Todo Web App with CI/CD PIPELINE

This main branch contains a Spring Boot To-Do application, along with configurations for its build, deployment, security, logging, monitoring, and CI/CD.

Here's a breakdown of the key components and their functionalities:

1. Application Core

- **`src/main/java/com/myTodoApp/MyFirstTodo/MyFirstTodoAppApplication.java`**: This is the main entry point of the Spring Boot application. It uses `@SpringBootApplication` to enable auto-configuration and component scanning.
- **`src/main/java/com/myTodoApp/MyFirstTodo/App/Todo.java`**: Defines the `Todo` entity, mapping it to a database table. It includes fields for `id`, `username`, `description`, `targetDate`, and `done` status. Validation rules, such as `description` having a minimum size of 10 characters, are also specified.
- **`src/main/java/com/myTodoApp/MyFirstTodo/App/TodoService.java`**: Provides business logic for managing `Todo` items. It includes methods for finding todos by username, adding, deleting, updating, and toggling the 'done' status of todos. This service initially uses a static list to simulate a database but also includes a `resetState()` method for testing purposes.
- **`src/main/java/com/myTodoApp/MyFirstTodo/App/TodoRepository.java`**: An interface that extends `JpaRepository`, providing standard CRUD operations for `Todo` entities and a custom method `findByUsername` to retrieve todos based on the username.
- **`src/main/java/com/myTodoApp/MyFirstTodo/App/TodoController.java`**: A Spring MVC controller that handles web requests related to To-Do items. It uses `TodoService` to perform operations like listing, adding, deleting, and updating todos. It also manages session attributes for the username.
- **`src/main/java/com/myTodoApp/MyFirstTodo/App/TodoControllerJpa.java`**: An alternative Spring MVC controller that uses `TodoRepository` (JPA) for persistence operations instead of the `TodoService`'s static list. It provides similar functionalities for managing todos.
- **`src/main/java/com/myTodoApp/MyFirstTodo/Login/WelcomeController.java`**: Handles requests for the application's welcome page. It retrieves the logged-in username and passes it to the `welcome.jsp` view.
- **`src/main/java/com/myTodoApp/MyFirstTodo/security/SpringSecurityConfiguration.java`**: Configures Spring Security for the application. It sets up in-memory user details with two users ("Ahmed" and "Abdul") and uses BCrypt for password encoding. It secures all URLs, displays a login form for unauthorized requests, disables CSRF, and allows frames for embedding (e.g., for H2-Console).

2. Web Layer (JSP)

The application uses JSP (JavaServer Pages) for its front-end views.

- **`src/main/webapp/WEB-INF/jsp/welcome.jsp`**: The welcome page, displaying a greeting to the logged-in user and a link to manage todos.
- **`src/main/webapp/WEB-INF/jsp/listTodos.jsp`**: Displays a list of todos in a table format, allowing users to view, delete, update, and toggle the completion status of tasks.

- **`src/main/webapp/WEB-INF/jsp/todo.jsp`**: A form for adding new todos or updating existing ones. It includes fields for `description` and `targetDate` and handles validation errors.
- **`src/main/webapp/WEB-INF/jsp/sayHello.jsp`**: Provides a standalone "Colorful To-Do List" interface, built with HTML, CSS, and JavaScript. It allows users to add, mark as done, and delete tasks directly within the page, without server-side persistence.
- **`src/main/webapp/WEB-INF/jsp/common/header.jspf`**: A JSP fragment defining the common HTML head section, including Bootstrap, Font Awesome, and datepicker CSS, along with a custom CSS file.
- **`src/main/webapp/WEB-INF/jsp/common/navigation.jspf`**: A JSP fragment for the application's navigation bar, including links to "Home" and "Todos," and a "Logout" option.
- **`src/main/webapp/WEB-INF/jsp/common/footer.jspf`**: A JSP fragment for the common footer section, including JavaScript libraries like Bootstrap, jQuery, and Bootstrap Datepicker.
- **`src/main/resources/static/custom/css/custom.css`**: Contains custom CSS rules for general styling, navbar, tables, buttons, and the welcome page, enhancing the application's visual appearance.

