Moderne Softwareentwicklung: MediTrack (Gruppe 2)

Github-Quelle: https://github.com/Nadolze/MediTrack (eventuell Umzug auf dedizierten GitHub-Server innerhalb der kommenden Woche)
Gruppenmitglieder: Jessica Windoffer, Timo Nadolny, Marcell Wolf de Lêu, Lea Ebitsch, Konstantin Königshofen

2. Übung - Einführung in CI/CD

Diskutieren Sie im Team (a) die grundlegenden Konzepte von Continuous Integration und Continuous Deployment und (b) wie CI/CD Ihre Softwareentwicklung unterstützt und wie automatisierte Tests und Builds in den Workflow integriert werden können. Halten Sie Ihre Diskussionsresultate fest, um diese in der Zwischenpräsentation zu vorzustellen.
Konzepte: Zentrale Codebasis, Automatisierte Builds, Automatisierte Tests, Schnelles Feedback, Versionierung (Lea)

Vorteile CI/CD: kürzere Veröffentlichungszeit (Deployment), kostengünstiger und einfach, dokumentierter Ablauf des Deployment-Prozesses, Fehler werden früher erkannt, Automatisierung (Konstantin)

Nutzen Sie Plattformen wie GitHub Actions, GitLab CI/CD, Travis CI oder Jenkins, um eine CI/CD-Pipeline in Ihrem Repository zu erstellen (Sie haben die freie Wahl). Begründen Sie Ihre Wahl.

GitHub Actions (Lea)
Vorteile
- Nahtlose Integration in GitHub
- Sehr einfache Einrichtung: Workflows können über .github/workflows/ eingerichtet werden
- Kostenlos für Public Projekte
- Viele vorgefertigte Actions
- Cloud-basiert

Nachteile:
- Starke Bindung an GitHub
- Im Vergleich zu Jenkins, weniger Kontrolle über die Build-Umgebung
- Komplexere Pipelines können unübersichtlich werden

Jenkins (Timo)
Vorteile:
- Mächtiges Tool, das über verschiedenste Repositories unterschiedlicher Anbieter funktioniert
- Lokaler Webservice inklusive Anmeldemöglichkeit und Administration
- Möglich auch auf externen Webserver zu etablieren, sodass alle Teammitglieder zugriff darauf hätten

- Mehrere Building-Prozessoren möglich
- Viele Plugins verfügbar (z.B. zum Server-Deployment nach dem Test, Bootstraps-APIs, …)
- Ressourcenüberwachung inkl. Cloud-Dienste und Nutzung deren APIs

Nachteile (für unsere Gruppe):
- Lokaler Webservice, der bei allen Projektteilnehmer eine Zusatzinstallation mit identischer Konfiguration erfordert
- Benötigt für GitHub Proxies, um aus der lokalen Umgebung eine öffentliche Adresse zu generieren (da GitHub im www). Möglich durch Anbindung via ngrok (kostenpflichtig bei persistentem Link)
- Medienbruch durch Nutzung unterschiedlicher System (Verlassen des GitHub-Universums)
- Sehr Mächtiges Tool, deswegen aufwendiger aufzusetzen

Fazit (Timo): Wenn wir uns die Mühe machen möchten, einen dedizierten (virtuellen) Server zu betreiben, ist dies für ein sehr geringes Entgeld möglich. Und das sollten wir auch tun, da so sichergestellt werden kann, dass alle Teammitglieder die selben Softwarestände, Abhängigkeiten und Konfigurationseinstellungen haben - denn diese sind durch den Server vorgegeben und werden direkt auf diesem via Jenkins geprüft. Ob es noch zusätzlich einen eigenen Git-Server auf dem vServer geben soll (der public readable wäre), muss noch (nachträglich) in einer Diskussion der Aufgabe 1 stattfinden. Aus kostengründen würden allerdings dennoch die Tests lokal stattfinden können, anstatt für ein Studentenprojekt noch einen zusätzlichen Test- und Produktivserver aufzusetzen.
Timo hat einen funktionierenden Workflow mit Jenkins und GitHub eingerichtet und wird diesen auf einen public Server übertragen.

Diskutieren Sie, wie ein Deployment-Prozess aussehen könnte. Wo und wie könnte Ihre Anwendung in Zukunft automatisch deployed werden? Welche Plattformen wären relevant? Vor- und Nachteile?

Diskussion
Wie ein Deployment-Prozess aussehen könnte, kann relativ einfach durch Prozessmodelliereung gemacht werden, indem wir den beschreiben. Also Main, dann Branch, dann Test, dann Branch zu Main, Main wird automatisch auf einen Test-Webserver gepushed, der wird von Menschen getestet (bzw. die Funktion), dann abgenommen, dann auf dem Produktivserver deployed (Timo)

Zuerst wird die Anwendung vom Entwickler lokal geändert, getestet und anschließend auf Github hochgeladen. Wenn nun Jenkins mit einem Server verwendet wird, erkennt es die Änderung automatisch und führt Unit- und Integrationtests durch. Wenn die Tests erfolgreich waren erstellt Jenkins ein Docker-Image und lädt diese in eine Container Registry (z. B. Docker Hub oder AWS ECR) hoch, die anschließend das Deployment durchführen (AWS- oder Azure-Umgebung mit Kubernetes oder Docker Compose).
(Konstantin)

Zusammenfassung/Vorschlag Timo/Konstantin
Um einen zielführenden Deployment-Vorgang zu etablieren, muss dieser für alle Gruppenmitglieder genau definiert werden.

