# Performanceoptimierung PipelineRuns Listing

|  |  |
| --- | --- |
| **Fragestellung** | Wie können wir die Performance beim Listing von PipelineRuns im Tekton Control Panel verbessern?  Aktuell wird vom Control Panel via Polling immer die vollständige Liste aller PipelineRuns vom Backend bezogen, was lange Ladezeiten und viel Datenvolumen zur Folge hat. Primärer Grund dafür sind die eingeschränkten Abfrage-Möglichkeiten von PipelineRuns via Kubernetes API (kein Paging, keine Sortierung, wenig Filter-Attribute). Wir möchten vom Client nach Möglichkeit nur Pipeline Runs laden, die sich seit dem letzten Poll geändert haben.  Eine Liste von 350 Pipeline Runs benötigt ca. 4 Sekunden Ladezeit. Diese Ladezeit ist ein dringlicher Indikator, dass hier optimiert werden muss. Zudem werden mit dem aktuellen Update-Mechanismus im Client gelöschte Pipeline Runs nicht aus der Liste entfernt bis im Browser die App reloaded wird. Auch diese Unschönheit soll mit den Optimierungen behoben werden. |
| **Rahmenbedingung** | * Datenquelle für das Listing ist nachwievor Openshift * Das bestehende API des Controllers soll keine Breaking Change erfahren (nur Erweiterungen) |
| **Annahmen** | * Die Möglichkeiten zur Abfrage von Ressourcen via Kubernetes API (fabric8 Library) bleiben eingeschränkt * Die Zahl der PipelineRuns wird eher zu- als abnehmen |
| **Alternativen / Varianten** | Zuerst wurden die Anforderungen an das API zur Abfrage von PipelineRuns definiert. Der Client möchte die folgenden Use-Cases abdecken:   * Listing aller PipelineRuns * Listing der letzten X PipelineRuns für ein Repository * Inkremetelle Updates von geänderten/gelöschten Records  Variante 1 **Spiegelung der PipelineRuns in einer In-Memory Datenbank.**  Eine Datendank mit Metadaten von PipelineRuns wird angelegt und vom Controller in Sync gehalten. Darüber können schnelle Abfragen von Client beantwortet werden. Die für das Listing benötigten DTOs werden komplett in der DB gespeichert und es muss für das Listing keine Abfrage an Openshift gemacht werden. Die Aktualisierung der DB geschieht aufgrund von Kubernetes Events, welche über einen Watcher subscribed werden. Beim Start der App werden alle PipelineRuns gelesen und die DB befüllt.   |  |  | | --- | --- | | **Vorteile** | **Nachteile** | | Schnelle Abfrage nach beliebigen Kriterien möglich | Kopie der Daten aus Openshift | |  | DB kann out-of-sync sein | |  | Höherer Memorybedarf | |  | Grösseres Refactoring |  Variante 2 **Benutzung der Quarkus Cache Funktionen für ein In-Memory Caching**  Die PipelineRunDTOs welche für das Listing benötigt werden, werden über eine mit @CacheResult annotierte Methode bezogen und können so direkt und ohne Abfrage an Openshift ausgegeben werden. Als Index kann direkt der Key-Index des Quarkus Caches benutzt werden. Das Filtering geschieht durch Iteration über die aus dem Cache gelesenen PipelineRunDTOs. Die Aktualisierung des Caches geschieht aufgrund von Kubernetes Events, welche über einen Watcher subscribed werden. Beim Start der App werden alle PipelineRuns gelesen und der Cache befüllt. Löschungen können ebenfalls in diesem Cache registriert und dem Client beim nächsten Poll mitgeteilt werden. Über den Scheduler kann periodisch der Cache geleert und neu aufgebaut werden, um ihn 100% wieder in sync zu bringen.   |  |  | | --- | --- | | **Vorteile** | **Nachteile** | | Schnelle Abfrage aus dem Memory | Partielle Kopie der Daten | | Anpassung der Filter-Kriterien einfach möglich | Cache kann out-of-sync sein | | Mit Quarkus Boardmitteln umsetzbar | Höherer Memorybedarf |  Variante 3 **Inkrementelle Updates via Websockets an den Client senden**  Die einzige Datenquelle bleibt Openshift und alle Daten werden direkt daraus gelesen. Der Client öffnet eine Websocket-Verbindung zum Controller, welcher über Änderungen an PipelineRuns über CloudEvent Notifications empfängt und diese direkt an den Client sendet.   |  |  | | --- | --- | | **Vorteile** | **Nachteile** | | Kein out-of-sync | Websockets + Angular unstabil | | Keine kopierten Daten | Keine Erfahrung mit Websockets + Quarkus | | Instand-Updates im Client | Initatialer Load im Client immer noch langsam | |
| **Entscheidung** | Es wurde **Variante 2** gewählt |
| **Begründung** | Ein Caching kann mit den verfügbaren Mitteln in Quarkus einfach umgesetzt werden. Nach dem Startup des Controllers können Anfragen blitzschnell aus dem Cache beantwortet werden. Das Empfangen von Kubernetes Events hat sich in ersten Tests als stabil erwiesen und optional kann der Cache periodisch im Hintergrund gelöscht und neu aufgebaut werden. Durch die Speicherung der für das Listing verwendeten DTOs im Cache (anstelle der kompletten PipelineRuns) bleibt der Memory-Bedarf überschaubar. |
| **Wer** | [Brüderli Thomas (IT-PTR-SL2)](file:///C:\display\~u233658) [Wallrapp Manuel (IT-PTR-EXT-EXT2 - Extern)](file:///C:\display\~ue64007) |
| **Wann** | 14.02.2022 |