

# DevOps Continuous Integration (1/2)



Building Competence. Crossing Borders.

# CI

## Continuous Integration

# Continuous Integration

In software engineering, continuous integration (CI) is the practice of merging all developer working copies to a shared mainline several times a day.

## Entstehung

Grady Booch first named and proposed CI in his 1991 method, although he did not advocate integrating several times a day.



pexels.com

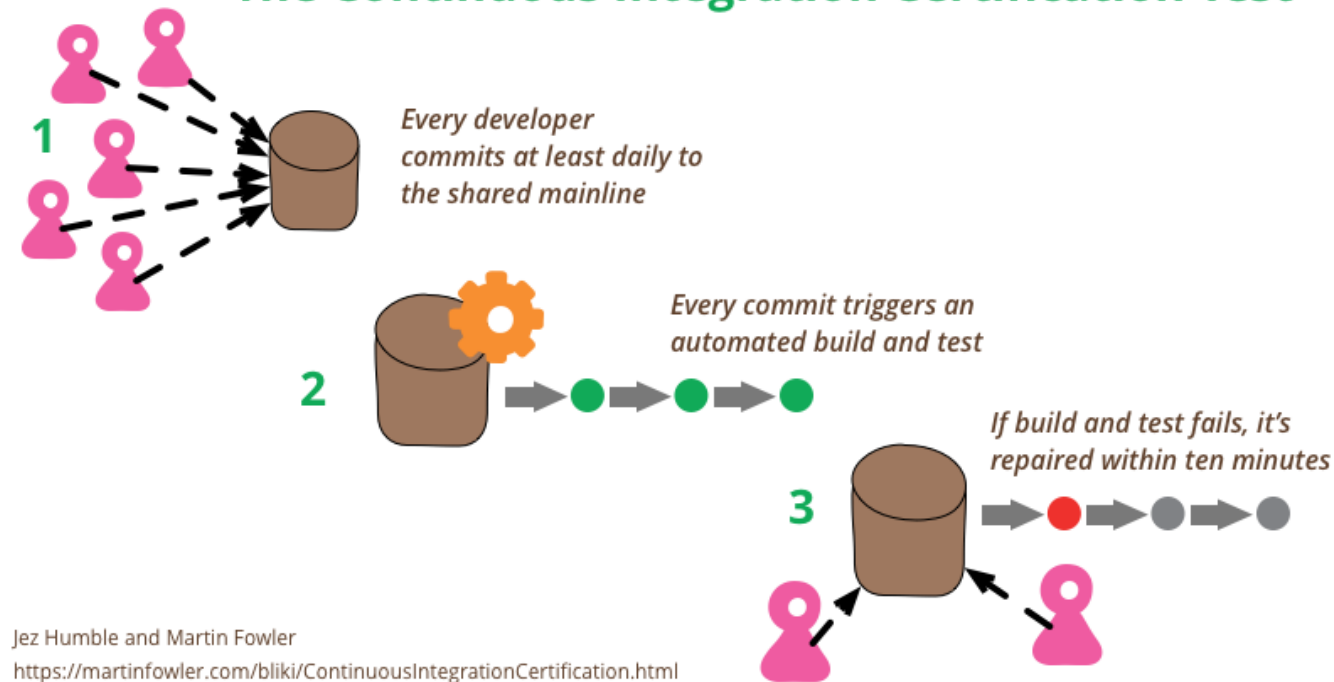
## Extreme programming

Extreme programming (XP) adopted the concept of CI and did advocate integrating more than once per day – perhaps as many as tens of times per day.

[https://en.wikipedia.org/wiki/Continuous\\_integration](https://en.wikipedia.org/wiki/Continuous_integration)

# Continuous Integration – die Herausforderung

## The Continuous Integration Certification Test



## Regeln und Metriken - Wie erreicht man CI?

- Commits sind erwünscht
- Single-Source-Repository
- Automatisieren des Build
- Testen des Build
- Schnelligkeit
- Unmittelbare Fehlerbehebung fehlerhafter Builds

# Wie erreicht man CI?

## **Commits sind erwünscht – keine Angst vor Commits**

Diese Regel ist ein Grundpfeiler für die CI. Ein Entwickler kann einen automatisierten Build einrichten und den Build mit jedem Commit ausführen lassen. Aber wenn die Kultur innerhalb der Organisation nicht «Commit-friendly» ist, gibt es keinen Mehrwert. Wenn ein Entwickler drei Wochen für ein Commit benötigt oder in eine andere Richtung lenkt, hat er die Integration verzögert und die Prinzipien missachtet. Bricht ein Build, muss das Team Wochen lang daran arbeiten, um herauszufinden, an welcher Stelle genau der Build gebrochen wurde.

