# Controller API Authentication aus Pipelines

Aus Tekton Pipelines sollen via Controller API weitere Pipelines gestartet werden können ([ESTA-4989](https://flow.sbb.ch/browse/ESTA-4989)). Um API Requests an den Controller senden zu können, müssen diese eine Authentifizierung enthalten, die vom Controller akzeptiert wird. In diesem Konzept wird beschrieben, wie ein solcher Authentifizierungs-Mechanismus aussehen könnte.

# Grundidee

***Für jede Pipeline wird ein JWT generiert und via Env Variable den Tasks zur Verfügung gestellt. Dieser Token ist nur solang gültig, wie der PipelineRun noch läuft.***

**Hintergrund:**

Die Tokens sollen möglichst kurzlebig und auch nur im Kontext eines spezifischen PipelineRuns nutzbar sein. Zwar könnten langlebige Tokens statisch in OpenShift Secrets hinterlegt werden, diese sind allerdings schwieriger zu rotieren. In den dynamischen Tokens können zudem Claims wie Username/Actor, welcher die aktuelle Pipeline gestartet hat, gespeichert werden und sie können damit gleich wie die OICD JWT (von Microsoft Login) behandelt werden.

# Token Generierung

Bei jedem Start eines PipelineRuns wird ein [JWT](https://jwt.io/introduction) generiert, welcher an diese Pipeline gebunden ist. Die Claims sehen wie folgt aus:

{

"iss": "esta-tekton-pipeline-controller@<tekton-namespace>",

"sub": "<pipelinerun-id>",

"exp": 1707920236,

"name": "John Doe",

"preferred\_username": "john.doe@sbb.ch",

"roles": [

"TektonPipeline"

],

"sbbuid": "u222555"

}

* Als Issuer (iss) wird der ESTA Tekton Controller und der Namespace-Name verwendet. Damit lassen sich die selbst ausgestellten Tokens von denen von Microsoft unterscheiden.
* Als Subject (sub) wird die ID (Name) des PipelineRuns eingetragen. Darüber kann später geprüft werden, ob der Token noch gültig ist
* Weitere Claims wie name, preferred\_username und sbbuid werden vom aktuellen Kontext (API Request) in den Pipeline Token übertragen.
* Der JWT wird mit einem Key signiert, welcher für jede Tekton Instanz individuell generiert wird.

# Token Validierung

Im Controller werden die JWT Tokens im Request Handling validiert. Dies wird aktuell bereits von OIDC in Quarkus gemacht, welcher mit dem Microsoft Login konfiguriert ist. Neu muss zuerst eine Fallunterscheidung aufgrund des Token Issuers gemacht werden. Ist es ein selbst ausgestellter Token, macht der Controller die Validierung selbst. Diese Validierung ist in folgende Schritte gegliedert:

1. Prüfung der Signatur
2. Prüfung des Ablaufdatums
3. Prüfung der PipelineRun Referenz (Subject) → der PipelineRun muss existieren und den Status Running haben.

# Übergabe an Pipeline Tasks

Die JWT Tokens sollen in den Pipeline Tasks als Env Variable (oder besser noch als File) konsumierbar sein. Von einer direkten Definition via podTemplate ist abzusehen, da so die effektiven Tokens sowohl im PipelineRun (Yaml) als auch im Pod (Yaml) sichtbar wären und von Tekton Results, Splunk, etc. auch gespeichert würden. Die Tokens sollen in einem OpenShift Secret angelegt und lediglich der Secret-Name als Pipeline Parameter übergeben werden. Siehe auch [How to use Kubernetes Secret in practice](file:///C:\display\CLEW\Recommended+practice+for+OpenShift+Secrets+and+Passwords#RecommendedpracticeforOpenShiftSecretsandPasswords-HowtouseKubernetesSecretinpractice).

Hierbei wird also für jeden PipelineRun auch ein Secret in OpenShift angelegt, welches nach Beendigung der Pipeline wieder gelöscht werden sollte. Evtl. lässt sich die Löschung über eine [Owner Reference](https://kubernetes.io/docs/concepts/overview/working-with-objects/owners-dependents/) an Kubernetes delegieren.

# Umsetzung in Quarkus

Allgemeine Infos zum Authentifizierungsprozess in Quarkus finden sich hier: <https://quarkus.io/guides/security-architecture>

Die Token Generierung kann mit quarkus-smallrye-jwt-build implementiert werden. Siehe <https://quarkus.io/guides/security-jwt-build>

Für die Signierung muss ein JSON Web Key (JWK) File angelegt werden. Dies kann als OpenShift Secret gespeichert und als Volume in den Controller Pod gemounted werden.

**JWK File**

{

"kty":"oct",

"kid":"secretKey",

"k":"<random-secret-key>"

}

In der Quarkus Config wird der Pfad zu diesem Key File für sign und verify angegeben:

**application.yaml**

smallrye:

jwt:

sign:

key:

id: secretKey

location: ${JWT\_SIGN\_KEY\_FILE:/config/jwt-sign-key.json}

verify:

algorithm: HS256

key:

location: ${JWT\_SIGN\_KEY\_FILE:/config/jwt-sign-key.json}

Die Unterscheidung für die interne JWT Validierung (oidc oder self-issued) kann über eine HttpAuthenticationMechanism Klasse gemacht werden. Siehe [Dealing with more than one HttpAuthenticationMechanism](https://quarkus.io/guides/security-customization#dealing-with-more-than-one-http-auth-mechanisms).

Durch die Verwendung von JWTAuthMechanism aus dem quarkus-security Modul wird die syntaktische Validierung des JWT (Signatur, Exp) bereits gemacht, so dass im Code nur noch die PipelineRun Referenz geprüft werden muss.

Ein (unvollständiger) POC ist in Commit [59242d8978](https://code.sbb.ch/projects/KD_ESTA/repos/esta-tekton-pipeline-controller/commits/59242d8978738460f1809e90018d850f2f6a9a7a) zu sehen und kann als Basis für die Umsetzung verwendet werden.