. 6. Apply the Manifests and Access the API via NodePort.

Docker & Kubernetes

Aim

To create a simple Flask API, containerize it using Docker, push the image to Docker Hub, and deploy the application using Kubernetes.

Software Used

1. Python: For creating the Flask API.
2. Docker: For containerizing the Flask application and pushing the image to Docker Hub.
3. Kubernetes: For deploying the containerized application.

Procedure

1. Create a Simple Flask API
2. Create a file named app.py

2. Containerize the Flask App Using Dockerfile

3. Build the Image Using Docker

4. Push the Image to Docker Hub

5. Create Kubernetes Manifests

6. Deploy Using Kubernetes

Output

1. A running Flask API accessible through endpoints / and /status.
2. Docker image pushed to Docker Hub.
3. Kubernetes deployment and service running with the API accessible via a browser or curl command.

Result

1. Successfully created, containerized, and deployed a simple Flask API.
2. Demonstrated integration of Docker and Kubernetes for application deployment and scaling.
3. Verified functionality by accessing API endpoints through a NodePort service.

Create a new GitHub repository for a personal project. Initialize the repository with a README.md file. Then, clone the repository to your local machine, make some changes to the README.md (e.g., add a project description), and push the changes back to GitHub. Tasks: • Create a GitHub repository. • Clone the repository to your local machine. • Edit the README.md file. • Stage, commit, and push your changes to GitHub.

Aim

To create a GitHub repository for a personal project, initialize it with a README.md file, clone it to the local machine, edit the README file, and push the changes back to GitHub.

Software Used

1. Git: Version control system.
2. GitHub: Online platform to host the repository.
3. Text Editor: Visual Studio Code, Notepad++, etc., for editing files.

Procedure

1. Create a GitHub Repository

1. Log in to your GitHub account.
2. Click on the + icon in the top-right corner and select New repository.
3. Provide a repository name (e.g., personal-project).
4. Check the option to Initialize this repository with a README.
5. Click on Create repository.

2. Clone the Repository to Your Local Machine

command prompt and run the following - git clone <repository-url>,cd personal-project

3. Edit the README.md File

4. Stage, Commit, and Push Changes to **GitHub**

Run - git status, git add README.md, git commit -m "new file",git push origin main

**Output**

1. The updated README.md file with the project description is visible in the GitHub repository.

**Result**

1. Successfully created a GitHub repository and initialized it with a README.md file.
2. Cloned the repository, made changes to the README.md, and pushed the updates back to GitHub.
3. Demonstrated proficiency in basic Git and GitHub operations.