**Beibehaltung eines Single-Source-Repository:** In komplexen Anwendungen nehmen Entwickler Änderungen häufig von einem «Main» ausgehenden Klon vor. Dieser sogenannte «Branch» schafft Komplexität und verhindert, dass alle an einer Single-Source-of-Truth arbeiten. Teams müssen mindestens einmal pro Tag – oder noch besser – bei jeder Änderung in «Main» einarbeiten.

# Wie erreicht man CI?

## Automatisieren des Build

Dies ist eine Vorgehensweise, die die meisten Unternehmen gut beherrschen. Einige jedoch, die behaupten, CI zu praktizieren, arbeiten mit geplanten (bspw. «nightly») oder kontinuierlichen Builds – ohne diese nach jeder Anpassung oder Änderung zu testen. Doch ohne Validierung und Testen des Builds erfolgt keine Continuous Integration.

## Testen des Build

Der erste Schritt des Validierungsprozesses ist die Kenntnis, dass ein Build mit Problemen tatsächlich fehlgeschlagen ist. Der nächste Schritt besteht darin, festzustellen, ob das Produkt des Builds funktionsfähig ist und ob es funktioniert wie erwartet. Diese Tests, sowohl schnelle funktionale als auch nicht-funktionale, sollten fester Bestandteil des Build-Prozesses sein.

# Wie erreicht man CI?

## **Schnelligkeit zählt**

Wenn es zu lange für den Buildserver dauert, eine Anwendung zu erstellen, werden Entwickler Änderungen nur zögerlich committen oder es wird am Ende eine große Anzahl an Änderungen geben. In beiden Fällen sinkt die Wahrscheinlichkeit, Fehler schnell zu erkennen.

## **Unmittelbare Fehlerbehebung fehlerhafter Builds**

Vor Jahren führte Toyota einen „Stop-the-Line“-Ansatz ein. Mitarbeiter erhielten die Ermächtigung, den Produktionsprozess sofort zu stoppen, sollten sie ein Problem entdecken. Gleiches gilt für Entwicklerteams: Ihre Aufgabe ist es, Probleme schnell zu finden und sofort zu beheben. CI fördert Prozesse, in denen Builds kontinuierlich validiert und committed werden, um Fehler einfach und schnell beheben zu können.

<https://www.dev-insider.de/wie-erreicht-man-continuous-integration-a-706469/>

# Jenkins

Tool für Continuous Integration  
Am einfachsten zu Installieren über Docker





# Jenkins Begriffe

## **Project**

A user-configured description of work which Jenkins should perform, such as building a piece of software, etc.

## **Build**

Result of a single execution of a Project

## **Step**

A single task; fundamentally steps tell Jenkins what to do inside of a Project

## **Artifact**

An immutable file generated during a Build or Pipeline run which is archived onto the Jenkins Master for later retrieval by users

# Jenkins Begriffe

## Trigger

A criteria for triggering a new Build.

## Agent

An agent is typically a machine, or container, which connects to a Jenkins master and executes tasks when directed by the master.

