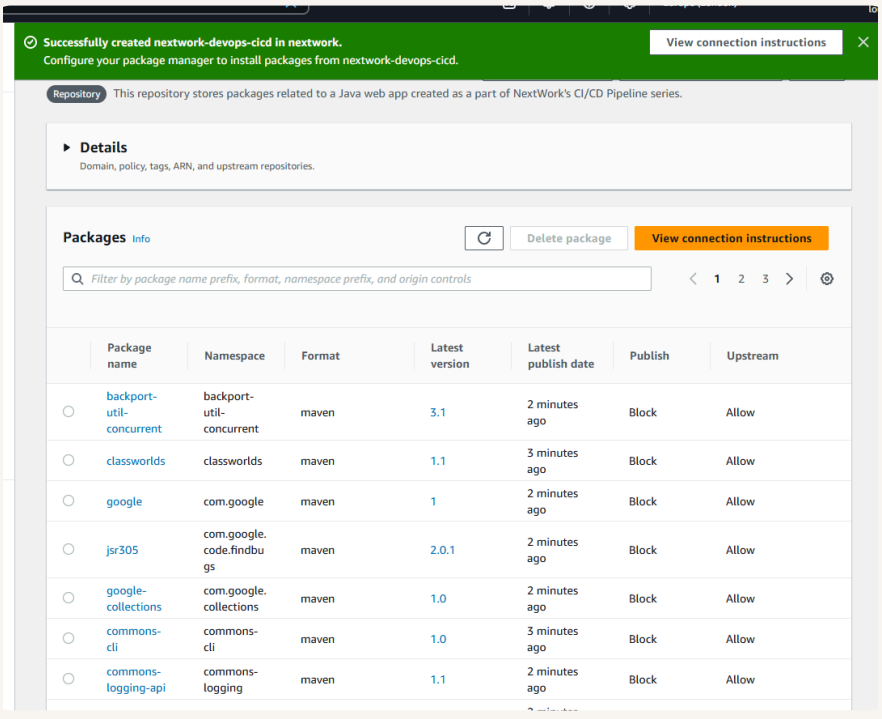
nextwork.org

# Secure Packages with CodeArtifact

Louis Moyo



L
**Louis Moyo**
NextWork Student


# Introducing Today's Project!

In this project, I will demonstrate how to set up and use AWS CodeArtifact to securely manage application dependencies. I'm doing this project to learn how to create a CodeArtifact repository, configure IAM permissions, and connect my EC2-hosted web app so it always uses verified, consistent packages in a CI/CD pipeline.

## Key tools and concepts

Services I used were EC2, IAM, and CodeArtifact. Key concepts I learnt include how to securely manage dependencies with CodeArtifact, how IAM policies and roles control access between services, and how Maven integrates with private repositories. I also gained experience with publishing and consuming custom packages, which mirrors real enterprise DevOps workflows.

## Project reflection

This project took me 2 hours to complete. The most challenging part was troubleshooting authentication errors between Maven and CodeArtifact. It was most rewarding to finally see my dependencies appear inside the CodeArtifact repository and to successfully publish and re-download my own package.

L

**Louis Moyo**
NextWork Student



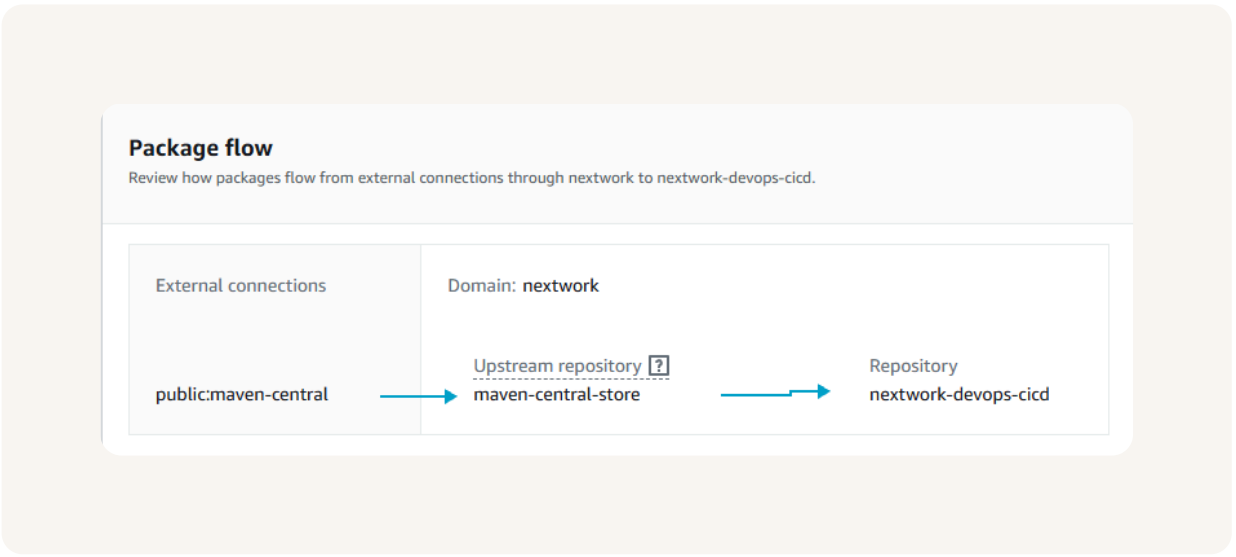
This project is part three of a series of DevOps projects where I'm building a full CI/CD pipeline on AWS. I'll be working on the next project soon, where I continue adding services and automation steps to make the pipeline production-ready.

