GitLab Certified Git Associate Exam

GitLab Organization Structure

- **GitLab** is a web-based DevOps lifecycle tool that provides a Git repository manager providing wiki, issue-tracking, CI/CD pipeline features, and much more.
- **Hierarchy**:
    - **Groups**: The highest level in GitLab's structure; used to organize projects and users.
    - **Subgroups**: Nested groups within groups, allowing better organization of related projects.
    - **Projects**: Repositories that store code and resources for individual applications or services.

Epics

- **Epics** are large bodies of work that encompass multiple issues and can span projects and milestones. They help break down significant initiatives into smaller, manageable tasks.
- Epics can be nested within other epics, providing multi-level tracking.
- Useful for tracking progress on large features, organizational goals, or multiple related projects.

Groups

- **Groups** are collections of projects and users. A group can include multiple projects and serve as a way to manage permissions for multiple repositories at once.
- You can control member access at the group level, making it easier to manage permissions across multiple projects.
- **Use Cases**: A group can represent a team, department, or product line.

Sub Groups

- **Subgroups** are hierarchical layers within groups, allowing for more detailed organization.
- They enable grouping projects by more specific criteria, such as by team, function, or feature.

- Example: A **Frontend** group might have subgroups for **React Team** and **Vue Team**.

Projects

- A **Project** in GitLab is essentially a repository that contains your code, issues, CI/CD pipelines, and other resources.
- Each project can be assigned members, have its own boards, milestones, and pipelines.
- Public and private projects can be created depending on access needs.

Issues

- **Issues** are GitLab's method of tracking tasks, bugs, and feature requests. Each issue can be assigned to a team member, labeled, prioritized, and linked to a specific milestone or board.
- **Issue templates** can be used to standardize issue creation across projects.

Participants

- **Participants** in GitLab refer to users involved in discussions or actions within an issue, merge request, or milestone.
- You can mention participants in comments by using the @ symbol followed by their username.

Labels

- **Labels** are used to categorize issues, merge requests, or epics. They provide a way to filter and organize work.
- Labels can be used to mark the status of an issue (e.g., **bug**, **enhancement**, **in progress**) or indicate priority.

Milestones

- **Milestones** allow you to group issues and merge requests under a common goal or timeframe, often linked to a release.
- Progress can be tracked as a percentage of completed issues tied to the milestone.

Boards

- **GitLab Issue Boards** provide a Kanban-style view for tracking issues and merge requests.
- You can create columns for issues in different stages, such as "To Do," "In Progress," and "Done."
- Boards can be configured per project or group level.

GitLab Terms

- **Repository**: Where your project's files are stored, including code, documentation, and CI configuration.
- **Pipeline**: A set of steps that GitLab CI/CD runs on your project, such as build, test, and deploy.
- **Merge Request**: GitLab's version of a pull request, allowing code to be reviewed and merged.

Branch

- A **Branch** is a parallel version of your project's code. You can create multiple branches from the main branch to work on features or fixes independently.
- Common branches include **main** (or **master**) and **development** branches.

Tag

- **Tags** are specific points in Git history often used to mark releases (e.g., `v1.0.0`).
- Tags are immutable, meaning they won't change and are useful for referring back to stable versions of the code.

Checkout

- **git checkout** is a command used to switch between different branches or to restore files in your working directory.

Merge

- **Merging** is the process of integrating changes from one branch into another, typically from a feature branch into the main branch.
- GitLab allows for **merge requests** to handle this with peer review and CI checks before merging.

Commit

- A **Commit** is a snapshot of your changes in the repository.
- Each commit records who made the changes, when, and the changes themselves.
- GitLab uses commits to track changes to files and logs them as part of the project history.

Push

- **Push**: The process of uploading commits from your local Git repository to a remote repository (e.g., GitLab).

Workspace

- A **Workspace** in GitLab is your working environment, including projects, pipelines, and other tools available for development.

