BabyPort

12. Juni 2024

Image wurde mit https://imageupscaler.com/ai-image-generator/ generiert am 05.12.2023.
(Prompt: Icon für eine Software mit Port und Container und Docker)

1. ÜBERSICHT

Die Software BabyPort dient als Container Management System zur Administration und Überwachung von Docker Containern. BabyPort erreicht das mittels einer GUI, die verschiedene administrative Funktionen auf Docker-Container abbildet.

2. ZIELE

1. Containerverwaltung: Dieses System verwaltet Container mithilfe einer Systemschnittstelle. Diese Schnittstelle wird vom Hauptsystem per MQTT angesteuert. Mit Hilfe dessen können verschiedene Container zur Überwachung hinzugefügt, entfernt (bzgl. der Schnittstelle), gestartet oder gestoppt werden.
2. Statusinformationen von Containern: Hierbei werden die Daten aufbereitet und zum Monitoring dargestellt.

3. AUFWANDS-STATISTIKEN

Arbeitsstunden pro Person und Hauptbeiträge

Person	Hauptbeitrag	Arbeitszeit (in Stunden (gerundet))
Felix Schiele	Plattform (DevOps) Agent (Software-Entwicklung) Test-Engineering Dokumentation	180
Nils Heinzelmann	Client-Server (Architekt) Client-Server (Software-Entwicklung) Agent (Software-Entwicklung) UML-Modellierung	154
Marius Wörfel	Client-Server (Software-Entwicklung) Agent (Software-Entwicklung) Test-Engineering Dokumentation	160
Sarah Ficht	Client-Server (Software-Entwicklung) UI-Design Innovation Research (UI-Improvements) Dokumentation	77

Arbeitsstunden pro Workflow

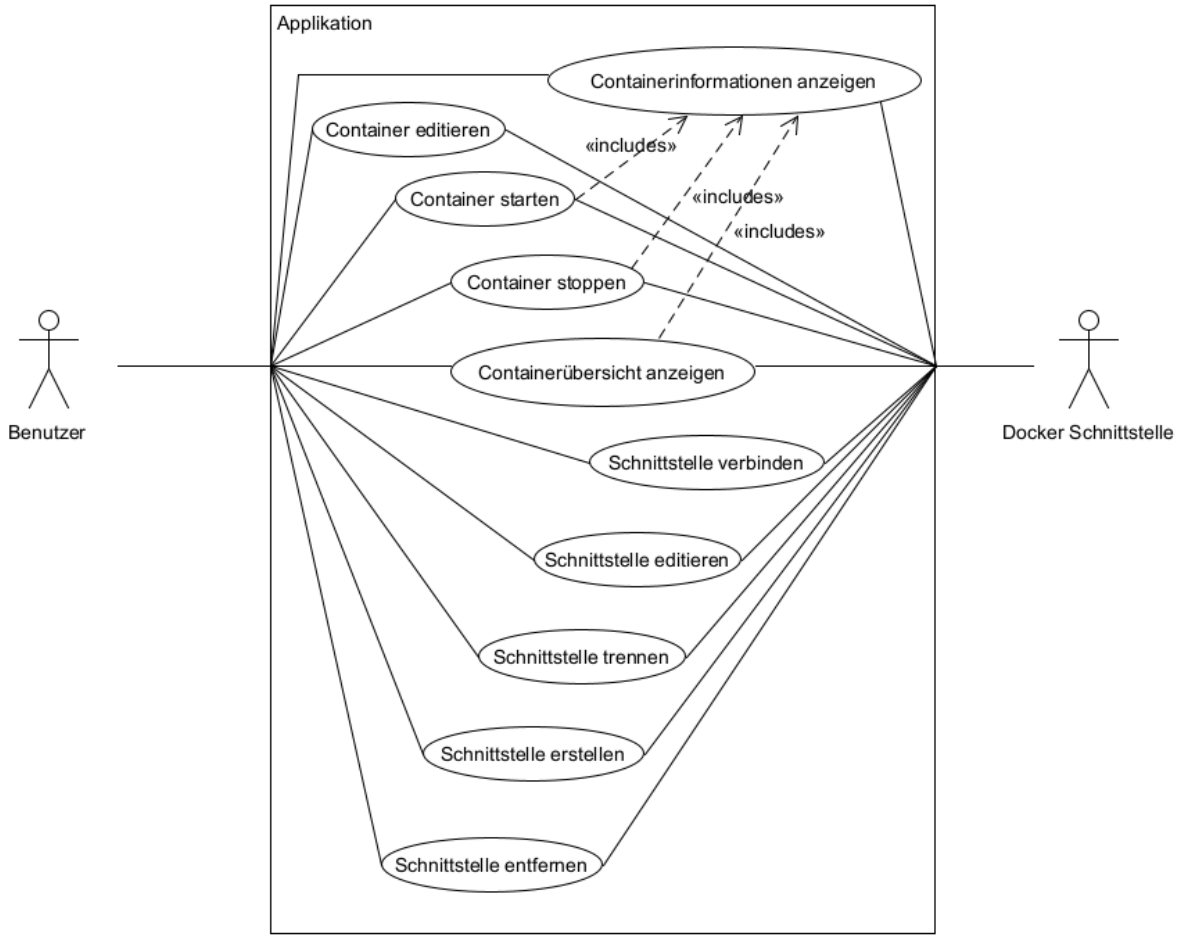
Workflow	Anteil an der Gesamtstundenanzahl	Arbeitszeit (in Stunden)
Requirements Analysis	1%	5,71
Project Management	4%	22,84
DevOps (Developer Empowerment)	10%	57,1
Dokumentation	30%	171,3
Entwicklung der Software	35%	199.85
Qualitätssicherung (Testing und Refactoring)	20%	114,2