→ In unseren Übungen verwenden wir keine Agents

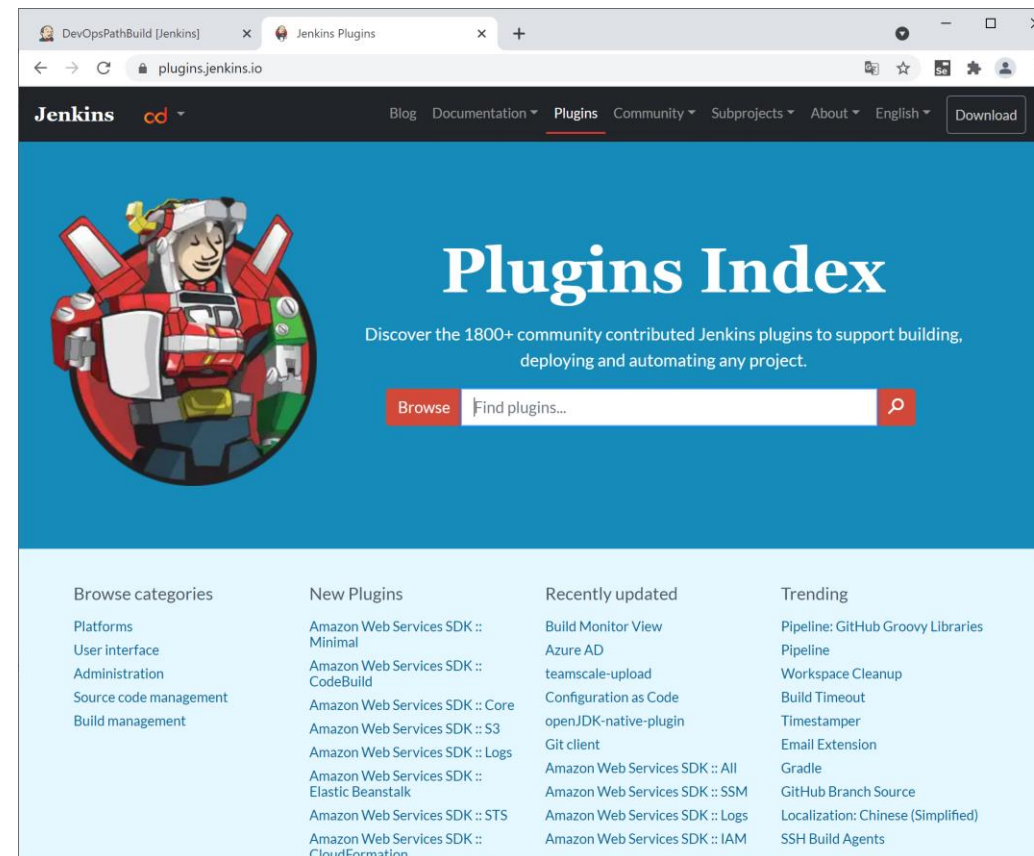
<https://www.jenkins.io/doc/book/glossary/>

# Jenkins Build Status

- Aborted** The Build was interrupted before it reaches its expected end. For example, the user has stopped it manually or there was a time-out.
- Failed** The Build had a fatal error.
- Successful** The Build has no compilation errors.  
Kann aber → **Stable** oder → **Unstable** sein
- Stable** The Build was **Successful** and no Publisher reports it as Unstable.
- Unstable** The Build had some errors but they were not fatal. A Build is unstable if it was built **successfully** and one or more publishers report it unstable. For example if the JUnit publisher is configured and a test fails then the Build will be marked unstable.

# Jenkins Plugins

Jenkins ist komplett modular aufgebaut und durch Plugins erweiterbar. Die Plugins für diesen Kurs (z.B. JaCoCo und JUnit Report) sind bereits vorinstalliert).



# Jenkins – Build erstellen

Build erstellen, um DevOpsDemo zu bauen

## 1. Jenkins Projekt

- Name
- Freestyle Project

Nach jedem Schritt Build  
testweise ausführen

## 3. Build Environment / Buildumgebung

- Delete Workspace before Build starts

## 4. Build / Buildverfahren

- Invoke Gradle Script / Gradle ausführen
- Use Gradle Wrapper
- Make gradlew executable
- Wrapper location: backend
- Tasks: test
- Root Build Script: backend (Advanced...)

## 2. Source Code Management

- Git: Repository URL einfügen
- Branch: \*/main (statt master)

### Build Steps

The screenshot shows the 'Build Steps' configuration in Jenkins. It features a list of steps: 'Gradle ausführen' (selected), 'Invoke Gradle', and 'Use Gradle Wrapper'. The 'Use Gradle Wrapper' step is expanded, showing a checked checkbox for 'Make gradlew executable' and a text field for 'Wrapper location' containing the value 'backend'. Below this, the 'Tasks' section shows a text field with the value 'test'. At the bottom, there is a dropdown menu set to 'Erweitert' and a link labeled 'Edited'. A yellow callout box labeled 'Advanced...' points to the 'Erweitert' dropdown.