Git Commands

- **git init**: Initialize a new Git repository.
- **git add**: Stage files for commit.
- **git commit**: Save changes to the repository.
- **git push**: Push local changes to the remote repository.
- **git pull**: Fetch and integrate changes from the remote repository.
- **git merge**: Combine different branches.
- **git branch**: Manage branches in your repository.

CI/CD

- **Continuous Integration (CI)**: Automatically testing your code after every commit to ensure new changes don't break the project.
- **Continuous Delivery (CD)**: Automating the deployment of tested code into production environments.

.gitlab-ci.yml File

- The `.gitlab-ci.yml` file is the configuration file that defines your CI/CD pipeline.
- Each pipeline consists of **jobs** with different stages (build, test, deploy), and these stages are executed in sequence.

GitLab Runner

- **GitLab Runner** is the application that processes the jobs defined in `.gitlab-ci.yml`.
- You can install GitLab Runner on a local machine, a server, or a container.

SSH Keys

- **SSH keys** allow secure access to GitLab repositories.
- They are used to authenticate Git operations and secure communication between your machine and the GitLab server.

Additional Tools Used for Code Review & Collaboration

- **Merge Requests**: Used for code reviews and merging changes into the main branch.
- **Code Snippets**: Share small, reusable blocks of code within a project or issue.
- **Discussions**: Inline comments on code, useful for peer reviews.

Snippets

- **Snippets** in GitLab allow you to share small sections of code with others.
- They can be public or private and help reuse code efficiently across multiple projects or teams.

Wikis

- **Wikis** in GitLab provide a space to store and organize project documentation.
- Each project or group can have its own wiki for managing internal guides, processes, and knowledge sharing.

Web IDE

- **GitLab Web IDE** allows you to edit files directly within your browser.
- It's a fully integrated development environment where you can edit code, commit changes, and submit merge requests without leaving the GitLab interface.

Docker

- **Docker** is a platform used to package applications and their dependencies into lightweight containers.
- GitLab supports Docker integration for building, testing, and deploying applications within CI/CD pipelines.

GitLab Security Scanning

- GitLab includes several security tools to scan your codebase for vulnerabilities, such as:
    - **SAST**: Static Application Security Testing.
    - **DAST**: Dynamic Application Security Testing.
    - **Container Scanning**: Analyzes Docker images for security vulnerabilities.

GitLab SAST Scanning

- **SAST (Static Application Security Testing)** scans your codebase for vulnerabilities before it's deployed.
- It checks for security issues in your source code and dependencies.
- Results are shown in merge requests, allowing developers to address issues before merging their code into the main branch.

This detailed breakdown should give you an in-depth understanding of each GitLab concept. Let me know if you'd like to explore any topic further!

4o

You said:
They need to be more in-depth

ChatGPT said:
ChatGPT

Absolutely! I'll dive deeper into each topic, elaborating on advanced features, use cases, and integration points in GitLab. Here's an even more detailed breakdown:

GitLab Organization Structure

- **Hierarchy Levels**:
    - **Instance Level**: This is the top-most level that manages the entire GitLab environment. It's often used in self-hosted GitLab setups and controlled by system administrators.
    - **Groups**: Represent a collection of related projects and users. Groups can be used to manage the roles and access controls for multiple projects.
    - **Subgroups**: Provide additional hierarchical layers under a group, allowing complex organizations to separate teams, departments, or even features within one parent group. Subgroups can inherit permissions from their parent group, simplifying user management.
    - **Projects**: These are individual repositories for your code. Each project can have its own CI/CD pipeline, issues, labels, and contributors, but project-level permissions can also be controlled by group membership.

Advanced Use Case: Large enterprises use multiple subgroups under a group to separate projects by departments (e.g., **Frontend Development**, **Backend Development**, and **Infrastructure**), but keep the overarching permissions inherited from the top-level group.

Epics

- **Scope and Purpose**:
    - Epics help break down significant business goals or large features into smaller, more manageable tasks. Each epic can contain multiple issues spanning multiple projects.