Arbeitsstunden pro Phase

Phase	Anteil an der Gesamtstundenanzahl	Arbeitszeit (in Stunden)
Concept	1%	5,71
Inception	2%	11,42
Elaboration	15%	85,65
Construction	52%	296,92
Transition	20%	114,2
Closeout	10%	57,1

4. OVERALL USE CASE DIAGRAMM

Der Fokus dieser Implementierungsphase lag auf der Container Interaktion. Damit sind alle CRUD Operationen, welche in dem Use Case Diagramm abgebildet sind, gemeint. Dabei lag ein großer Teil der Implementierung auf Seiten der UI. Zudem wurde an der Schnittstelle gearbeitet. Für die korrekte Umsetzung aller Use Cases ist jedoch eine Verbindung zwischen der Hauptapplikation und dem Agent erforderlich. Diese wurde zu großen Teilen fertiggestellt.



Disclaimer

Aufgrund der aktuellen Implementierung ist es nicht möglich, einen Container nach dem Start zu editieren. Es ist zwar möglich, den Namen des Systemaccess-Points und den Broker, über den gesendet wird, zu ändern, nicht jedoch die Umgebungsvariablen, das Bild oder den Namen des Containers.

5. ARCHITEKTURSTILE UND ARCHITEKTURENTSCHEIDUNGEN

Die einzelnen Software-Teile des Server-Clients wurden in Komponenten unterteilt und diese Komponenten dann in einzelnen Packages zusammengefasst. Die Haupt-Architektur des Softwareentwurfs umfasst die *MVC* (Model-View-Controller) Architektur. Die Wahl MVC zu verwenden hat den Hauptgrund, da wir durch die Trennung von Model, View und Controller eine künstliche Abstraktionsebene schaffen, in der wir die MQTT-Clients/Kommunikation einfach integrieren können, ohne die Software Architektur negativ zu beeinflussen.

Ferner wurde der Agent als Sidecar implementiert. Da wir nicht nur eine Hauptapplikation besitzen, benötigen wir eine weitere Komponente, den Agent, womit ein Zugriff betriebssystemunabhängig auf andere Geräte möglich wird. Daher haben wir das Sidecar Pattern gewählt, da dieses in der Industrie (Kubernetes Sidecar Container / Jenkins Build Prozessor Agents) für genau diesem Zweck Anwendung findet.

Ferner werden verschiedene Design-Patterns (wie z.B. *Builder*, *Factory*, Strategy, Command*)* verwendet, um Code den zu entkoppeln, eine klare Struktur zu implementieren und die Wartbarkeit zu gewährleisten.

Die Haupt Kommunikation zwischen den Komponenten besteht aus einem *Observerpattern* welches eigenen Events erstellen, empfangen und versenden kann.

Aus dieser Modellierung ergibt sich, dass wir insbesondere die folgenden SOLID-Prinzipien versuchen einzuhalten.

- **Interface-Segregation-Prinzip:** Alle Funktionalitäten, die dazu dienen, zwischen Komponenten zu kommunizieren, wurden in viele kleine Interfaces oder andere Abstraktion-Konstrukte unterteilt, um so viele Abhängigkeiten wie möglich zu vermeiden.
- **Single-Responsibility-Prinzip:** Beim Entwurf wurde besonders darauf geachtet, dass eine Klasse nur einen Abhängig zu einer anderen besitzt, um eine Clean-Code-Architektur zu erzielen.

Außerdem haben wir uns vorgenommen, weitere Prinzipien wie **KISS** und **DRY** zu verwenden, um eine bessere Lesbarkeit sowie Erweiterbarkeit in der Zukunft zu gewährleisten.

5. VERWENDETE BIBLIOTHEKEN

In diesem Abschnitt werden alle verwendeten Bibliotheken nach Komponente, Bibliothek und Zweck aufgelistet.

Komponente	Server-Client	Agent (Sidecar)	Zweck
Bibliotheken	Jackson	Jackson	Bibliothek zur Erzeugung verschiedener Dateien auf der Basis von Objekten (Object-Mapper)
	Eclipse Paho	Eclipse Paho	Bibliothek für die Maschine-zu-Maschine-Kommunikation. In unserem Fall verwenden wir es für die MQTT-Kommunikation zwischen Server-Client und Agent.
	-/-	Args4J	Bibliothek für die Übergabe von Argumenten über Kommandozeilenparameter.
	FlatLaf	-/-	Bibliothek zum verbesserten Theming von Swing UI Komponenten.
	Junit 5 (Jupiter)	Junit 5 (Jupiter)	Testframework, mit dem wir Unit-Tests für die Komponenten schreiben können.
	Mockito	Mockito	Bibliothek, die es ermöglicht, Komponenten in Java zu simulieren, um andere Komponenten besser testen zu können.