3. Build & Dependency Management

- **`pom.xml`**: The Maven Project Object Model file, defining the project's dependencies (e.g., Spring Boot starters, JPA, MySQL connector, Spring Security, WebJars for Bootstrap and jQuery), build plugins, and project metadata.
- **`mvnw`** and **`mvnw.cmd`**: These are Maven Wrapper scripts for Unix-like and Windows systems, respectively. They allow building the project without a pre-installed Maven, by automatically downloading the correct Maven version if it's not present.

4. Database Configuration

- **`src/main/resources/application.properties`**: Configures the Spring Boot application, including view prefixes/suffixes for JSPs, logging levels, date format, and MariaDB datasource settings (URL, username, password). It also specifies Hibernate DDL auto-update and shows SQL for debugging.
- **`src/main/resources/data.sql`**: SQL script to populate the `todo` table with initial data when the application starts, useful for development and testing.

5. Detailed Documentation: Docker & CI/CD Pipelines

This document provides an in-depth look at how Docker is used for containerization and how GitLab CI/CD automates the build, test, and deployment processes for the Spring Boot To-Do application.

1. Application Containerization (Dockerfile)

The `Dockerfile` in this project is responsible for creating a lightweight and reproducible Docker image for the Spring Boot To-Do application.

- **Base Image**: `FROM openjdk:17-jdk-slim`
  - This specifies the base image as OpenJDK 17 with a slim Debian Buster distribution, which is ideal for production environments due to its small size, reducing image layers and potential vulnerabilities.
- **Proxy Settings**:

  Dockerfile

  ```
  ARG http_proxy
  ARG https_proxy
  ARG no_proxy
  ENV HTTP_PROXY=$http_proxy
  ENV HTTPS_PROXY=$https_proxy
  ENV NO_PROXY=$no_proxy
  ```

  - These `ARG` instructions allow proxy details to be passed during the Docker image build process, enabling the image to download external dependencies even if the build environment is behind a corporate proxy. The `ENV` instructions then set these proxies as environment variables inside the container for runtime operations.
- **Working Directory**: `WORKDIR /app`
  - Sets the current working directory inside the container to `/app` for subsequent commands, organizing the application files.
- **System Dependencies and Proxy Configuration for `apt`**:

  Dockerfile

  ```
  RUN echo 'Acquire::http::Proxy "http://proxy.th-wildau.de:8080";' \
  > /etc/apt/apt.conf.d/01proxy && \
  apt-get update && apt-get install -y \
  curl \
  maven \
  --no-install-recommends && \
  rm -rf /var/lib/apt/lists/*
  ```

  - This critical step configures `apt` (Debian's package manager) to use the specified proxy for downloading system packages like `curl` (useful for network debugging) and `maven` (needed for building the application within the image).
  - `--no-install-recommends` reduces the installed package size, adhering to the "slim" base image principle.
  - `rm -rf /var/lib/apt/lists/*` cleans up `apt` cache to further minimize image size.
- **Copying Project Files**:

  Dockerfile

  ```
  COPY pom.xml /app/
  COPY src /app/src/
  ```

  - Copies the Maven Project Object Model (`pom.xml`) and the entire `src` directory (containing Java source code, resources, and JSPs) into the `/app` directory within the container.
- **Maven Proxy Settings (Inside Container)**:

  Dockerfile

```
RUN mkdir -p ~/.m2 && \
echo '<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
https://maven.apache.org/xsd/settings-1.0.0.xsd"> \
<proxies> \
<proxy> \
<id>th-wildau-proxy</id> \
<active>true</active> \
<protocol>http</protocol> \
<host>proxy.th-wildau.de</host> \
<port>8080</port> \
<nonProxyHosts>www.google.com|*.th-wildau.de|*.tfh-
wildau.de</nonProxyHosts> \
</proxy> \
</proxies> \
</settings>' > ~/.m2/settings.xml
```

  - This creates a `settings.xml` file for Maven within the container's `.m2` directory. This ensures that Maven itself, when run inside the container during the build phase, can also use the proxy to download project dependencies.
- **Maven Build**:

  Dockerfile

```
RUN mvn dependency:resolve
RUN mvn package -DskipTests
```

  - `mvn dependency:resolve` downloads all necessary project dependencies. This is done as a separate layer to leverage Docker's build cache: if only source code changes, this layer can be reused.
  - `mvn package -DskipTests` compiles the application and packages it into a JAR file, skipping tests to speed up the image build process (tests are run in a separate CI/CD stage).
- **Expose Port**: `EXPOSE 8080`
  - Informs Docker that the container listens on port `8080` at runtime.
- **Run Application**: `ENTRYPOINT ["java", "-jar", "target/MyFirstTodoApp-0.0.1-SNAPSHOT.jar"]`
  - Defines the command that runs when the container starts. It executes the Spring Boot JAR created in the `mvn package` step.

2. Local Development & Deployment Orchestration (Docker Compose)

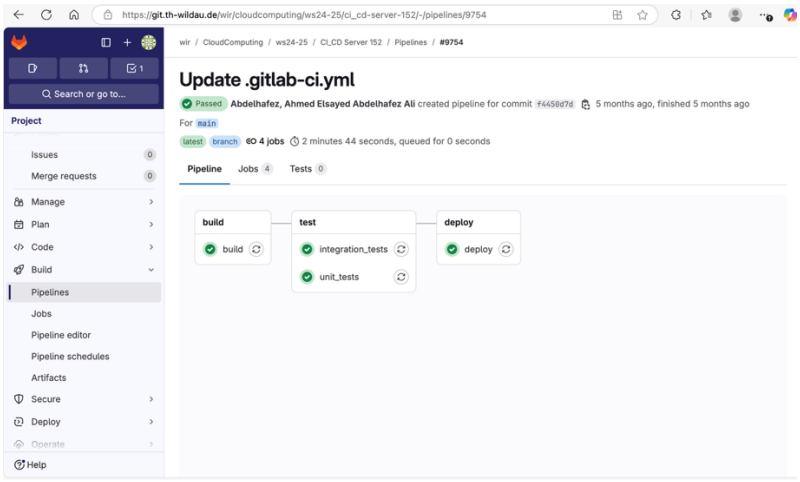
The `docker-compose.yml` file defines a multi-container Docker application environment, enabling easy setup and management of the application along with its dependencies (database, database administration tool).

- **Version**: `version: '3.8'`
  - Specifies the Docker Compose file format version.
- **Services**:
  - **`web` (Spring Boot Application)**:
    - `build`: Instructs Docker Compose to build the image for this service using the `Dockerfile` in the current context (`.`).
    - `args`: Passes proxy settings as build arguments to the `Dockerfile`.
    - `ports`: Maps host port `8080` to container port `8080`, allowing the application to be accessed from the host machine.
    - `environment`: Sets proxy environment variables inside the `web` container for runtime use, and specifies the MariaDB connection details (`MYSQL_ROOT_PASSWORD`, `MYSQL_DATABASE`, `MYSQL_USER`, `MYSQL_PASSWORD`).
    - `depends_on`: Ensures the `mariadb` service starts before the `web` service.
    - `networks`: Connects to the `app-network`.
  - **`mariadb` (MariaDB Database)**:
    - `image`: Uses the `mariadb:10.5` Docker image.
    - `container_name`: Assigns a specific name `mariadb` to the container.
    - `environment`: Sets up the root password, database name, and user credentials for MariaDB.
    - `ports`: Maps host port `3306` to container port `3306` for direct database access.
    - `volumes`: Mounts a named volume `db_data` to `/var/lib/mysql` for persistent storage of database files, ensuring data survives container restarts.
    - `networks`: Connects to the `app-network`.
    - `restart: always`: Configures the container to always restart if it stops.
  - **`phpmyadmin` (Database Management Tool)**:
    - `image`: Uses the `phpmyadmin/phpmyadmin` Docker image.
    - `container_name`: Assigns a specific name `phpmyadmin` to the container.
    - `environment`: Configures phpMyAdmin to connect to the `mariadb` service using the defined user and password.
    - `ports`: Maps host port `8081` to container port `80`, allowing access to phpMyAdmin via a web browser.
    - `networks`: Connects to the `app-network`.
- **Volumes**:
  - `db_data`: A named volume defined with `driver: local` to manage persistent data for the MariaDB service.
- **Networks**:
  - `app-network`: A bridge network that allows all services to communicate with each other using their service names (e.g., `web` can connect to `mariadb` using the hostname `mariadb`).