# Build überprüfen

Tutorial

DevOpsDemoBuild #5 Console X

localhost:8084/job/DevOpsDemoBuild/5/console

Jenkins

Suchen (CTRL+K)

Dashboard > DevOpsDemoBuild > #5

Status

Änderungen

Konsolenausgabe

Als formatierten Text anzeigen

Build-Informationen editieren

Build '5'

Data

er Build

✓ **Konsolenausgabe**

```
Started by user admin
Running as SYSTEM
Building in workspace /var/jenkins_home/workspace/DevOpsDemoBuild
[WS-CLEANUP] Deleting project workspace...
[WS-CLEANUP] Deferred wipeout is used...
[WS-CLEANUP] Done
The recommended git tool is: NONE
No credentials specified
Cloning the remote Git repository
Cloning repository https://github.com/mosazhaw/DevOpsDemo.git
> git init /var/jenkins_home/workspace/DevOpsDemoBuild # timeout=10
Fetching upstream changes from https://github.com/mosazhaw/DevOpsDemo.git
> git --version # timeout=10
> git --version # 'git version 2.39.2'
> git fetch --tags --force --progress -- https://github.com/mosazhaw/DevOpsDemo.git +refs/heads/*:refs/remotes/origin/* # timeout=10
> git config remote.origin.url https://github.com/mosazhaw/DevOpsDemo.git # timeout=10
> git config --add remote.origin.fetch +refs/heads/*:refs/remotes/origin/* # timeout=10
```

task:classes

Task:compileTestJava

Jeder Build erhält fortlaufende Nummer

Build-Details ausgeben (Konsole)

# Build Workspace

Tutorial

Workspace von „DevOpsDemoBuild“ auf „master“

DevOpsDemoBuild /

- .git
- .github/workflows
- backend
- frontend
- .gitignore
- DevOps.code-workspace
- Dockerfile
- package-lock.json
- README.md

File	Modified	Size	View
.gitignore	09.01.2024, 07:45:48	54 B	<a href="#">View</a>
DevOps.code-workspace	09.01.2024, 07:45:48	306 B	<a href="#">View</a>
Dockerfile	09.01.2024, 07:45:48	659 B	<a href="#">View</a>
package-lock.json	09.01.2024, 07:45:48	93 B	<a href="#">View</a>
README.md	09.01.2024, 07:45:48	1017 B	<a href="#">View</a>

[\(Alle Dateien als ZIP-Archiv herunterladen\)](#)

Workspace (pro Job,  
nicht pro Build)





# Testresultate und Abdeckung darstellen


Integration von JUnit und JaCoCo

## Post-build Action hinzufügen

- Pfad zu den XML beachten:  
`**/test-results/test/*.xml`

## Post-Build-Aktionen


 **Veröffentliche JUnit-Testergebnisse.** 



Testberichte in XML-Format

Es sind reguläre Ausdrücke wie z.B. 'myproject/target/test-reports/\*.xml' erlaubt. Das genaue Format können Sie [der Spezifikation für @includes eines Ant-Filesets](#) entnehmen. Das Ausgangsverzeichnis ist der [Arbeitsbereich](#).

