

# Enterprise Programming 1

## Lesson 11: CI and Deployment

Prof. Andrea Arcuri

# About these slides

- These slides are just high level overviews of the topics covered in class
- The details are directly in the code comments on the Git repository

# Continuous Integration (CI)

# Code Evolution

- Every time there is a change in the code, you want to know if application is still working fine
- Possible problems:
  1. Code does not compile
  2. Changes broke some functionalities, and some tests now fail
- When to check? At each *Git Push*
- Could ask developers to always do a “*mvn clean verify*” before each commit, but:
  1. They might forget
  2. Test cases might take **hours** to run

# CI Server

- A server that automatically pulls from Git at each push
- Build your application and run all tests
- Inform developers (eg by email) if a build fails
- Can keep track of build history
- Can check a Git PR (Pull Request) before merging it
- *Jenkins* is the most used CI server, which you can install on your machines
- Extremely useful when working in teams
  - If you end up working in a company not using CI, **run away!!!**

# CI Providers

- There are some CI providers, that are *free* for *open-source* projects
  - GitHub Actions, CircleCI, Travis (until 2020), etc.
- **GitHub Actions**
  - need to activate it from GUI of repository
  - need configuration file, e.g., *.github/workflows/ci.yml*

Workflows

New workflow

All workflows

CI



Tell us how to make GitHub Actions work better for you with three quick questions.

Give feedback



All workflows

Filter workflows

5 results		Event ▾	Status ▾	Branch ▾	Actor ▾
	<b>renamed security module</b> CI #5: Commit e21ad8e pushed by arcuri82	spring-2021	6 minutes ago In progress	...	
	<b>updated some dependencies</b> CI #4: Commit fcc2d83 pushed by arcuri82	spring-2021	14 minutes ago In progress	...	
	<b>getting rid off Travis</b> CI #3: Commit d4f6d5c pushed by arcuri82	spring-2021	8 hours ago 33m 18s	...	
	<b>configuring NodeJS in CI</b>	spring-2021	9 hours ago	...	

# Testing, Security and Development of Enterprise Systems



This repository contains a set of examples related to the testing, security and development of enterprise systems. Currently, this repository focuses on Java/Kotlin, targeting frameworks like Spring and Java EE.

The material in this repository is used in two university-level courses at the university college [Høyskolen Kristiania](#). In particular:

Can add “badge”  
to show current  
status of the  
build





# Database Maintenance

# Database Migrations

- So far, by configuring “*create-drop*” in Hibernate, we were always recreating the schema of the database
- This of course does not work in production... you do not want to delete your database at each application restart!!!
- A possible (but not good) solution is to use “*update*”
  - It will create a database (based on your entities) if not existing, otherwise will try to update the current one

# Issues with “update”

- What if you are adding a new column in a *@Entity*?
- What if you are changing the schema by refactoring some *@Entity* classes (e.g., split one in two)?
- What if by mistake/bug some *@Entity* classes are deleted?
- *What will happen to the current rows in such tables in the database?*

# Solution

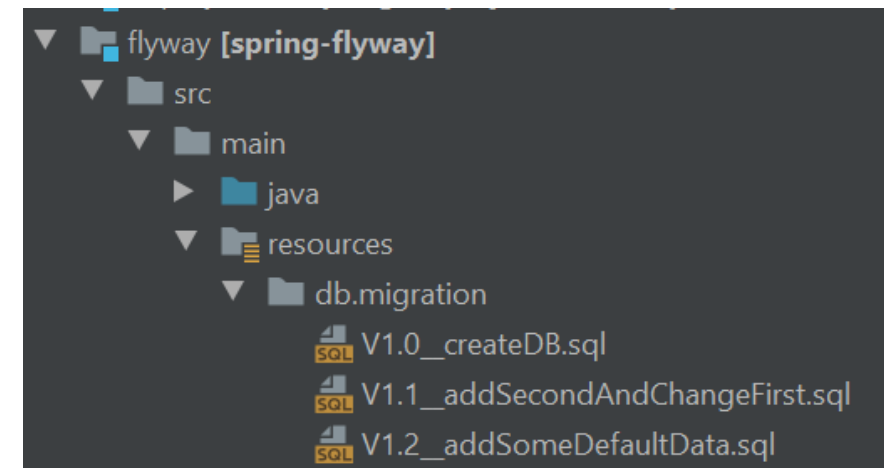
- The evolution of a database has to be handled with special tools
- *@Entity* classes should just map what is currently in the database, not driving its schema creation/update
  - apart from the very beginning before doing a first production release
- No “*create-drop*”, nor “*update*”, but rather “**validate**”
  - throw exception if *@Entity* classes do not match what in database schema
- Tools to use: *Flyway* or *Liquibase*