    - Epics are hierarchical in nature, meaning an epic can have sub-epics. This allows you to build complex project plans where an overarching goal is divided into smaller goals, each tracked individually.
    - **Cross-Project Epics**: Epics are not confined to a single project. They can include issues from different projects, which is critical for organizations that work on interrelated projects that must be coordinated.

Advanced Use Case: An engineering team working on a large product feature, such as "User Authentication Overhaul," would create an epic. Inside this epic, there would be issues for the UI/UX changes, backend changes, and security updates, each worked on by different teams and within different projects, but all tracked under one epic.

Groups

- **Detailed Permissions Model**:
    - Groups allow granular control over project access. You can assign roles such as **Guest**, **Reporter**, **Developer**, **Maintainer**, and **Owner** at the group level, controlling what actions each user can take.
    - Groups also allow shared **runners** (for CI/CD), **container registries**, and **artifact repositories** across projects.
    - **LDAP/SCIM Integration**: In enterprise setups, groups can be synced with external identity providers like LDAP or SCIM, ensuring that organizational roles automatically map to GitLab permissions.

Advanced Use Case: In large enterprises, where multiple teams contribute to different services, groups make it easy to create a secure and manageable organization. For example, the **Security Group** might have access to every project in the company to run vulnerability scans but not to modify code.

Sub Groups

- **Use Cases**:
    - Subgroups enable further categorization within a group, helping organize projects by teams, client departments, or technologies.
    - **Inheritance**: Subgroups can inherit the same permissions, pipelines, and CI configurations from their parent group. This can reduce administrative overhead and provide consistent security policies across a large enterprise.

Advanced Use Case: A large tech firm may create a top-level group for their **Product Team**, then split the team into **Mobile**, **Backend**, and **Frontend** subgroups. Each of these subgroups may have additional subgroups for specific technologies (e.g., **Android** and **iOS** in Mobile).

Projects

- **Advanced Features**:
    - **Merge Requests (MR)**: GitLab projects rely heavily on MRs for peer review and code collaboration. Each MR can include reviewers, assignees, and automated checks (e.g., linting or testing).
    - **CI/CD Pipelines**: Projects have integrated CI/CD pipelines defined in the `.gitlab-ci.yml` file. Each project can have its own pipeline setup to run tests, build artifacts, deploy to production, or trigger external systems.
    - **Protected Branches and Tags**: Projects can enforce branch protection rules to control who can merge or push code to specific branches, such as the `main` branch.

Advanced Use Case: A project could include automation tools like Docker to package applications or run tests using CI/CD. For instance, a **Machine Learning** project may run data preprocessing pipelines and training models using GitLab's CI features with GPU runners.

Issues

- **Advanced Features**:
    - **Issue Boards**: GitLab supports issue boards (Kanban-style) that allow you to visually track the status of your issues across customizable columns (e.g., **To Do**, **In Progress**, **Review**, **Done**). You can configure boards to automatically move issues based on their labels.
    - **Weight and Priority**: Issues can be weighted and assigned priority, which helps teams plan their workload and determine which issues to address first.
    - **Linked Issues**: GitLab allows issues to be linked to other issues or merge requests. This helps in tracking dependencies (i.e., one issue is blocked until another is completed).

Advanced Use Case: In agile development, issues are used to track tasks, bug fixes, and user stories. They are often linked to MRs, and issue boards can be used for sprint planning and tracking progress visually.

Participants

- **Roles and Permissions**:

    - Participants in issues or merge requests (MRs) can play roles like **Assignee**, **Reviewer**, or **Commenter**. They can actively take part in discussions, approvals, and code reviews.
    - Participants are notified about any activity, comments, or updates in the issues/merge requests where they are involved.

Advanced Use Case: For code reviews, each MR is assigned a participant as a **Reviewer** who must approve the changes. This ensures accountability and consistency across code quality.

Labels

- **Advanced Labeling**:
    - GitLab labels are customizable and color-coded to make it easy to filter and search issues or merge requests. You can set up **Scoped Labels** to limit labels to specific projects or groups.
    - **Label Prioritization**: You can assign labels such as `priority::high` to issues that need immediate attention, or labels like `feature` and `bug` to classify work.