`**/test-results/test/*.xml`

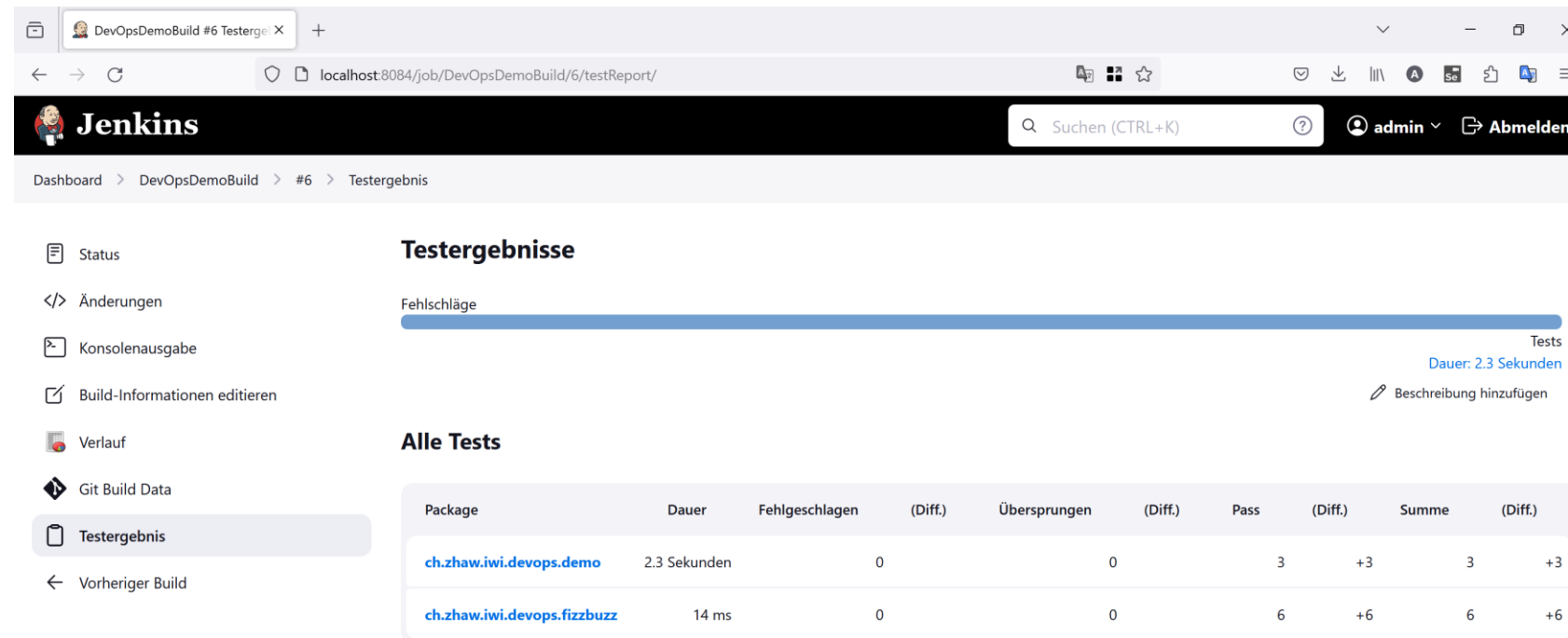


☐ Lange Standard-Out/-Error Ausgaben aufbewahren

☐ keep all the properties

# JUnit Integration in Jenkins

Tutorial



The screenshot shows the Jenkins web interface for a build named 'DevOpsDemoBuild #6'. The breadcrumb trail is 'Dashboard > DevOpsDemoBuild > #6 > Testergebnis'. The left sidebar contains links to 'Status', 'Änderungen', 'Konsolenausgabe', 'Build-Informationen editieren', 'Verlauf', 'Git Build Data', and 'Testergebnis' (which is highlighted). The main content area is titled 'Testergebnisse' and shows a progress bar for 'Fehlschläge' (Failures) with a duration of 'Dauer: 2.3 Sekunden'. Below this, the 'Alle Tests' section displays a table of test results.

Package	Dauer	Fehlgeschlagen	(Diff.)	Übersprungen	(Diff.)	Pass	(Diff.)	Summe	(Diff.)
<a href="#">ch.zhaw.iwi.devops.demo</a>	2.3 Sekunden	0		0		3	+3	3	+3
<a href="#">ch.zhaw.iwi.devops.fizzbuzz</a>	14 ms	0		0		6	+6	6	+6

REST API Jenkins 2.426.2

## Post-Build Action hinzufügen

- Record JaCoCo coverage report
- Keine Anpassungen der Parameter notwendig

### ☰ Veröffentliche die JaCoCo Testabdeckung ?

Path to exec files (e.g.: \*\*/target/\*\*/\*.exec, \*\*/jacoco.exec)