6. SOFTWARE TOOLS UND PLATTFORM

Software-Tools

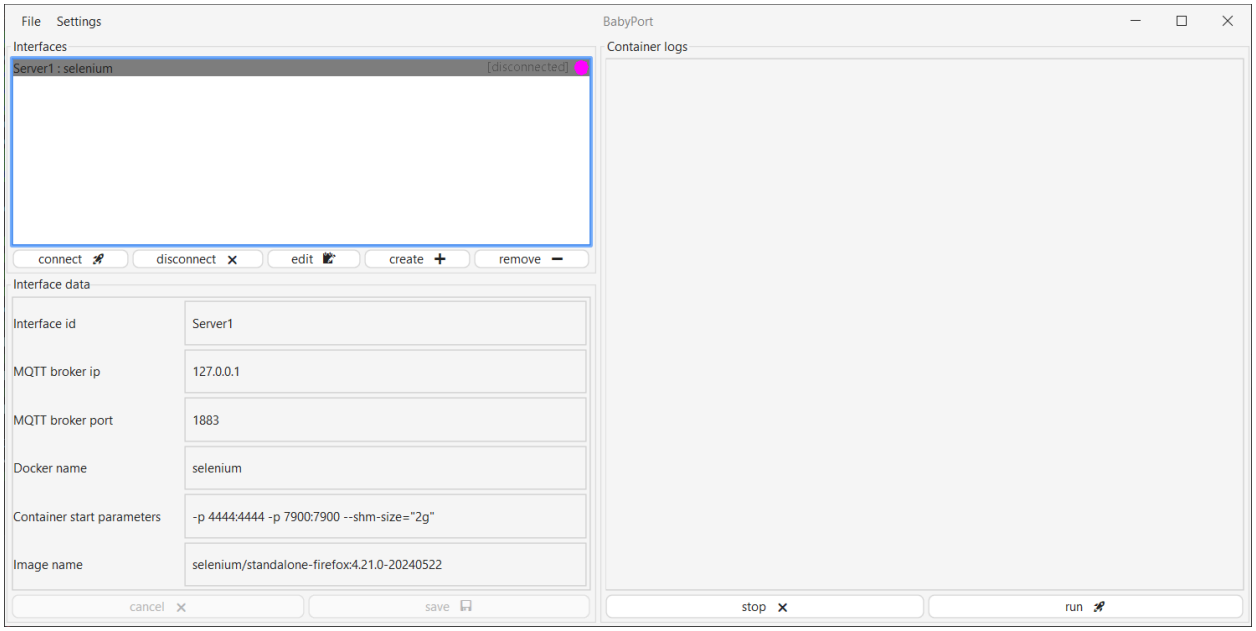
- Java (Sprache)
- Java Swing & Java AWT (UI-Framework)
- JUnit 5 (Unit Tests)
- Docker (Engine und Docker CLI)
- MQTT (Kommunikation Standard)
- SonarLint (Linter für Entwicklungsumgebung)
- Mockito (Mock-Testing Framework)
- Maven (Dependency Management + Building)
- Jetbrains IntelliJ Ultimate (Entwicklungsumgebung)
- Eclipse (Entwicklungsumgebung)
- Figma (UI-Mockup)
- Visual Paradigm (UML-Diagramm Editor)
- UMLet (UML-Diagramm Editor)

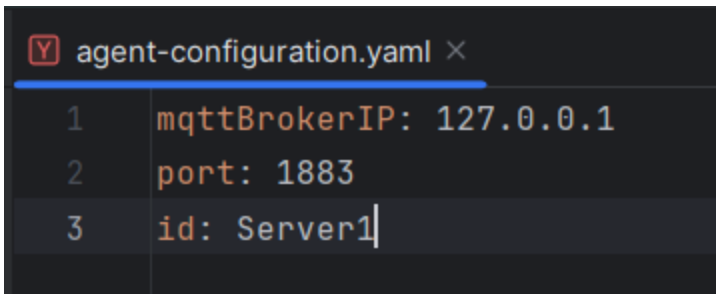
Plattform

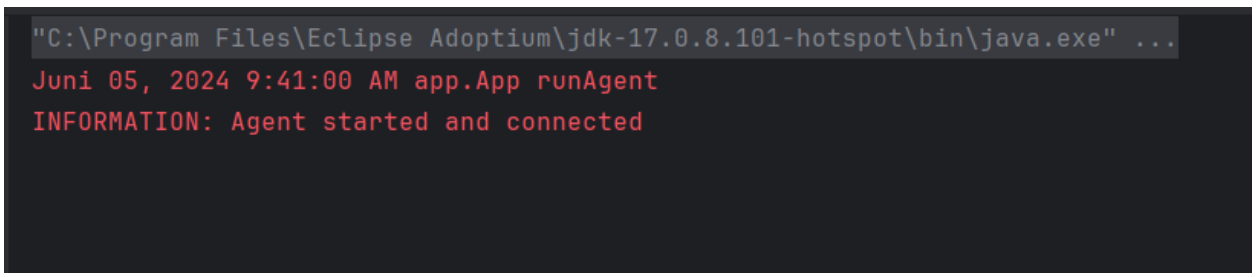
- Ubuntu Server (MQTT-Broker und Agent)
- Desktop Environment (Linux, MacOS, Windows)
- SonarQube (statische Code-Analyse)
    - CheckStyle von Google (für Java)
    - Quality Gate Settings entsprechen Abgabebedingungen
- Snyk (Security Analysis)
- Jenkins (CI/CD)
- Renovate (Dependency Management)
- GitLab (Source-Code-Control)
- Jira (Projektmanagement + Zeiterfassung)
- Medium (Blog Posting Plattform)
- GitHub (Plattform für Dokumente Veröffentlichungen)