Advanced Use Case: In a large development team, you can use labels for sprint planning, such as labeling issues with `sprint::week_1` and `sprint::week_2`. GitLab also allows bulk labeling to categorize multiple issues at once.

Milestones

- **Advanced Tracking**:
    - Milestones in GitLab are useful for tracking progress toward a specific goal or feature release. They can include multiple issues and merge requests from different projects, which allows cross-project tracking.
    - **Milestone Burndown Charts**: GitLab provides burndown charts that help track the progress of issues within a milestone, showing how many issues remain open.

Advanced Use Case: A product team might create milestones for **Version 1.0** of their product, associating all the features, bugs, and tasks that need to be completed for that version. The milestone gives a comprehensive view of progress across various teams.

Boards

- **Detailed Configuration**:

    - Boards allow you to visually track the flow of work with customizable columns representing different stages in your development process (e.g., **Backlog**, **In Progress**, **Code Review**, **QA**, **Done**).
    - Boards can be scoped to groups, projects, and even epics, giving teams flexibility in how they manage workflows.

Advanced Use Case: Agile teams use boards to manage sprint tasks. Each board could have columns for the stages of the workflow, allowing team members to move issues across columns as they progress.

GitLab Terms

- **Repository**: A repository contains your project's files, including code, documentation, and CI/CD configurations.
- **Merge Request (MR)**: A MR is a request to merge code changes from one branch to another. It includes inline code reviews, discussions, and tests.
- **CI/CD Pipeline**: A pipeline is a set of automated processes that GitLab CI/CD uses to build, test, and deploy your application.

Advanced Use Case: GitLab pipelines are used to automate the release process. For instance, upon a successful MR, a pipeline might automatically deploy code to a staging environment, run integration tests, and deploy to production.

Branch

- **Branching Strategies**:
    - GitLab supports several branching strategies, including **Git Flow** and **Trunk-based Development**.
    - In **Git Flow**, there are long-lived branches (e.g., `main` and `develop`) and short-lived feature branches. In **Trunk-based Development**, developers commit directly to the `main` branch, with small and frequent merges.

Advanced Use Case: In a continuous delivery pipeline, a `feature` branch might be created for every new feature. After development and testing, this branch is merged into the `develop` branch for final integration before deployment.

Tag

- **Advanced Tagging**:

    - Tags are often used for marking stable versions or releases. GitLab allows lightweight tags (created locally) and annotated tags (which store additional metadata).
    - Tags are helpful for software versioning (e.g., `v1.0`, `v2.1`).

Advanced Use Case: Before deploying to production, you might tag the last commit in your `main` branch as `v1.0.0`, creating a reference point for the code that can be easily accessed later.

Checkout

- **Advanced Use**:
    - `git checkout` is used for switching branches or restoring files. It can also be used to inspect older versions of your project.
    - You can use `git checkout -b <new-branch>` to create and switch to a new branch.

Advanced Use Case: Developers use `git checkout` to switch between multiple feature branches and hotfix branches, ensuring isolated development.

Merge

- **Merge Conflict Resolution**:
    - When two branches have conflicting changes, GitLab provides visual conflict resolution tools in the UI, guiding developers through resolving conflicts before merging.
    - **Fast-forward merges** vs. **three-way merges**: Fast-forward merges only move the head of the branch forward, while three-way merges create a new commit to combine changes.

Advanced Use Case: In complex projects, conflicts may arise when multiple teams work on overlapping files. GitLab's conflict resolution interface helps mitigate such issues.

Commit

- **Advanced Concepts**:
    - Commits are snapshots of your project at specific points in time. Each commit has a SHA-1 hash, which uniquely identifies it.

    - Best practices include writing meaningful commit messages, splitting large changes into multiple commits, and using **squash commits** to reduce the number of commits in MRs.