/\*\*/\*\*/\*.exec

Inclusions (e.g.: \*\*/\*.class)

Exclusions (e.g.: \*\*/\*Test\*.class)

Path to class directories (e.g.: \*\*/target/classDir, \*\*/classes)

/\*\*/classes

Path to source directories (e.g.: \*\*/mySourceFiles)

\*\*/src/main/java

Inclusions (e.g.: \*\*/\*.java, \*\*/\*.groovy, \*\*/\*.kt, \*\*/\*.kts)

\*\*/\*.java, \*\*/\*.groovy, \*\*/\*.kt, \*\*/\*.kts

Exclusions (e.g.: generated/\*\*/\*.java)

☐ Disable display of source files for coverage ?

☐ Change build status according to the defined thresholds ?

☐ Always run coverage collection, even if build is FAILED or ABORTED ?

Jenkins

localhost:8084/job/DevOpsDemoBuild/lastBuild/jacoco/

Jenkins

Suchen (CTRL+K)

Dashboard > DevOpsDemoBuild > #7 > Jacoco

Status

Änderungen

Konsolenausgabe

Build-Informationen editieren

Delete build '#7'

Git Build Data

Testergebnis

Testabdeckung

Vorheriger Build

JaCoCo Coverage Report

Download jacoco.exec binary coverage file

90

80

70

60

50

40

30

20

10

0

Zeilen abgedeckt

Zeilen nicht abgedeckt

Overall Coverage Summary

Name	instruction	branch	complexity	Zeilen
all classes	56% <div></div> M: 290 C: 364	20% <div></div> M: 40 C: 10	41% <div></div> M: 54 C: 37	51% <div></div> M: 90 C: 94

Coverage Breakdown by Package

Name	instruction	branch	complexity	Zeilen
ch.zhaw.iwi.devops.demo	M: 290 C: 258 47% <div></div>	M: 40 C: 2 5% <div></div>	M: 54 C: 24 31% <div></div>	M: 90 C: 64 42% <div></div>
ch.zhaw.iwi.devops.fizzbuzz	M: 0 C: 106 100% <div></div>	M: 0 C: 8 100% <div></div>	M: 0 C: 13 100% <div></div>	M: 0 C: 30 100% <div></div>

Jacoco > ch.zhaw.iwi.devops.demo > ToDoController

```
47:     @GetMapping("/count")
48:     public int count() {
49:         return this.todos.size();
50:     }
51:
52:     @GetMapping("/services/todo")
53:     public List<PathListEntry<Integer>> todo() {
54:         var result = new ArrayList<PathListEntry<Integer>>();
55:         for (var todo : this.todos.values()) {
56:             var entry = new PathListEntry<Integer>();
57:             entry.setKey(todo.getId(), "todoKey");
58:             entry.setName(todo.getTitle());
59:             entry.getDetails().add(todo.getDescription());
60:             entry.setTooltip(todo.getDescription());
61:             result.add(entry);
62:         }
63:         return result;
64:     }
65:
66:     @GetMapping("/services/todo/{key}")
67:     public Todo getTodo(@PathVariable Integer key) {
68:         return this.todos.get(key);
69:     }
70:
71:     @PostMapping("/services/todo")
72:     public void createTodo(@RequestBody Todo todo) {
73:         var newId = this.todos.keySet().stream().max(Comparator.naturalOrder()).orElse(0) + 1;
74:         todo.setId(newId);
75:         this.todos.put(newId, todo);
76:     }
```