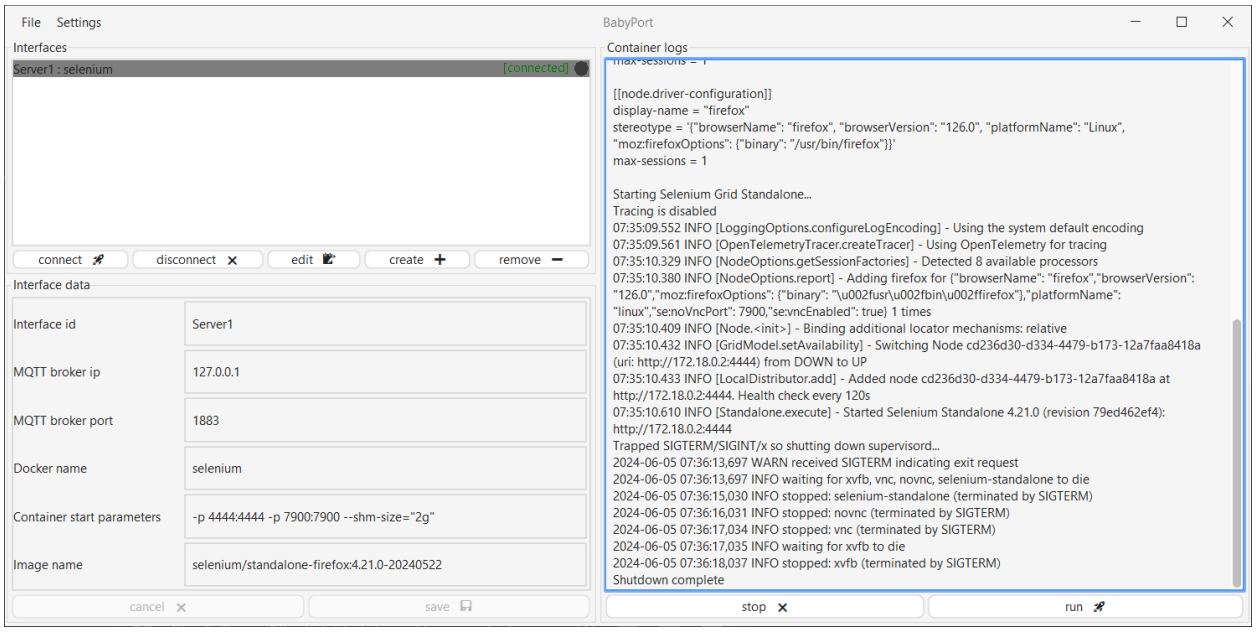
7. HIGHLIGHTS DER DEMO

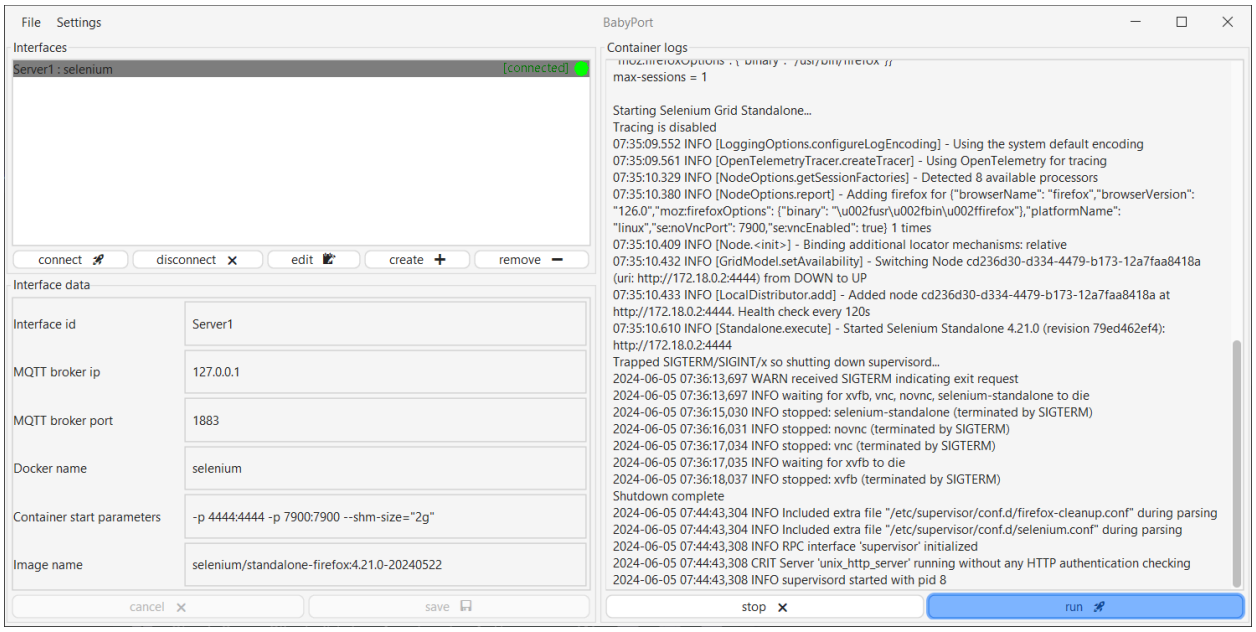
- Anlegen eines Containers in der UI
- Nutzung der Agentconfig zur Konfiguration des Agents
- Starten des Agents
- Verbindungsherstellung des Server-Clients zum MQTT-Broker
- Server-Client lädt State und Logs des Docker Containers
- Starten des Containers -> Aktualisierung des States und Logs











8. TESTING

Welche Arten von Tests werden eingesetzt?