3. CI/CD Pipeline (`.gitlab-ci.yml`)

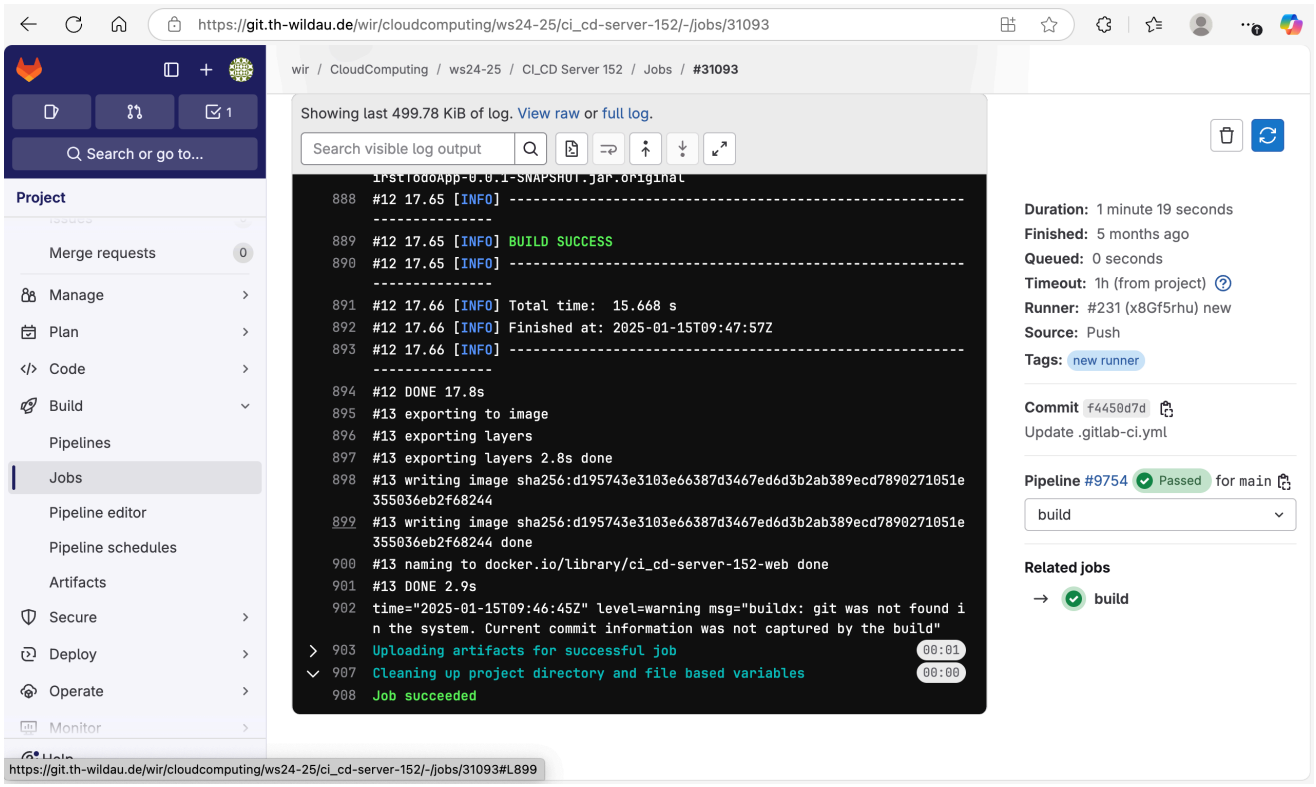
The `.gitlab-ci.yml` file defines the automated continuous integration and continuous deployment pipeline using GitLab CI. It integrates directly with Docker and Docker Compose to manage the application lifecycle.

- **`stages`**:
  - `build`: Compiles the application and builds Docker images.
  - `test`: Runs unit and integration tests.
  - `deploy`: Deploys the application using Docker Compose.
  - `cleanup`: (Currently commented out) For post-pipeline cleanup tasks.