Advanced Use Case: A team might follow a **conventional commit** pattern (`feat:`, `fix:`, `chore:`) for better tracking of what each commit does, which can help automate versioning with tools like **semantic-release**.

Push

- **Git Push**:
    - `git push` updates the remote repository with commits made locally. You can push to different branches or even tags.
    - GitLab allows integration with CI/CD pipelines to automatically trigger builds, tests, or deployments on every push to a branch.

Advanced Use Case: When a developer pushes to the `main` branch, it can trigger an automatic build and deployment to a production environment, enabling rapid deployment workflows.

Workspace

- **Definition**: In GitLab, **Workspace** refers to the developer's working environment, which includes repositories, pipelines, issues, MRs, and additional features like Snippets and the Web IDE.
- Each workspace is fully integrated with GitLab's tools, enabling smooth collaboration and development processes.

Advanced Use Case: Developers can configure their workspace to automatically fetch the latest issues assigned to them or to display merge request updates as they happen.

Git Commands

- **Detailed Overview**:
    - **git fetch**: Downloads new data from the remote repository but does not integrate it with the local working directory.
    - **git rebase**: Allows integrating changes from one branch to another by replaying commits on top of the target branch.

    - **git cherry-pick**: Apply the changes introduced by an existing commit to a different branch.

Advanced Use Case: Developers use `git rebase` to keep a feature branch up to date with the latest changes in the main branch before merging, maintaining a clean history.

CI/CD

- **Advanced CI/CD Features**:
    - GitLab's CI/CD integrates deeply with Git repositories, allowing automatic testing, deployment, and monitoring of your applications.
    - **Triggers**: You can define triggers to run pipelines on specific events, like when new code is pushed or a merge request is created.
    - **Artifacts**: Each pipeline job can generate artifacts (e.g., binaries, reports) that can be stored and shared between jobs or retained for future use.
    - **Environments**: GitLab allows you to define environments like **staging**, **testing**, or **production** and associate them with specific pipeline stages.

Advanced Use Case: A continuous deployment pipeline can be set up where, after passing all tests in the CI stage, a deployment job automatically pushes code to a production environment using Kubernetes.

.gitlab-ci.yml File

- **In-depth Breakdown**:
    - **Stages**: Define the different parts of the pipeline (e.g., **build**, **test**, **deploy**).
    - **Jobs**: The specific tasks that are executed in each stage. Each job can have different **scripts**, **artifacts**, and **rules** to determine when they should run.
    - **Variables**: You can define environment variables at the job or pipeline level, which are used to configure the pipeline dynamically.
    - **Caching**: To speed up jobs, cache commonly used dependencies or build artifacts, ensuring that jobs reuse them without downloading or generating them repeatedly.

Advanced Use Case: In more complex pipelines, multiple environments (e.g., **dev**, **stage**, **prod**) can be configured, and the pipeline can run different jobs for each based on the branch being pushed to.

GitLab Runner

- **Advanced Features**:
    - GitLab Runner is a lightweight agent used to run jobs specified in the `.gitlab-ci.yml` file.
    - **Runner Types**:
        - **Shared Runners**: Available to all projects in a GitLab instance.
        - **Specific Runners**: Assigned to individual projects or groups.
    - **Autoscaling**: GitLab Runners can be configured to autoscale on cloud platforms like AWS or GCP to handle a large volume of jobs efficiently.

Advanced Use Case: For large teams, GitLab can deploy Runners in different environments (e.g., dedicated build runners on AWS and testing runners on on-prem servers) to speed up pipeline execution.

SSH Keys

- **Advanced Use**:
    - SSH keys are used for secure communication between your machine and GitLab, allowing operations like cloning repositories, pulling code, and pushing commits without needing to re-enter credentials.
    - SSH key types: GitLab supports both **RSA** and **ED25519** key pairs, with the latter being more secure.
    - SSH keys can be linked to your GitLab profile for repository access or used in CI pipelines to access external systems.

Advanced Use Case: You can set up **Deploy Keys** in GitLab, which allow a specific server or service to have read or write access to repositories securely, without needing user credentials.