Die aktuelle Teststruktur sieht vollautomatisierte Unit- und Integrationstests vor. Zusätzlich werden manuelle E2E (End-to-End)-Tests sowie API-Tests für die MQTT-Anbindung durchgeführt. Zum besseren Verständnis der genannten Testarten folgen deren Definitionen. Unit- und Integrationstests werden nach der Definition von Fowler durchgeführt, der sich auf eine der bekanntesten Publikationen, The Test Pyramid von Mike John, bezieht.

Unit-Tests

Unit-Tests stellen sicher, dass eine bestimmte ``Unit`` der Codebasis wie erwartet funktioniert. Je nach Programmiersprache und -paradigma kann sich die Definition einer ``Unit`` unterscheiden. Für unser Projekt im Rahmen der objektorientierten Programmiersprache Java bedeutet ``Unit`` ein logischer Zusammenschluss von Methoden einer Klasse bis hin zu einer ganzen Klasse.

Integration Tests

Integration-Tests beziehen sich nicht wie Unit-Tests auf mehrere logisch zusammenhängende Methoden, sondern testen mehrere Code Komponenten und stellen deren korrekte Zusammenarbeit sicher. Im Rahmen dieses Projekts bedeutet dies in der Regel, dass mehrere Klassen aus mehreren Java Packages getestet werden.

Neben diesen beiden Testarten, die automatisiert durchgeführt werden, kommen auch manuelle E2E- und API-Tests zum Einsatz. Ziel dieser ist es, große Teile der Anwendung in Kombination aus Server-Client und Agent zu testen.

Bei E2E-Tests werden per Button-Clicks über die Swing UI Events ausgelöst, die über die MQTT-Schnittstelle und den Broker an den Agent weitergeleitet werden. Der Agent führt je nach Anfrage die passenden Docker-Befehle aus. Dabei werden auch Events vom Agent zurück an den Server-Client gesendet. Um diese E2E-Tests manuell durchzuführen, muss zunächst ein lokaler MQTT-Broker gestartet werden, bei dem sich im Anschluss sowohl Agent als auch ServerClient beim Start verbinden.

Manuelle API-Tests verfolgen ein ähnliches Ziel wie die E2E-Tests, jedoch muss dafür in der Regel nur eine Anwendung, Server-Client oder Agent, gestartet und damit getestet werden. Der Fokus liegt darauf, dass bei ausgelösten Eingaben, die gewünschten Ergebnisse über MQTT gesendet oder die richtigen Docker-Befehle ausgeführt werden.

Was ist der Zielwert für die Testabdeckung?

Unser fester Zielwert für die Testabdeckung in Bezug auf die Coverage ist **80%** aller Zeilen.

Zusätzlich haben wir weitere Metriken, die über die Testabdeckung hinausgehen, um die Qualität unserer Software zu überprüfen. Das Quality Gate unterscheidet zwischen New Code und All Code. Für New Code verwenden wir die Einstellung, dass jeder neue Push in GitLab als New Code analysiert wird. Dies hat den Hintergrund, dass bei jeder gepushten Codeänderung die Qualitätsanforderungen eingehalten werden sollen.

Welche automatischen Testwerkzeuge werden genutzt?

- JUnit Jupiter 5 mit JUnit Params zur Parametrisierung
- Mockito
- Jacoco (Coverage Report in XML)

Wie werden Testfälle verwaltet?

Alle Testfälle werden vollautomatisch über eine CI/CD Pipeline ausgeführt, so dass nachvollzogen werden kann, welcher Testfall in welcher Code Version fehlgeschlagen ist und warum. Die Testausführung in der Pipeline ist in mehrere Schritte unterteilt. Zuerst werden alle Unit-Tests ausgeführt, danach alle Integrationstests. Für den Code Coverage Report mit Jacoco werden Unit- und Integrationstests zusammengeführt.

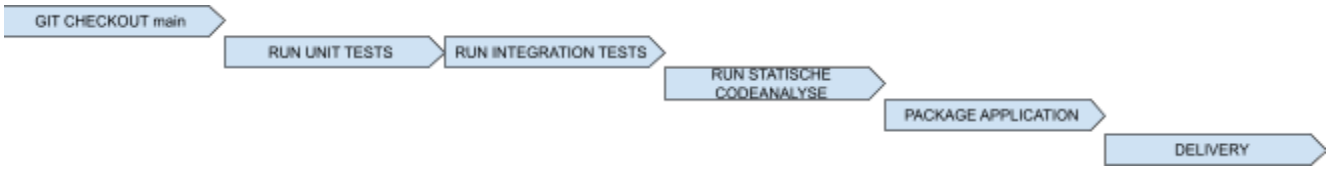
9. AUFBAU UND EINSATZ VON CI/CD

Unsere CI/CD Pipeline besteht aus verschiedenen Phasen, die am Ende der Pipeline zu einem fertigen Paket führen. Alle Komponenten, die von uns geschrieben werden, müssen diese Schritte der Pipeline bei jedem Push in Main oder bei der Erstellung eines Merge-Requests erfolgreich durchlaufen. Wenn der Code diese Schritte nicht erfolgreich durchlaufen kann, wird er nicht akzeptiert. Sobald eine Pipeline Stufe nicht erfolgreich ist, wird der Durchlauf abgebrochen und ist fehlerhaft. Alle unsere Pipelines sind in Code geschrieben, so dass alle Änderungen ständig in Git (https://github.com/babyport/babyport/tree/main/pipelines) verfolgt werden.