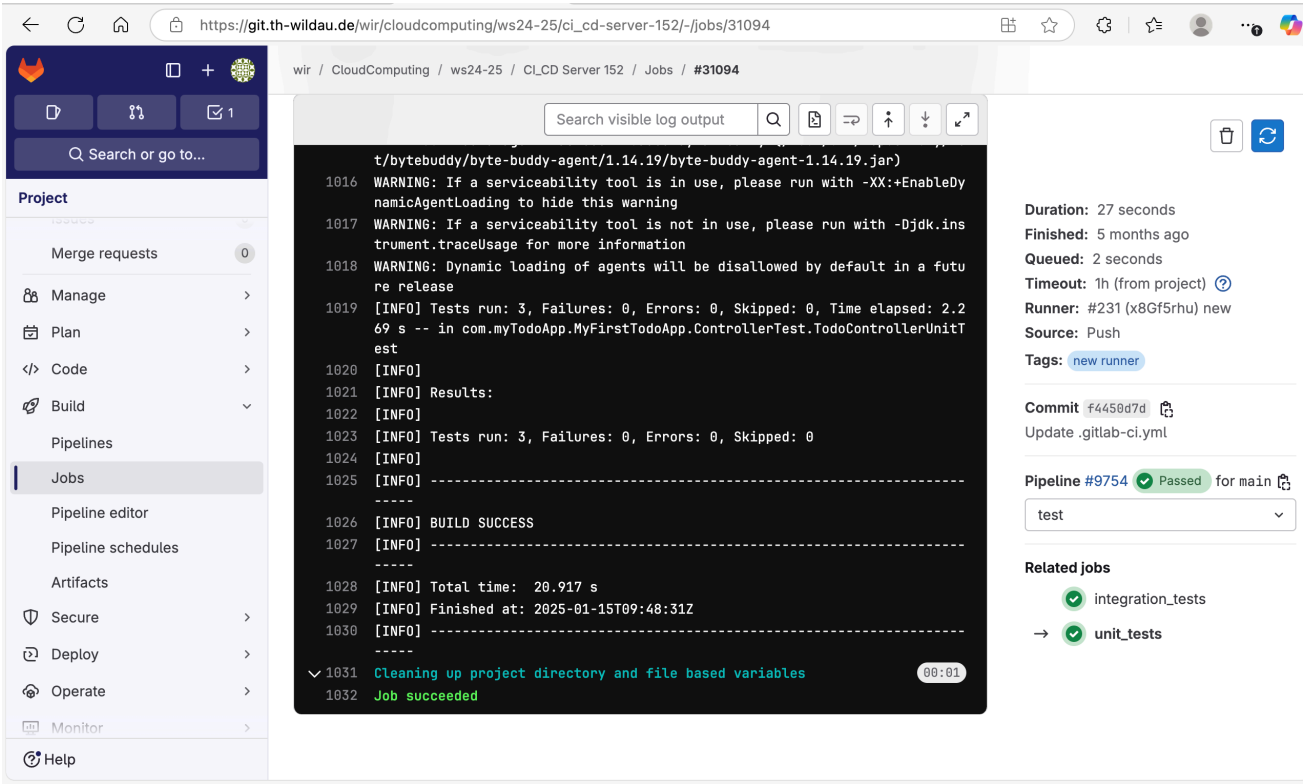


- **`before_script`**:
  - `mkdir -p ~/.m2`: Ensures the Maven local repository directory exists.
  - Sets up a Maven `settings.xml` with a proxy configuration for `th-wildau-proxy`. This allows Maven to download dependencies via the proxy during the build steps in CI.
- **`variables`**:
  - `HTTP_PROXY`, `HTTPS_PROXY`, `NO_PROXY`: Passed to Docker builds and other scripts within the pipeline to ensure network connectivity.
  - `DOCKER_IMAGE`: Specifies the full name and tag for the Docker image to be used for the application. This is typically used for `docker push` or `docker pull` if images are pre-built or pushed to a registry.
- **Global `image`**:
  - `name: maven:3.9.9-eclipse-temurin-23`: Sets the default Docker image for all jobs in the pipeline (unless overridden by a job-specific `image`). This provides a consistent Java and Maven environment for builds and tests.
- **Job Definitions**:
  - **`build` Stage Job**:
    - **`image: docker:20.10`**: Overrides the default Maven image to use a Docker image, necessary for executing Docker commands.
    - **`script: docker-compose build --no-cache`**: Executes `docker-compose build`. This command reads the `docker-compose.yml` and its referenced `Dockerfile` to build the application's Docker image (and any other service images defined for building, though here it's primarily the `web` service). `--no-cache` ensures a fresh build.