Additional Tools Used for Code Review & Collaboration

- **Advanced Code Review**:
    - **Inline Code Comments**: During a code review in a merge request, users can add inline comments directly to specific lines of code. This is useful for pinpointing feedback or suggestions.
    - **Multiple Approvals**: GitLab supports requiring multiple approvers for merge requests, ensuring that code is reviewed by more than one person.
    - **CI Integration**: CI pipelines can be integrated with merge requests, allowing automatic tests to run and block merging if tests fail.

Advanced Use Case: Large development teams often require automated code analysis (e.g., **SonarQube** integration) in merge requests, adding another layer of quality control.

Snippets

- **Advanced Features**:
    - GitLab **Snippets** allow users to share small pieces of code, scripts, or configuration files with others.
    - Snippets can be either public, internal (visible to logged-in users), or private.
    - Snippets can be forked and commented on, making them useful for collaborative debugging or sharing solutions within the team.

Advanced Use Case: You can use snippets to share deployment scripts with your team. A snippet could contain the necessary bash script for automating Kubernetes deployments.

Wikis

- **Advanced Features**:
    - Each GitLab project or group can have its own **Wiki** to store documentation and internal guides.
    - Wikis support **Markdown** for formatting, and **version history** allows tracking changes to documentation over time.
    - GitLab Wikis are ideal for **internal knowledge sharing**, providing documentation on how to set up projects, standards for code, and processes.

Advanced Use Case: A **DevOps** team could maintain a centralized Wiki within GitLab to document all infrastructure setup processes, best practices, and CI/CD pipeline instructions.

Web IDE

- **Advanced Features**:
    - GitLab's **Web IDE** provides a fully integrated development environment directly in the browser, where you can edit files, commit changes, create branches, and even submit merge requests.
    - It offers **live previews** for changes (for web applications), and integration with the CI/CD system allows you to see pipeline results in the same interface.
    - The Web IDE also supports **Git** operations, like stashing changes and pushing commits.

Advanced Use Case: A quick bug fix or small edit to documentation can be done directly in the Web IDE without the need to clone the repository or use a local development environment.

Docker

- **Docker Integration with GitLab**:
    - Docker is a widely used platform for containerizing applications. In GitLab, Docker is often used in CI/CD pipelines to build, test, and deploy applications in a consistent environment.
    - GitLab supports **Docker Runners**, allowing jobs to be executed in isolated Docker containers.
    - **Container Registry**: GitLab provides a built-in container registry, where you can store and manage your Docker images alongside your code.

Advanced Use Case: In a microservices architecture, each service could be containerized and pushed to the GitLab container registry. The CI/CD pipeline could then deploy and run these services using Kubernetes.

GitLab Security Scanning

- **Advanced Security Features**:
    - GitLab provides several built-in security tools, such as **Static Application Security Testing (SAST)**, **Dependency Scanning**, and **Secret Detection**.
    - These tools are integrated into the CI/CD pipeline, allowing security vulnerabilities to be detected early in the development process.
    - **Dynamic Application Security Testing (DAST)**: This tool runs live security checks against running web applications, checking for common vulnerabilities like **SQL injection** or **Cross-Site Scripting (XSS)**.

Advanced Use Case: A project can automatically run SAST scans after every merge request, and GitLab will flag any vulnerable code before it gets merged into production.

GitLab SAST Scanning

- **Detailed SAST Overview**:
    - **SAST** scans the source code for known security vulnerabilities before it is compiled or executed. GitLab integrates this into CI/CD pipelines and reports issues found in the **merge request** view.
    - **Supported Languages**: GitLab SAST supports a wide variety of languages, including **JavaScript**, **Python**, **Ruby**, **Java**, **Go**, and more.
    - **Reporting**: Results from SAST scans are shown in merge requests, allowing developers to address security issues before the code is merged.

Advanced Use Case: A project using **Node.js** with dependencies could be automatically scanned for known vulnerabilities, and dependency updates can be flagged before going into production, ensuring the use of secure libraries.