November 13, 2025

A1: Versions, Releases & Containerization

This assignment follows the process that has been laid out in the *assessment design* document. Please create the required organization and your repositories and prepare the submission document that contains links to everything.

1 Assignment

The assignments in this course start from a basic ML project and will lead you through the steps to release and operate a web application that represents a *sensible use case* for the project.

Run the Base Project: We will use the →SMS Checker project as a running example throughout the lectures. As is, the application only has rudimentary features and provides a minimalistic frontend that is connected to a trivial backend. Try to get the application working locally by following the instructions provided in the repository.

Figure 1 provides an overview over the current architecture and the extensions that you are supposed to introduce throughout the course. The current architecture has three components, a HTML/JS page that is served through →Spring Boot. The application has an API gateway, which delegates the REST request from the website to the model service. This is required to prevent issues caused by →cross site scripting protection. All other elements that are shown in the figure have to be built as an extension.

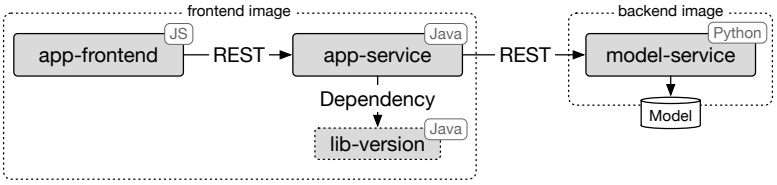

Figure 1: Reference Architecture

You are supposed to mature the application. All components should become independent and only interact with each other through reuse of dependencies or via REST APIs. To operate the application, the `app` and the `model service` should be released as two container images and it must be possible to deploy them separately from each other. You need to create and release a version-aware package, `lib-version`, that is used by the `app-service`. It has no specific functionality, but you need to show that you understand how to release and reuse libraries.

Create a Team Organization: The assignments in this course will bring together very different implementation aspects and operate them together. To make it easier to find and trace all moving pieces, create an open-source →organization called `doda<year>-team<num>` on GitHub (e.g., `doda25-team12`) and create the following repositories.

- `model-service`
- `lib-version`
- `app` (if desired, could also be split into two repositories `app-frontend` an `app-service`)
- `operation` (also include the *documentation* of assignment 4 here)

The `operation` repository should be the main entrypoint into the project. Add a `README.md` to the repository that introduces your high-level architecture and links to the other corresponding repositories, so visitors can easily understand your project and find all relevant information.

> Info
> Make sure that all repositories are public, so your peers and examiners can access your results.

The contents of all repositories will evolve throughout the course when you work on the different assignments. Future assignments might require you to iterate over your current status results and refine components as needed. At the end of the course, your organization should have a consistent state that reflects all assignments.

Extend the Application: In this assignment, you are supposed to extend the current application and start maturing the release engineering practices. This will help us later, when we will focus more on the operation of the application. In the following, you will find various feature requests for the application. Please note that we are not assessing the complexity, creativity, or layout of the application, we only focus on engineering aspects.

> Info
> You are allowed to migrate the client application to, for example, →Angular JS. However, the `app-frontend` must only communicate with a dedicated `app-service` API and not with the `model-service` directly.

■ *F1: Create a Version-aware Library:* Add a *version-aware* Maven library `lib-version` to your organization. The library should contain a class `VersionUtil` that can be asked for its version. The implementation should parse meta-data that is included in the package or explicitly store the version in a separate resource file. The `app` should depend on the library and use the `VersionUtil` somewhere. This feature request has two purposes: a) it allows you to illustrate that you understand how to release and reuse a library and b) the awareness about the version can prove useful later, for example, to add meaningful system information during the monitoring.

> Info
> Do not rely on parsing data from the version-control system, like tags, when determining the version during runtime, as it might not always be available when the library is reused.

■ *F2: Library Release:* A workflow is used to automatically package and version `lib-version` and to release it in the GitHub package registry for Maven.

> Caution
> Do *NOT* release your projects to *Maven Central*! Do not pollute release repositories with test releases!

■ *F3: Containerize both Frontend and Backend:* Create a `Dockerfile`s for the `app` and the `model-service` that allows to build two container images that could be used together to run your app.

■ *F4: Multi-architecture Containers:* Your released container images →support multiple architectures, at the least, they support both `amd64` and `arm64`.