Kategorien	Pipeline Stufe	Zweck
Quellcodeverwaltung	GIT CHECKOUT main	In diesem Schritt wird der HEAD von main ausgecheckt, ausgelöst durch einen Push oder Merge Request.
Testing	RUN UNIT TESTS	In diesem Schritt werden die Unit-Tests der jeweiligen Komponente ausgeführt.
	RUN INTEGRATION TESTS	In diesem Schritt werden die Integrationstests der jeweiligen Komponente durchgeführt.
Statische Code Analyse	RUN STATISCHE CODEANALYSE	In diesem Schritt wird eine SonarQube-Analyse des

		Quellcodes durchgeführt. Außerdem wird die Code Coverage berechnet und als JaCoCo Bericht an SonarQube übergeben.
Packaging der Applikation	PACKAGE APPLICATION	In diesem Schritt wird die Anwendung mit Hilfe des Maven Package Goals gepackt.
Auslieferung in die dev-Umgebung	DELIVERY TO SANDBOX	In diesem Schritt wird die gepackte JAR aus dem vorherigen Schritt für manuelle End-to-End-Tests an die Sandbox übergeben.
Auslieferung in die produktion-Umgebung	DELIVERY TO PRODUCTION	In diesem Schritt wird geprüft, ob die Anwendung in der Entwicklungsumgebung funktioniert. Wenn dies der Fall ist, wird die Komponente in die Produktionsumgebung ausgeliefert.

Die folgende Abbildung zeigt schematisch, in welcher Reihenfolge die Pipeline durchlaufen werden muss, um eine gepackte JAR-Applikation in der Sandbox abzulegen.



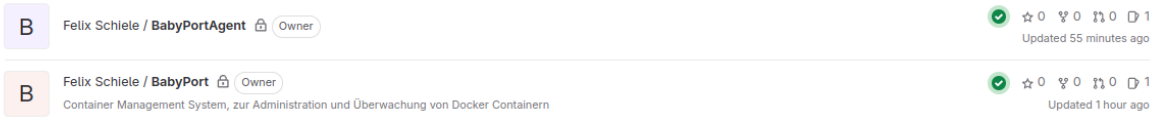
Eine grafische Darstellung der Pipeline Steps

Stages

Unser Setup sieht zwei Stages vor. Wir verwenden eine dev und eine prod Stage. Das bedeutet, dass alle Schritte der Pipeline auf der dev Umgebung ausgeführt werden. Sobald die Healthchecks für die dev Umgebung in Ordnung sind, wird die prod Pipeline getriggert, die dann die Applikation baut und an die prod Umgebung ausliefert.

Pipeline Status

Um die Pipeline noch besser in den Workflow zu integrieren, wird der Status der Pipeline an GitLab gespiegelt. Das bedeutet, dass der Entwickler direkt im Merge Request sehen kann, ob die Pipeline durchlaufen wird oder nicht, wobei der Merge solange blockiert wird, bis die Pipeline wieder grün ist.



Auszug aus GitLab mit dem Pipeline Status

10. METRIKEN

Die Software entspricht unseren Qualitätsstandards, wenn alle Metriken, die unser Quality Gate abdeckt, erfüllt sind. Wir unterscheiden hier zwischen Pflicht und optionalen Metriken.

Pflicht-Metriken

Da wir von Anfang an eine Sonarqube-Instanz unseren Code statisch analysieren lassen, haben wir uns für drei Metriken von Sonarqube entschieden. Diese Metriken müssen dann auch für die Endversion erfüllt sein. Wir haben uns für die Metriken von Sonarqube entschieden, da wir diese sehr einfach verfolgen können.

Wir haben uns für die folgenden Metriken entschieden:

- Sicherheit (Security) mit folgendem Wertebereich [A (bestes) — F (schlechtestes)]
- Zuverlässigkeit (Reliability) mit folgendem Wertebereich [A (bestes) — F (schlechtestes)]
- Wartbarkeit (Maintainability) mit folgendem Wertebereich [A (bestes) — F (schlechtestes)]

Die genauere Definition der Metriken kann hier gefunden werden:
https://docs.sonarsource.com/sonarqube/latest/user-guide/metric-definitions/

Optionale-Metriken

Zusätzlich zu den drei Pflicht-Metriken für die wir uns während der Entwicklungszeit entschieden haben, erfüllen wir zusätzlich noch die folgenden Metriken:

Conditions

Conditions on New Code

Metric	Operator	Value
Issues	is greater than	0
Security Hotspots Reviewed	is less than	100%
Coverage	is less than	80.0%
Duplicated Lines (%)	is greater than	3.0%
Maintainability Rating	is worse than	A