# Flyway

- All operations are done on SQL files, by writing SQL commands
- Each migration file has a version number, in increasing order
- *Flyway* will check if any new migration has not been applied yet, and apply them otherwise, just *once*
- It creates its own table to keep track of which migrations have been applied so far on a database
- SpringBoot *automatically* runs *Flyway* if found on classpath

# Folder db/migration

- Migration files in the form *Vx.y\_someName.sql*
- Executed in version order, eg *V1.0* executed before *V1.1*
- Recall that IntelliJ will represent folders inside **/resources** as packages
  - eg, there is no “*db.migration*” folder, but rather “*db/migration*”



Logging

# Log Statements

- It is important to keep track of what is going on in an application
- Especially important when there are bugs, and you want to save the stack-trace of the exceptions
  - needed for example to help debugging such possible bugs
- Logging is a bit tricky, and so there are several libraries that help doing it
  - e.g., because doing I/O, it can impact the JIT compiler... and you could have optimizations in which a logging framework writes on a buffer, and then another thread reads from it to do the I/O...



# SLF4J and Logback

- *SLF4J* is the most common library for logging in Java
- It is a “*facade*”, i.e., set of base classes/interfaces, where the actual implementation is in a different library, and it is abstracted away
  - i.e., you only import classes from SLF4J in your code
- Can use different logging framework bindings for SLF4J, where *Logback* is the most popular
  - for example, think of *SLF4J* as *JPA*, and *Logback* as *Hibernate*
- Many third-party libraries use SLF4J, but will not provide a binding
  - this enables you to have a single binding for your whole app, including the 3<sup>rd</sup>-party libraries

# Loggers

- Usually creating one logger per class, named with the full name of the class itself (including the package)
  - created using *LoggerFactory.getLogger(name)*, and typically stored in a final static variable
- Configurations will be in a *logback.xml* file
- For testing, can have a different *logback-test.xml* file which will have precedence
- Many possible configurations
  - eg, what to do with the logs? Should just be on the console? Should be written to a file? Should be sent to a remote server? Etc.

# Log Statements and Levels

- Different methods: eg, **log.debug(msg)** and **log.error(msg)**
- Based on the LEVEL of logging, some messages can be discarded
  - for example, you could tell the system to discard DEBUG logs, have WARN only on console, and ERROR on console and also saved on a file
- Levels: **TRACE, DEBUG, INFO, WARN** and **ERROR**
- Those are in order: when you activate a level, all levels above it are activated as well
  - eg, activating DEBUG does activate everything but TRACE

# Setting Log Levels

- Levels can be fine-tuned
- You can have a log-level for the whole application, e.g. typically WARN or ERROR
- Then, can override the level for some specific loggers
  - e.g., you could put it to INFO for your classes, but not the ones of the 3<sup>rd</sup>-party libraries
- When testing/debugging some classes, you could put DEBUG for just those

# String Concatenation

- Consider: *log.debug("" + x + "=" + y)*
- That would be bad: often debug-level logs are ignored (especially in production), and so computing *"" + x + "=" + y* would be a total waste of CPU cycles
- String concatenation is **expensive**: recall that Strings are immutable, and at each *+* we create a completely new String object
- Solution: *log.debug("{}={}", x, y)*
  - log statements allow string interpolation, with *{}* as placeholder
  - if a log is ignored (eg level WARN), then the string is discarded without the need to interpolate it