■ *F5: Multi-stage Images:* At least one of your Dockerfiles illustrates how to use →multiple stages to reduce image size, for example, by avoiding to include the apt cache in the image.

■ *F6: Flexible Containers:* It should be possible to indicate the URL of the `model-service` as an ENV parameter for the `app` image. For both containers, it should be possible to define the port on which each service runs through an ENV variable, with a default of 8080 (`app`) and 8081 (`model-service`), if undefined.

■ *F7: Docker Compose Operation:* The `operation` repository contains all information about running your application and operating the cluster. For now, add a docker compose file that allows to start your application and a `README.md`. We later add all scripts for provisioning and deployment into this repository as well (Vagrant, Ansible, Docker Compose, Kubernetes, etc.). The Docker compose file does not need to grow with the later complexity of the Kubernetes deployment, but *it must stay runnable* with the latest image versions. The `README.md` should provide a *concise* documentation that includes an explanation on how to start the application (e.g., parameters, variables, requirements). It should also provide pointers to relevant files that help outsiders understand the code base.

■ *F8: Automated container image releases:* The `app` and `model-service` repositories contain workflows to package, version, and release the corresponding container images in the GitHub registry. The version is automatically determined by the workflow. Basic workflows are triggered through a Git version tag, more advanced solutions store the version information in meta-data (e.g., in the `pom.xml`) and find a good way to trigger the workflow.

> Info
> For all versioning tasks of the assignment, it is assumed that there is always only a *single source of truth*, i.e., a developer has to specify a new versions only in exactly one location. The version might be relevant in multiple places (e.g., in additional resource files), but these will be automatically propagated by the workflow.

■ *F9: Automated training and release of new models:* The `model-service` repository has a dedicated workflow for training, versioning, and releasing a new model. The resulting files are released in an public, *accessible* location, e.g., as attachments to a GitHub release, so they can be downloaded without authentication. It is sufficient if that workflow is manually triggered on demand.

Info
For example, you could explore →gh to understand how to create releases or attach files from the CLI.

■ *F10: No hard-coded model in model-service:* The `model-service` does not contain a hard-coded version of the model. The container should use a model that is provided to the container through a volume mount, or, if none has been provided, once download a specified model *on start* to this volume mount.

■ *F11: Container image pre-releases:* An advanced versioning concept is introduced in the `app` and `model-service` repositories. Releases on `main` should still follow the same logic of before (e.g., stable release on version tag), however, the repository should always point to a pre-release version like `1.2.3-SNAPSHOT` or `1.2.3-pre`. The workflow should release a stable version and then update the version to the next pre-release version afterwards.

For example, assume that the repository contains the current pre-release version `1.2.3-SNAPSHOT`, a release workflow on `main` should bump the version to `1.2.3`, release the version, and then bump the version again to `1.2.4-SNAPSHOT`, such that the same stable version is not accidentally released a second time.

In branches, the new release and versioning concept should release pre-releases for every successful build. These pre-releases should be strictly incrementally numbered (e.g., using a timestamp) and contain the branch name, for example, `app-1.2.3-251112-124753-new-feature`.

Info
All stable releases of all your artifacts must be properly and automatically versioned. However, the advanced versioning concept must only be implemented for the releases of the containers for *both* `app` and `model-service`.

Caution
You are not allowed to use existing GitHub actions like →release-please or →action-gh-release, as they trivialize the assignment, while hiding the technical details that you are supposed to understand.

2 Submission

In this course, you will use a Git organization with several repository to collaborate within your team. This assignment will be assessed by checking your team repositories and the interactions on GitHub. To register your a submission, you must submit a *submission document* to the REMLA course on →Peer. The detailed assessment process is described in the *assessment overview* document that you can find on Brightspace.

Info
Please note that we will also check individual contributions to validate that all team members have passed the knock-out criteria of the course.

3 Assessment

This assignment needs to comply with the *Basic Requirements* as laid out in the *assessment design* document. Each of the following subsections presents a partial grade, which ranges from *Insufficient* (1) to *Excellent* (10), representing the weighted average of all corresponding rubric items. Grades can be affected by a *Pass/Fail* in the *Data Availability* (see Assignment 1). If information is unavailable or unclear, the respective grades will be *Insufficient*.