Conditions on Overall Code

Metric	Operator	Value
Bugs	is greater than	0
Code Smells	is greater than	0
Condition Coverage	is less than	80.0%
Issues	is greater than	0
Reliability Rating	is worse than	A
Security Rating	is worse than	A
Unit Test Failures	is greater than	0
Vulnerabilities	is greater than	0

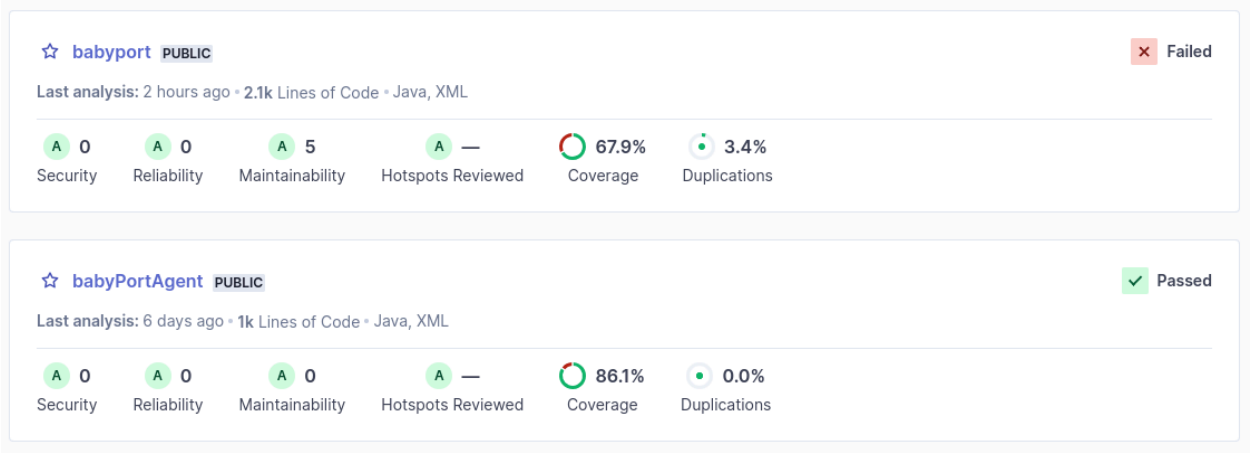
Auszug aus dem Sonar Quality Gate

Anmerkung zum Einsatz der Metriken

Wenn diese Metriken erreicht werden, gibt uns das den notwendigen Indikator, keine groben Fehler in Bezug auf Sicherheit, Zuverlässigkeit und Wartbarkeit gemacht zu haben. Dennoch verstehen wir diese Metriken nicht als 100%ige Garantie, dass die Software in all diesen Punkten unfehlbar ist.

Daher versuchen wir immer, Metriken so einzusetzen, dass sie die Softwarequalität verbessern, aber wir folgen ihnen nicht immer blind, wenn sie die Software verschlechtern, sei es die Wartbarkeit oder die Lesbarkeit des Codes.

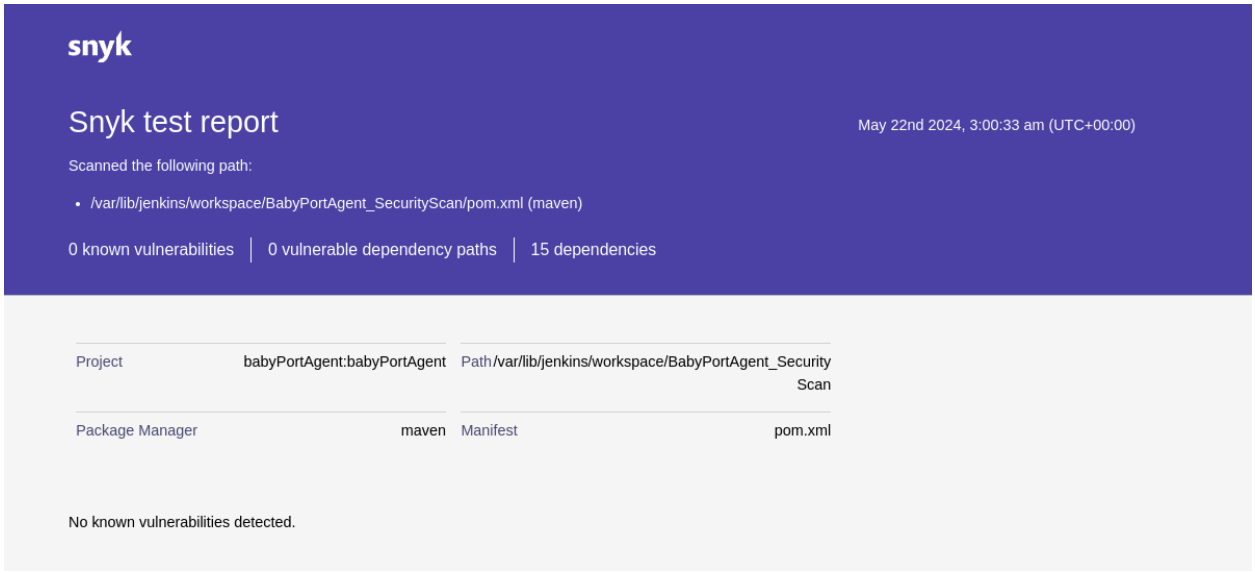
Aktueller Stand der Software Qualität (Metriken)



