

Tampere University

COMP.SE.140 Continuous Development and Deployment - DevOps

Autumn 2025

Project end report

Teemu Salonen

06.12.2025

Contents

1	Instructions for the teaching assistant.....	1
1.1	Application credentials.....	1
1.2	Notes on testing the project.....	2
1.3	Optional features implemented.....	2
1.4	Details about the application	3
2	Description of the CI/CD pipeline.....	4
3	Reflections: main learnings and worst difficulties.....	6
4	Amount of effort used	7

1 Instructions for the teaching assistant

1.1 Application credentials

The application is hosted on a virtual machine with the following access:

- Monitoring console UI: XXX
- API: XXX

Note: The IP address of the VM is XXX but accessing the application using the plain IP will cause certificate warnings or errors because the TLS/SSL certificate is valid only for XXX.

To access the monitoring console UI and retrieve the API access token, use the following credentials:

- Username: admin
- Password: password123

Retrieving the Bearer JWT Token

The JWT token is required to authenticate API requests. Follow these steps to obtain it:

1. Open your browser and navigate to the monitoring service on port 8198 using the instructions above.
2. You will be prompted with a login page. Enter the credentials above.

3. Upon successful login, the application will provide a JWT token.
4. Token details:
 - Validity: 1 hour
 - Storage: The token is also saved in your browser's LocalStorage.

Storing tokens in LocalStorage was done for simplicity for this setup but has security drawbacks (like potential exposure to cross-site scripting attacks). Cookies would be a better solution.

1.2 Notes on testing the project

Based on the instructions, I assumed that the reviewer will deploy both versions of the application during testing and review. Therefore, the application is not currently live, although it was previously deployed to verify functionality.

To trigger the pipeline, an empty commit may be required. For example:

- `git commit --allow-empty -m "Trigger pipeline for version 1.0"`
- `git push origin project1.0`

1.3 Optional features implemented

The following additional features have been implemented beyond the minimum requirements:

- **Advanced monitoring**
 - The monitor service has been improved using the Docker API/SDK through the `dockerode` npm library [dockerode](#).
 - This implementation allows the monitor service UI to display CPU utilization for individual containers, memory usage for individual containers and The time elapsed since each container was last observed as "living".
- **Authentication and management control**
 - The application enforces authentication as described in section 1.1.
 - The monitor service provides users with a JWT token for API access. All HTTP requests to the API must include the following header: `--header "Authorization: Bearer <token>"`.
 - Note that this is a very basic JWT authentication and user details is stored in environment variables for simplicity.

- The JWT token is stored in the browser's LocalStorage, which makes it easily accessible for client-side operations. While convenient, this approach is not secure for production environments and is used here only for simplicity. The token is valid for 1 hour, meaning that even after logging out, the retrieved token can still be used to make API calls until it expires.

1.4 Details about the application

The application has the following characteristics, behaviors and limitations:

- **Version switching**

- The gateway determines the available stacks and identifies the currently active version by reading simple text files. It then appends the corresponding live suffix to the Docker service names, allowing it to route requests to the correct active container.
- While this approach is functional here, it is probably not the most robust solution for larger production environments.
- The UI make take a while to update to the new version (displaying minutes), especially in the case of the container details.

- **Version discarding**

- Old versions are discarded by the monitor service, with operations validated through the gateway. The monitor service already has Docker access via the dockerode library.
- Workflow for discarding an old version:
 - 1) An authenticated user clicks the "Discard Old" button in the UI.
 - 2) The request is sent to the API gateway, which adds a header: "x-discard-version: <version_to_remove>". The value corresponds to the version that is not currently live.
 - 3) The request is forwarded to the monitor service endpoint, which reads the header.
 - 4) The monitor service stops and removes all containers associated with the specified version.

- **Data fetching and UI updates**

- The monitor service fetches new data every 5 seconds.

- The UI updates at the same interval, which may cause slight delays in reflecting changes such as log sizes or container status.
- **Service dependencies**
 - Services have hardcoded URLs for internal communication. This means the application expects all required service versions to be deployed together successfully.
- **Container removal and UI lag**
 - Removing containers (e.g., discarding the old version) can be slow to reflect in the UI after clicking the discard-old button. It may take up to 30 seconds to get a response back and for the removed containers to disappear from the containers table, as it takes time for the actual container operations to complete.