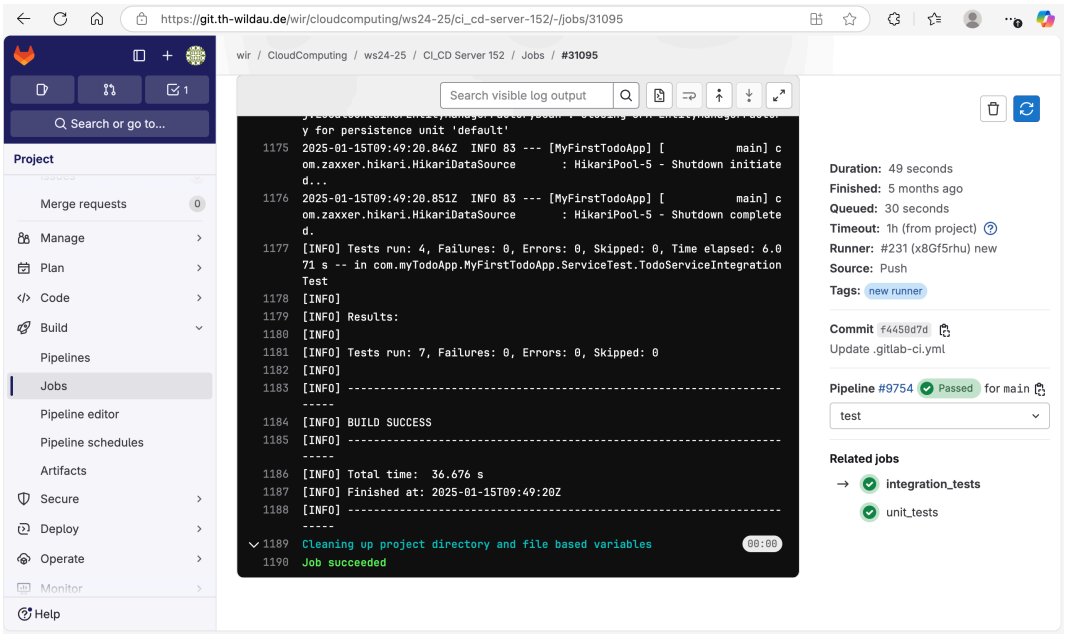
- **`artifacts`**: Specifies that the `target/*.jar` file should be saved as a pipeline artifact, making the compiled application available for later stages (e.g., if a deploy job needed to explicitly copy the JAR into a new image, though `docker-compose build` often encapsulates this).
- **`tags: new runner`**: Ensures this job runs on a GitLab runner with the specified tag.