Zunächst sollte im Vorfeld definiert werden, welches Teammitglied konkret an welcher Aufgabe wann arbeitet. Dafür könnte Zusatzsoftware wie Jira genutzt werden, um den Informationsfluss für alle Teammitglieder zu gewährleisten. In einem einfachen Projekt wäre dies aber übertrieben und würde mehr Zeit für das Erlernen neuer Software aufwenden, als es für die Projektübersichtlichkeit nutzt. Alternativ kann für die Aufgabenaufteilung auch mit einer einfachen Tabelle gearbeitet werden, die die unterschiedlichen Aufgabenpakete und Verantwortlichkeiten benennt.

Der Verantwortliche sollte als Erstes, bevor ein Modul, Abschnitt, Element, etc. erstellt oder verändert wird, den aktuellen Stand vom Git-Repro fetchen oder direkt pullen. Wenn es keine Differenzen zu eventuell bereits weiterentwickelten Abschnitten gibt, kann die Weiterentwicklung des Produkts anhand der ToDo-Liste erfolgen.

Wenn ein Feature entwickelt ist, wird dieser zunächst auf einem eigenen Branch ausgelagert und lokal getestet. Wichtig hierbei ist es, einen dementsprechenden Testfall in Java zu entwickeln und diesen mit zu speichern. Ist der Testfall entwickelt und lokal getestet worden, wird der Branch auf das Remote-Git übertragen. Durch den Push-Vorgang erhält Jenkins den Auftrag, das Projekt in der Serverumgebung zu builden und anschließend zu testen. Ist der Test erfolgreich, wird der Branch den anderen Teammitgliedern zur Freigabe vorgelegt. Erfolgt eine Freigabe von mindestens 2 Projektverantwortlichen, wird der Branch auf das Main übertragen. Im Anschluss werden entweder die gebuildeten Daten direkt auf dem Server deployed oder via Docker, etc. paketiert und als Gesamtimage hochgeladen.

Diese Vorgehensweise weisen auch einige Vor- und Nachteile auf. Der große **Vorteil** liegt am konsistenten Code über alle Teammitglieder hinweg. Unterschiedliche Stände werden minimiert, sodass Bugs durch “Restmaterial” minimiert werden. Durch die Verteilung von Aufgabenblöcken vor der Programmierung wird Doppelarbeit vermieden und nachvollziehbare Verantwortlichkeiten definiert. Im Nachfragefall (“Wieso wurde XY so umgesetzt”?) kann direkt der Programmierer identifiziert werden, der für den Abschnitt verantwortlich ist. Weiterhin kann so auch gut parallel an verschiedenen Features gearbeitet werden: Wenn ein Feature das Hauptprogramm nicht verändert, an dem Person A aktuell arbeitet, kann Person B bereits andere Teile weiterentwickeln. Und es fördert in desem Zusammenhang die modulare Denkweise in der Softwareentwicklung direkt mit dem Projektstart.

Ein weiterer Vorteil: Es entstehen keine Software-Versionskonflikte, da von Anfang an mit der Konfiguration des Servers gearbeitet wird. Insbesondere bei Java ist es wichtig, dass eine einheitliche Version eines einheitlichen Builds verwendet wird.

Bei der Verwendung von Paketierung wie Docker würde das System immer wieder frisch aufgesetzt werden. Bugs, die z.B. durch Restartefakte wie Cachefiles, zu vielen Usern, Fehler durch Veränderung der Datenbank, etc. entstehen, würden dadurch wieder minimiert. Es kann immer mit derselben Datenbank und einem frischen System getestet werden, ohne

alle Daten erneut eingeben zu müssen. Der Tester kann sich also immer auf die Integrität der Daten verlassen.

Nachteilig ist der erhöhte Aufwand der initialen Einrichtung. Ein Projekt dieser Größenordnung würde auch ohne automatisches deployen und builden funktionieren. Je größer das Projekt wird (oder wenn im Laufe des Projekts neue Mitarbeiter hinzukommen), desto mehr lohnt sich dieser Aufwand dann. Aus Studiengründen ist dieser Nachteil hier aber zu vernachlässigen.
Zudem entstehen bei Lösungen, die die Arbeitsweisen effizienter gestalten, grundsätzlich höhere Kosten. So ist Git Actions bspw. kostenlos, die ersten Tests unter den Gruppenmitgliedern zeigten aber schon Abweichungen in der Lauffähigkeit. Jenkins klappt sehr gut lokal, benötigt aber für die Anbindung an Git globale Verfügbarkeit und gleiche Konfigurationen auf allen lokalen Geräten aller Gruppenmitglieder. Das kostet zumindest Zeit und Abspracherunden. Die Verfügbarkeit könnte auch durch einen lokalen, freigegebenen Server eines Gruppenmitglieds gewährleistet werden - oder per Proxy eines Fremdanbieters. Eines kostet Strom und benötigt Backup-Lösungen, das Andere Zeit durch häufige manuelle Updates in der Git-Konfiguration. Bei der Verwendung von Docker, etc. gehen alle zusätzlichen Informationen verloren, die nicht Teil der Paketierung sind. So muss entweder dafür gesorgt werden, dass die Testdatensätze immer auch aktualisiert oder die Datenbank als Teil der Paketierung entfernt wird - mit dem Risiko, dass Datenbankfehler als Fehler im Code missinterpretiert werden.

3. Übung: Systemarchitektur Ihres Projektes modellieren

Event Storming durchführen: Führen Sie als Team eine Event Storming-Session durch, um die wichtigsten Ereignisse (Events) in Ihrem System zu identifizieren. Sammeln Sie die relevantesten Ereignisse in Bezug auf Ihre Projektidee (mind. zwei) und visualisieren Sie diese auf einem (digitalen) Whiteboard (z.B. miro, Zoom-Whiteboard).
https://miro.com/app/board/uXjVJ5a7sXo=/?share_link_id=729943574885