Veebirakenduse projekt (ITI0302)

Lab 3 — A pipeline that deploys on every push

Siim Rebane

Goal of today

- By the end of this lab:
  - Your assignment 1 repository has a `.gitlab-ci.yml` with three stages: build, dockerize, deploy
  - A push to `main` **builds the jar, pushes an image to Docker Hub, and restarts the container on your server** with no step done by hand
  - `http://<ip>/api/version` returns the **commit SHA** of what you pushed
  - Your first **merge request** with a green pipeline is merged
- The lecture 3 deck is the reference: every file and every command today is in it
- Stuck at any step → raise your hand

Before you start

- Lab 2 is done: your application runs on your server in a container, behind nginx, and `http://<ip>/api/<resource>` answers from your laptop
- `ubuntu` is in the `docker` group on the server (`docker ps` works without `sudo`)
- Your project builds on your laptop and is pushed to your **iti0302-2026** repository on gitlab.cs.taltech.ee
- You can log in to hub.docker.com, or you create the account now: it takes two minutes

Work in this order. Each step has a check; do not go on until the check passes.

Step 1: Docker Hub account and token

1. hub.docker.com → Sign up, or sign in. Note your **username**: it is the first half of every image name you push
2. app.docker.com → **Personal access tokens** in the left menu → Generate new token
   - Description: `iti0302-ci`
   - Access permissions: **Read & Write**
   - Expiry: choose a date after the end of the semester
3. Copy the token now. It is shown once

Keep the browser tab open; the token goes into GitLab in step 3.

Check: you have a username and a token that starts with `dckr_pat_`.

Step 2: a deploy key for the pipeline

On your laptop, a new key pair, only for the pipeline, no passphrase:

```
ssh-keygen -t ed25519 -C "gitlab-ci-deploy" -f ~/.ssh/iti0302-deploy -N ""
```

Public half to the server. On your laptop, print it and copy the whole line:

```
cat ~/.ssh/iti0302-deploy.pub
```

On the server, paste it as a new line at the end of the file that lists allowed keys, and save. Same on macOS, Linux and Windows:

```
nano ~/.ssh/authorized_keys
```

Check: logging in with the new key works and needs no password:

```
ssh -i ~/.ssh/iti0302-deploy ubuntu@<ip> hostname
```

Step 3: CI/CD variables in GitLab

In your repository: Settings → CI/CD → **Variables** → Add variable. Five of them:

Key	Value	Flags
`DOCKER_USERNAME`	your Docker Hub username	
`DOCKER_PASSWORD`	the token from step 1	**Mask variable** on
`DOCKER_IMAGE`	`<username>/iti0302-2026`	
`SERVER_IP`	your server's IP	
`SSH_PRIVATE_KEY`	contents of `~/.ssh/iti0302-deploy`, all lines including BEGIN and END	**Expand variable reference** off

- Leave **Protect variable** off on all five
- `cat ~/.ssh/iti0302-deploy` prints the private key; copy everything it prints

Check: the Variables table lists five keys; `DOCKER_PASSWORD` shows as masked.

Step 4: enable shared runners

Settings → CI/CD → **Runners** → under "Shared runners", turn on **Enable shared runners for this project**.

A new project has this off. Without it, every pipeline you create today sits at *pending* and never starts.

Check: the Runners page lists several available shared runners, among them "Generic runner on Taltech cloud with Docker executor".

Step 5: wire `/api/version` (1 of 3)

`application.properties`:

```
app.commit=${APP_COMMIT:local}
```

A new controller, `VersionController.java`, next to your other controllers:

```
@RestController
public class VersionController {

    @Value("${app.commit}")
    private String commit;

    @GetMapping("/api/version")
    public Map<String, String> version() {
        return Map.of("commit", commit);
    }
}
```

Check: run the application in IntelliJ; `http://localhost:8080/api/version` returns `{"commit":"local"}`.

Step 5: wire `/api/version` (2 of 3)

`Dockerfile`: two new lines after `FROM`, the rest as in lab 2:

```
FROM eclipse-temurin:25-jre-alpine
ARG COMMIT=local
ENV APP_COMMIT=$COMMIT
RUN addgroup -S app && adduser -S app -G app
WORKDIR /app
COPY app.jar app.jar
RUN chown app:app app.jar
USER app
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Check: `docker build --build-arg COMMIT=test -t t .` on your laptop, if Docker is installed there, then `docker run --rm -p 8080:8080 t` and `curl localhost:8080/api/version` returns `{"commit":"test"}`. No Docker on the laptop: the pipeline is the check.

Step 5: wire `/api/version` (3 of 3)

`compose.yaml`: the image now comes from Docker Hub, so `build: .` goes and `image:` changes:

```yaml
services:
  app:
    image: ${DOCKER_IMAGE}:latest
    container_name: app
    ports:
      - "8080:8080"
    restart: unless-stopped
```

`${DOCKER_IMAGE}` is filled in from a `.env` file that the deploy job writes on the server. Locally the file no longer builds anything; that is intended.

Step 6: `.gitlab-ci.yml`, part 1: stages and build

A file named exactly `.gitlab-ci.yml` in the repository root. Maven:

```yaml
stages:
  - build
  - dockerize
  - deploy

build:
  stage: build
  image: maven:3.9-eclipse-temurin-25
  script:
    - ./mvnw -B clean package -DskipTests
    - cp target/*.jar app.jar
  artifacts:
    paths:
      - app.jar
```

Step 6: build job for Gradle, and other Java versions

Gradle projects use this `build` job instead:

```
build:
  stage: build
  image: gradle:9-jdk25
  script:
    - chmod +x ./gradlew
    - ./gradlew clean bootJar
    - cp build/libs/*.jar app.jar
  artifacts:
    paths:
      - app.jar
```

If your project is on Java 21, the image is `maven:3.9-eclipse-temurin-21` or `gradle:9-jdk21`. The image's Java must not be older than the `java.version` in your project, or the build fails with `invalid target release`.

Step 6: `.gitlab-ci.yml`, part 2: dockerize

```yaml
dockerize:
  stage: dockerize
  image: docker:latest
  services:
    - docker:dind
  variables:
    DOCKER_TLS_CERTDIR: ""
  script:
    - echo "$DOCKER_PASSWORD" | docker login -u "$DOCKER_USERNAME" --password-stdin
    - docker build --build-arg COMMIT=$CI_COMMIT_SHORT_SHA -t $DOCKER_IMAGE:$CI_COMMIT_SHORT_SHA -t $DOCKER_IMAGE:latest .
    - docker push $DOCKER_IMAGE:$CI_COMMIT_SHORT_SHA
    - docker push $DOCKER_IMAGE:latest
  only:
    - main
```

Copy it as is. The only project-specific value, `DOCKER_IMAGE`, comes from the variable. `only: main` means a feature branch does not push images; only `main` does.

Step 6: `.gitlab-ci.yml`, part 3: deploy

```
deploy:
  stage: deploy
  image: alpine:latest
  before_script:
    - apk add --no-cache openssh-client
    - eval $(ssh-agent -s)
    - echo "$SSH_PRIVATE_KEY" | tr -d '\r' | ssh-add -
    - mkdir -p ~/.ssh && chmod 700 ~/.ssh
  script:
    - scp -o StrictHostKeyChecking=no compose.yaml ubuntu@$SERVER_IP:/opt/app/compose.yaml
    - ssh -o StrictHostKeyChecking=no ubuntu@$SERVER_IP "cd /opt/app && echo DOCKER_IMAGE=$DOCKER_IMAGE > .env && docker compose pull && docker compose up -d"
  only:
    - main
```

Same: copy as is. The server, the key and the image name all come from variables. The job writes into `/opt/app`, the conventional place for an application on a Linux server. Create it once, on the server, owned by the user the pipeline logs in as:

```
sudo mkdir -p /opt/app && sudo chown ubuntu:ubuntu /opt/app
```

Check: `ls -ld /opt/app` shows `ubuntu ubuntu`. The `~/app` from lab 2 can stay or go; the pipeline never looks there.

Step 7: branch, push, watch

Not on `main`. A feature branch, as the course rule says from this week:

```
git checkout -b ci-pipeline
git add .gitlab-ci.yml Dockerfile compose.yaml src/
git commit -m "Add CI pipeline and /api/version"
git push -u origin ci-pipeline
```

In GitLab: Build → **Pipelines**. A pipeline appears for your commit with three jobs. Click a job to see its log live.

Check: the pipeline has one job, `build`, and it is green. There is no `dockerize` or `deploy` job, because the branch is not `main`.

If `build` is red, read its log from the first `ERROR` line. The lecture deck's table of log messages covers the usual ones.

Step 8: merge request, merge, deploy

1. GitLab shows "Create merge request" for your branch. Create it, source `ci-pipeline`, target `main`
2. The MR page shows the pipeline for the branch. Wait until it is green
3. **Merge**
4. Build → Pipelines: a new pipeline runs, for `main`. This time all three jobs run; on hub.docker.com your repository `iti0302-2026` appears, with a tag named after the merge commit

Real timing on the TalTech runners: about two minutes from merge to the container restarted.

Check:

```
curl http://<ip>/api/version
```

returns the short SHA of the merge commit, which is the one at the top of `main` in GitLab. `{"commit":"local"}` means the old container is still running: look at the `deploy` job's log.

Step 9: the rule that stays

Settings → Merge requests → **Merge checks** → tick **Pipelines must succeed**. Leave "Skipped pipelines are considered successful" unticked. Save.

From now on GitLab refuses to merge an MR whose pipeline is red. This is the course rule from week 3 on, and now your repository enforces it for you.

Check: the setting is saved. Then one more change on a new branch, MR, green, merge, and `/api/version` changes again. That second round is exactly what the assignment 1 check does after the deadline.

If you are done

- Add tests to the pipeline: remove `-DskipTests` (or run `./gradlew test`) and see a failing test turn the pipeline red before anything is deployed

Before you submit: the checker

The assignment is checked by a script against your URL. The same script is available to you now, at any time before the deadline:

https://iti0302.digivader.com/a1-check/

Enter your student code and the URL where your application answers, for example `http://193.40.157.99`. It calls every endpoint the assignment lists, in order, shows what it expected and what came back, and deletes what it created. Check as often as you like.

Check: all nine steps green, and the commit it reports is the one at the top of `main`.

Common mistakes today

Symptom	Cause
Pipeline *pending* forever	shared runners not enabled (step 4)
`denied: requested access to the resource is denied`	`DOCKER_IMAGE` does not start with your Docker Hub username
`unauthorized: incorrect username or password`	token pasted with a space, or `DOCKER_USERNAME` is your e-mail instead of the username
`Permission denied (publickey)` in deploy	the public key line is not in `authorized_keys`, or `SSH_PRIVATE_KEY` is not the whole file
`scp: /opt/app/compose.yaml: Permission denied`	`/opt/app` is missing or not owned by `ubuntu`; the `sudo mkdir` + `chown` line in step 6
`error in libcrypto` in deploy	"Expand variable reference" left on for the key
deploy green, `/api/version` still `local`	`ARG` / `ENV` lines missing from the Dockerfile, or `--build-arg` missing
deploy skipped on `main`	`only:` has a typo, or the default branch is `master`; check Settings → Repository



Questions