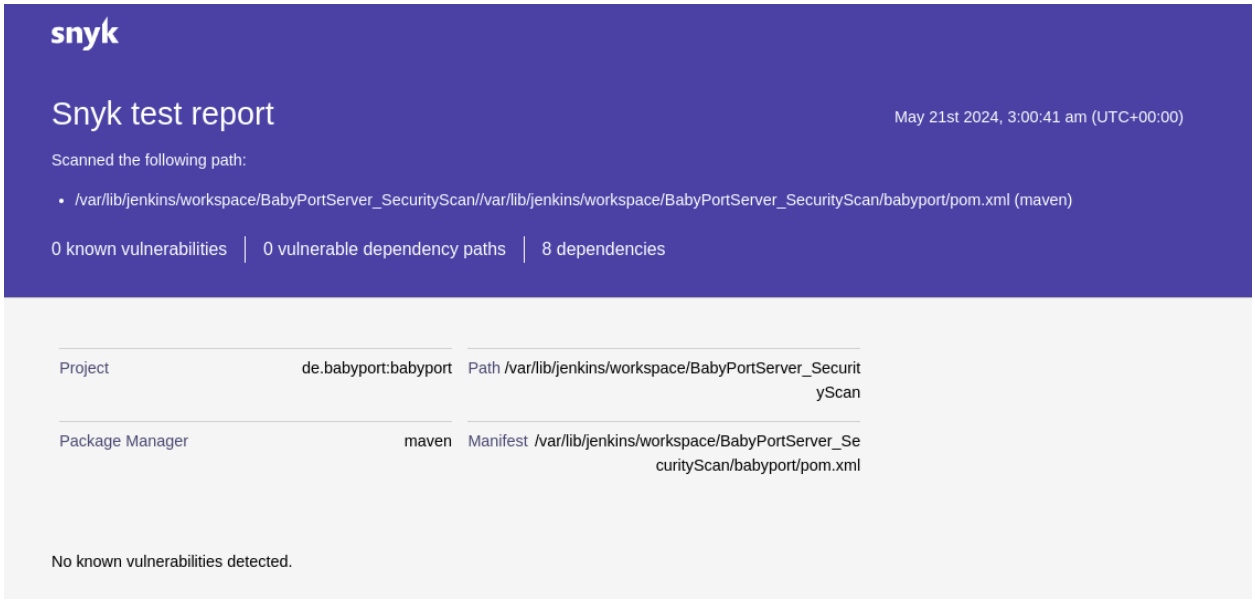
Sonarqube-Messungen zum Zeitpunkt der Abgabe für beide Komponenten

Security Analysis (Snyk)

Wir verwenden Snyk um unser Projekt zusätzlich zu SonarQube auf potentielle Sicherheitsschwachstellen zu scannen. Dies machen wir über eine zusätzliche Pipeline, die jeden Tag um 05:00 Uhr beide Komponenten auf potentielle Sicherheitsschwachstellen scannt und den Report als Artefakt sowohl in Snyk in der Cloud als auch als Artefakt in Jenkins ablegt.



Snyk Report des Agents



Snyk Report des Servers

11. Besonderheiten des Projektes (DevOps)

Renovate

Wir verwalten alle unsere Abhängigkeiten über Renovate. Dadurch müssen wir uns keine Gedanken darüber machen, wann ein Update herauskommt oder wann wir es einspielen wollen. Renovate läuft jede Stunde in einer separaten Pipeline und check alle Abhängigkeiten beider Projekte, also Server-Client und Agent. Sobald der Merge Request für ein Update von Renovate erstellt wird, wird automatisch ein Pipeline Run ausgelöst und wir können dann für den Merge Request entscheiden, ob wir das Update integrieren oder nicht.

Merge Request Abnahme Prozess

Während der Projektlaufzeit haben wir begonnen, unseren main branch für alle direkten Codeänderungen zu sperren und eingeführt, dass alle Änderungen in einem feature branch durchgeführt werden müssen und ein Merge Request erstellt werden muss. Zusätzlich ist es notwendig, dass ein zweiter Entwickler diesen Merge Request genehmigen muss, bevor dieser in den Main (als Produktion) gemerged wird. Dadurch werden Fehler vermieden und es gibt weniger Probleme in production, da der Code nach dem 4-Augen-Prinzip geprüft wurde.

12. Besonderheiten des Projektes (Features)

Agent Configuration as Code

Unser Agent wird mit Hilfe einer Config-Datei im yaml-Format gestartet, in der alle für die Laufzeit relevanten Parameter definiert werden können. Außerdem kann diese Config aus den Eingaben im Server-Client generiert werden, so dass ein Benutzer sie nicht ständig selbst ausfüllen muss.

UI-Verbesserungen mit FlatLaf

Durch den Einsatz von FlatLaf hat unser Swing UI einen neuen Anstrich bekommen und ist dadurch deutlich moderner geworden. Es gibt nun einen Dark Mode, einen Light Mode und einige optische Verbesserungen.

Export der Container Logs

Die Software ermöglicht nicht nur das Starten, Stoppen und Erstellen, sondern auch das Exportieren der Logs jedes Containers als Datei auf den Computer, auf dem der Server-Client läuft.