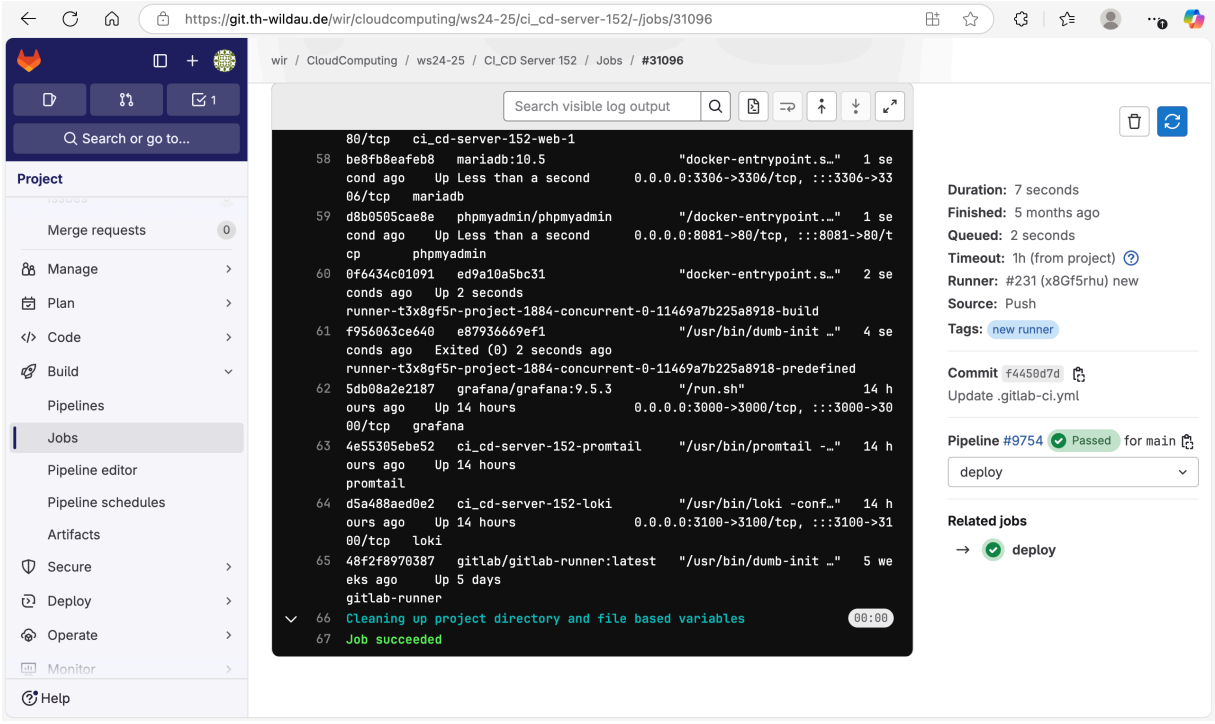
- **`test` Stage Jobs**:
  - **`unit_tests`**:
    - **`image: maven:3.9.9-eclipse-temurin-23`**: Uses a Maven image to run Java tests.
    - **`script: mvn test -Dtest=*UnitTest`**: Executes Maven tests, specifically targeting classes ending with `UnitTest`.
    - **`tags: new runner`**: Specifies runner tag.



- **`integration_tests`**:
  - **`image: maven:3.9.9-eclipse-temurin-23`**: Again, a Maven image for testing.
  - **`services: - name: mariadb:10.5 alias: mariadb`**: Crucially, this job declares a `mariadb` Docker service. GitLab CI will automatically start this MariaDB container for the duration of the test job, allowing integration tests to connect to a real database. The `alias: mariadb` ensures the application can connect using `jdbc:mysql://mariadb:3306/...`.
  - **`variables`**: Sets MariaDB credentials for the test database.
  - **`script`**:
    - Includes a `until bash -c 'echo > /dev/tcp/mariadb/3306'; do sleep 1; done` loop to wait for the MariaDB service to be fully up and listening for connections before running tests.
    - Executes `mvn test -Dtest=*IntegrationTest`, targeting integration test classes.
  - **`tags: new runner`**: Specifies runner tag.



- **`deploy` Stage Job**:
  - **`image: docker:20.10`**: Uses a Docker image to perform deployment operations.
  - **`script`**:
    - `docker-compose down || true`: Stops and removes any previously running Docker Compose services. The `|| true` prevents the job from failing if no containers are running.
    - `docker-compose up -d`: Starts all services defined in `docker-compose.yml` in detached mode, effectively deploying the application, database, and phpMyAdmin.
    - `docker ps -a`: Lists all Docker containers (running and stopped) for verification.
  - **`environment`**: Labels the deployment with a `production` environment and a URL (`http://10.100.8.152`).
  - **`only`**: This job is configured to run only when changes are pushed to the `main` branch or the `change-submit-button` branch, ensuring controlled deployments.
  - **`tags: new runner`**: Specifies runner tag.



- **`cleanup` Stage Job (Commented Out)**:
  - If enabled, this job would run `docker system prune -af --volumes` using an `alpine:3.18` image to remove all stopped containers, networks, images, and volumes, regardless of previous job failures (`when: always`). This is a good practice for keeping runner environments clean.