2 Description of the CI/CD pipeline

As mentioned in section 1.2, the pipeline trigger may require an empty commit to activate, for example:

- `git commit --allow-empty -m "Trigger pipeline for version 1.1"`
- `git push origin project1.1`

Pipeline stages and tools

1. Build

- Uses `golang:1.25-alpine` image with necessary dependencies (Python, Node.js, Bash, Git).
- Builds the project services and saves artifacts for later stages (e.g., `service1/build_deps/`, `monitor/node_modules/`).

2. Test

- Runs very basic unit tests using the project test scripts.
- Ensures that built services are functioning correctly before packaging.

3. Package

- Uses Docker to build images for `service1`, `service2`, `storage`, `monitor`, and `api-gateway`.
- Images are tagged and pushed to the Harbor registry (`harbor.treok.eu/devops_project_nctesa`).

- After packaging, project images are cleaned up to reduce space usage.

4. Smoketest

- Pulls and starts all application containers using Docker Compose.
- Checks that each container is running, logging errors if startup fails.
- Stops and removes containers and cleans up images after testing.

5. Deploy

- Uses Ansible and SSH to deploy the application on the target VM injecting required environment variables.
- Prepares the VM by installing required packages and ensures Docker service is running.
- Sets up the application directory and files:
 - 1) Creates directories for the app and Nginx configuration/state.
 - 2) Copies docker-compose.yaml to the VM.
 - 3) Creates a .env file with all required environment variables, service tags, credentials, and JWT secret.
- Manages services:
 - 1) Logs into the Harbor registry and pulls the latest images.
 - 2) Starts or updates the application stack with: "docker compose up -d --no-recreate".
 - 3) Updates version metadata (live and available stacks) for proper version switching.
- Verifies deployment by listing running containers with their names, images, and status.
- Cleans up SSH keys after deployment for security.

Notes on pipeline behavior

- Only monitor, service1, and service2 are actively deployed for version 1.1. Other services remain unchanged as changes to them are not needed.
- Docker-in-Docker (docker:24-dind) is used to allow image building and container management within the CI runner.
- Environment variables manage service tags, registry credentials, and deployment configuration.

3 Reflections: main learnings and worst difficulties

Some learnings and things noticed:

- JWT storage: Storing the JWT token in LocalStorage is simple and convenient but not very secure. A more secure approach would be to use cookies.
- Container pull policy: The course GitLab runner does not seem to support the "if-not-present" pull policy for Docker containers. Only "always" policy is allowed. This means all containers must be pulled every time, even if they technically could exist locally.
- I could have implemented automatic HTTP-to-HTTPS redirection, as currently, requests to plain HTTP pages or API result in the error: "The plain HTTP request was sent to HTTPS port."
- The text files used to track the live API version, and the available versions are currently using loose restrictions, which could allow unauthorized users or processes to modify them, potentially causing version inconsistencies, downtime, or security issues in a real production application.

Main difficulties:

- Nginx lua configuration
 - o Error: unknown directive "lua_shared_dict".
 - o Cause: The default Docker image nginx:stable-alpine does not include Lua support.
 - o Solution: Switched to openresty/openresty:alpine which includes Lua.
- Code and testing issues
 - o Encountered basic issues with some test setups, which required debugging and minor fixes.
- Request forwarding through nginx
 - o Initially, requests from gateway to the API did not reach the backend services.
 - o Cause: The default OpenResty Nginx configuration blocked forwarding.
 - o Solution: Modified the Dockerfile to remove the default config and set up proper forwarding.
- DNS resolution for containers
 - o Requests could not resolve Docker container names.

- Solution: Added a DNS resolver configuration in Nginx to allow proper container name resolution.
- Version switching issues
 - Switching between application versions using the provided stack file did not work during local testing.
 - Cause: Lua parsing failed to recognize line breaks because the file was edited on Windows (CRLF vs. LF).
 - Solution: Converted line endings to LF to allow Lua to correctly parse and distinguish between versions.

4 Amount of effort used

- Rough estimated hours: 30