REST API

Jenkins 2.426.2

FS2024

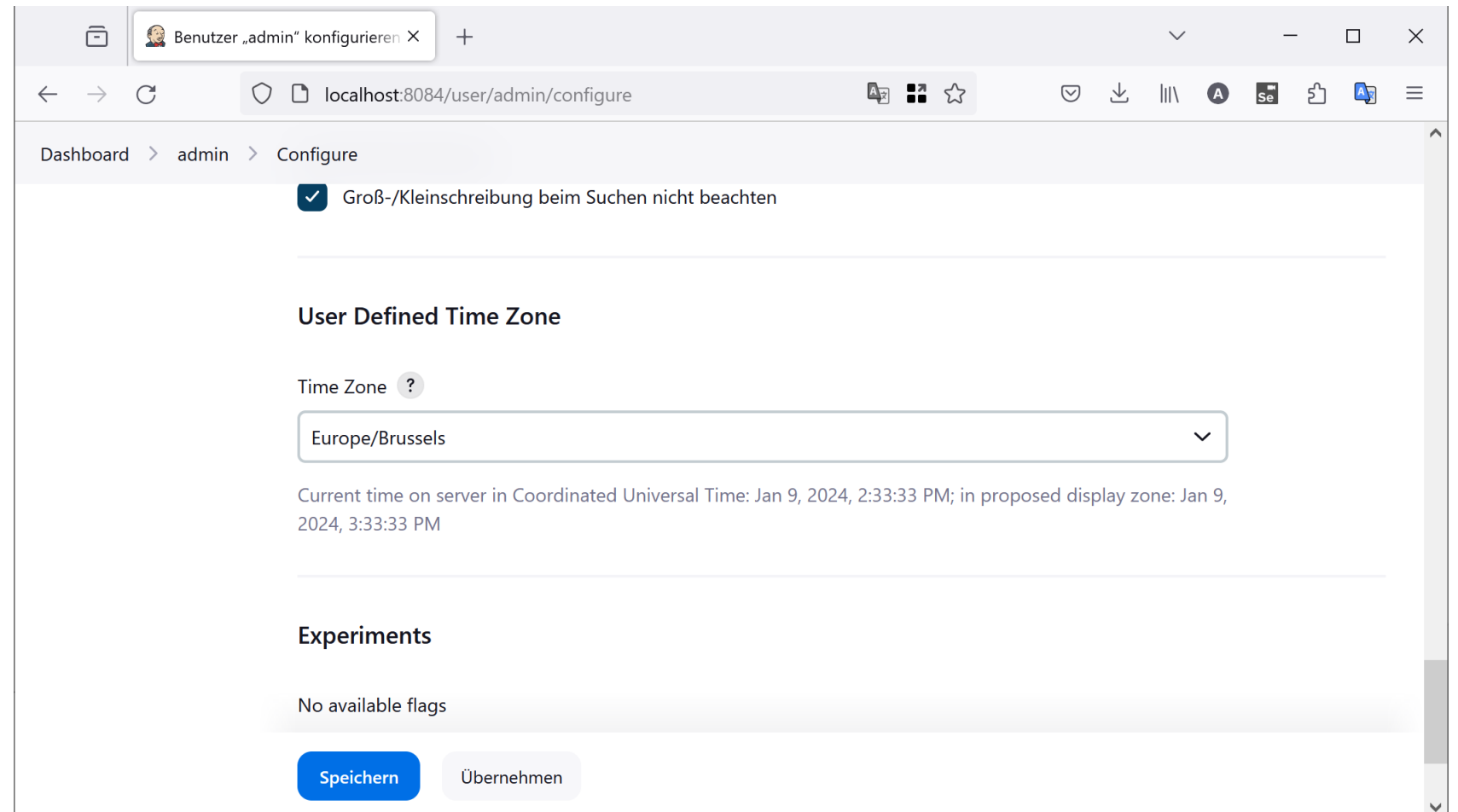
Adrian Moser – Institut für Wirtschaftsinformatik

21

zhaw School of Management and Law

# Jenkins Zeitzone

Wenn die Zeitangaben falsch sind, kann die Zeitzone für den **admin**-Benutzer gesetzt werden:



# Lernjournal

## Ziel

- Jenkins verstehen und anwenden können
- DevOpsDemo mit Jenkins bauen und Code-Coverage sowie Unit-Test Reports erstellen

## Checkliste

- ✓ Docker und Jenkins aufsetzen
- ✓ Build für eigenen Fork von DevOpsDemo erstellen
- ✓ JUnit einbinden und dokumentieren
- ✓ JaCoCo einbinden und dokumentieren
- ✓ Anpassungen am Code vornehmen und Build jeweils erneut starten
- ✓ Dokumentation von Veränderungen bei Testabdeckung
- ✓ Dokumentation von Veränderungen bei Anzahl Tests