Positive rubric results are defined incrementally: Grades higher than level N can only be achieved when all items of level N have been achieved. Partial completions will be graded by considering the state of the lowest incomplete level. For example, a full completion of *Sufficient* and partial completions of *Good* will end up in the range between 6-8. Any contributions on the *Excellent* level are ignored, until *Good* has been fully completed. In contrast to positive results, we only explain by example which delivered state of a rubric item we will consider as *Insufficient* and *Poor*.

3.1 Basic Requirements

Caution
The following requirements apply to all assignments/rubrics of the course. If any rubric item cannot be graded, e.g., because information is missing or because grading criteria cannot be applied to a deviation from the proposed reference architecture, the corresponding rubric items will be assessed as *Insufficient*.

Data Availability:

Pass The structure of the GitHub organization follows the requested template and all relevant information is accessible. The `operation` repository contains a `README.md` file the provides all necessary information to find and start the application.

Fail The relevant information cannot be found in the expected location. The `README.md` does not provide sufficient information to deploy the application.

3.2 Graded Requirements

Automated Versioning & Releases:

Insufficient
- Some artifacts of the project are not versioned or shared in an artifact registry.
- The library is released on public repositories like Maven Central.

Poor
- *All* artifacts (`model service`/`app` *images*, `lib-version` *package*) are released, but the release is not automated in a GitHub workflow.

Sufficient
- *All* artifacts of the project are versioned and shared in GitHub package registries of the organization.
  Some programming languages (e.g., Python, Rust) support consuming dependencies directly from Git repositories. Such a solution is acceptable, when the release is stable, i.e., dependents reference a version and not just a branch.
- The packaging and releases of all artifacts are automated in workflows.
- All artifacts have a *single source of truth* for their versioning.
- Releases are automatically versioned through a release tag like `v1.2.3` that triggered the workflow.

Good
- After a stable release, `main` is set to the next pre-release version.
  Automatic increase of *Patch* versions, bumps to *minor* or *major* versions can remain manual.
- The version is maintained in project meta-data and *not* taken from, for example, a Git tag.
- The `model-service` has a manually triggered workflow to release a new model as an attachment to a GitHub release.

Excellent
- The versioning strategy of the container images supports releasing multiple versions of the same pre-release, tied to a branch or pull request.
- The release workflows create a new commit and tag it with a version. The version meta-data that is contained in this version-tagged commit is consistent with the version tag.

Software Reuse in Libraries:

Insufficient
- The `lib-version` is not included as external dependencies.

Poor
- The `lib-version` is reused in `app` through a package manager, but is referenced through an unstable release (e.g., a reference to a SNAPSHOT release).

Sufficient
- The `app` reuses a *stable version* of `lib-version` as an external dependencies through a regular package manager.

Good
- Runtime dependencies in all projects are defined via package managers (e.g., `pom.xml` or `requirements.txt`) and not just installed when building the container images.

Excellent
- The version string in `lib-version` is automatically updated with the actual package version in the release workflow, i.e., either it is directly taken from automatically updated project metadata, or from an automatically generated file.

Containers & Orchestration:

Insufficient
- No `docker-compose.yml` exists.
- The communication between the components is broken and the app does not work.

Poor
- The `docker-compose.yml` file contains a serious configuration attempt, but the startup fails or the application does not fully work.
- Services other than `app-service` are exposed/accessible on `localhost`.
- Container images are built on-the-fly and not downloaded from a registry.

Sufficient
- The operation repository contains a `docker-compose.yml` to start up the application and use it.
- The `app-service` is the only service that is accessible from the host.
- The `app` has an ENV variable that defines the location (DNS name, port) of the `model-service`.
- The Docker compose file uses a volume mapping, a port mapping, and an environment variable.

Good
- The compose file works with the same images as the final Kubernetes deployment does.
  Make sure to update the `docker-compose.yml` at the end of the project.
- The listening ports of `app-service` and `model-service` can be configured through ENV variables.
- The exposed port on `localhost` can be freely adjusted, by just changing the `docker-compose.yml`.
- The model is not part of the container image, i.e., it can be updated without creating a new image by refering to a *specific* model version that is downloaded on-start.
- A local cache is used so the model is not just downloaded on every container start.

Excellent
- All released container images →support multiple architectures, at least `amd64` and `arm64`.
- At least one Dockerfile uses →multiple stages.
- Services have a →restart policy.
- The deployment uses an →environment file, e.g., to configure names/versions of images.