# Cloud Deployment

# Deployment

- When your application is ready, you need to *deploy* it
- But where?
- You can host your own servers, but then you need to handle everything by yourself
  - hardware (purchasing and maintenance), backups, DNS, etc.
- Many companies do it, but can be difficult for startups and private individuals

# Cloud Deployment

- Different companies provide cloud hosting solutions for your applications, which frees you from hardware issues, but for a price
- *Amazon Web Services (AWS)* is perhaps the most famous/used one
  - eg, *Netflix* runs on AWS
- *Automated scaling*: if you need more load, automatically rent more nodes, and automatically scale down if less load
  - this is also good for applications targeting a specific country (eg Norway), in which you will not get much load during the night



# Definition of “Cloud”



# Heroku

- One of the main cloud providers
- Using this one in the examples because, at the time of this writing, it provides *easy* to use *free* hosting
  - note, this might change at any time
- Supporting Java and SpringBoot applications
- Maven plugin to deploy your self-executable JAR by command line
- *Automatically* setting up environment variables to configure Spring to use Heroku's databases

# Using Heroku

- First you need to create an account at [www.heroku.com](https://www.heroku.com)
- Install *Heroku CLI*, which allows you to interact with Heroku from command line
- On the web interface, create an “app” with a name of your choice. In these slides, I will use “*quizgame-pg5100*”
  - as names are unique, you will need to choose a different name

# Jar Deployment

- Configure the *heroku-maven-plugin*
- Need to run Maven
- **mvn clean package heroku:deploy -Dheroku.logProgress=true**
- The application will then be available online at <https://quizgame-pg5100.herokuapp.com>
  - Note the HTTPS protocol, ie encrypted
- But before accessing it, you need to configure its environment

# From Command Line (CLI)

- **heroku login**
  - will setup credential for the other commands.
  - note: if using Windows, this does not work on GitBash, and need to do this command once from a regular Terminal
- **heroku ps:scale web=1 --app quizgame-pg5100**
  - enable the node resources needed to run the application
  - Note: might get an error like *"Scaling dynos... ! Couldn't find that process type"* if you haven't deployed the JAR yet at least once
- **heroku addons:create heroku-postgresql --app quizgame-pg5100**
  - add a Postgres database
- **heroku pg --app quizgame-pg5100**
  - see current status of Postgres database
- Note: some (all?) these commands can also be done from web interface



You are not logged in

Log In

Sign Up

# Quiz Game

Log in to play!!!

Code available at:

[https://github.com/arcuri82/testing\\_security\\_development\\_enterprise\\_systems](https://github.com/arcuri82/testing_security_development_enterprise_systems)

# Continuous Delivery (CD)

- Deployment can be done from Maven, as part of the build
- You could trigger a deployment at each Git Push from a CI server (eg, Travis or Jenkins)
- Of course, only if code compiles and all tests pass...
- But you might want to have a special Git branch for deployment
  - eg, development on a “*development*” Git branch and deployment on a “*deployment*” branch, done only when changes in “*development*” are *merged* into the “*deployment*” branch

# What Next?

- With what learned so far, you can build a whole functional web/enterprise application
  - GUI, security, testing, database, cloud deployment, etc.
- But this kind of *monolithic* application does not *scale* too well for large systems
- Enterprise 2 “*advanced*” course:
  - Dig into Web Services (eg REST) and details of HTTP
  - Distributed systems, in particular *Microservices*
  - Integration with frontends using JS + AJAX + WebSockets



# Git Repository Modules

- *NOTE: most of the explanations will be directly in the code as comments, and not here in the slides*
- **.travis.yml**
- **intro/spring/flyway**
- **intro/spring/logging**
- **intro/spring/deployment**
- Exercises for Lesson 11 (see documentation)