# CodeArtifact Repository

CodeArtifact is an AWS managed artifact repository service that provides a secure, centralized place to store, share, and manage software packages such as Java dependencies, npm modules, Python libraries, and more. It integrates directly with build and CI/CD tools so teams can seamlessly pull dependencies without relying on untrusted public sources. Engineering teams use artifact repositories because they improve security, reliability, and control over software supply chains. Instead of every developer downloading packages from the internet, the team uses one trusted repository that enforces access permissions, guarantees consistent versions, and ensures builds remain reproducible even if public package registries become unavailable.

A domain in CodeArtifact is like a container for multiple repositories within the same organisation or project. Instead of configuring permissions and security policies separately for every repository, I can apply them once at the domain level and they automatically cascade down to all repositories inside it. This makes it much easier to enforce consistent access control, encryption, and lifecycle policies across all package repositories. My domain is called nextwork, and it will serve as the central place for managing policies and security settings for all repositories I create under it. This way, whether I'm storing Java packages, Node.js modules, or Python dependencies, they all inherit the same domain-level governance - making the system both scalable and secure.

A CodeArtifact repository can have an upstream repository, which means it is linked to another repository that it can fetch packages from when they are not already stored locally. This allows CodeArtifact to act as a secure caching layer: the first time a dependency is requested, it is pulled from the upstream source and then cached in my repository for all future builds.


Package flow
Review how packages flow from external connections through nextwork to nextwork-devops-cicd.
External connections
Domain: nextwork
public:maven-central
Upstream repository
maven-central-store
Repository
nextwork-devops-cicd

L
**Louis Moyo**
NextWork Student


# CodeArtifact Security

## Issue

To access CodeArtifact, we need an authorisation token because it acts as a temporary password that lets Maven securely connect to our repository. Without it, our EC2 instance doesn't know who we are, so access is denied. I ran into an error retrieving the token because my EC2 instance doesn't yet have the necessary IAM permissions to generate it.

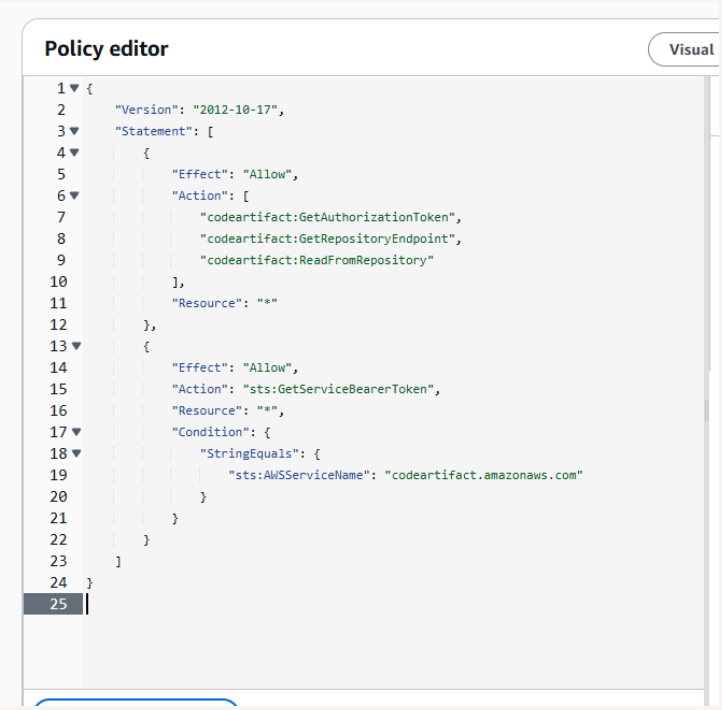
## Resolution

To resolve the error with my security token, I created an IAM policy that allowed CodeArtifact actions, attached it to a new IAM role, and assigned that role to my EC2 instance. This resolved the error because the instance could now automatically assume the role and retrieve temporary credentials, letting the export token command succeed.

It's security best practice to use IAM roles because they remove the need to hard-code or share long-term access keys on servers. Instead, AWS automatically provides temporary credentials to the instance, which are rotated and scoped only to the permissions defined in the role. This reduces the risk of credential leaks, enforces the principle of least privilege, and ensures secure, auditable access to AWS services.

# The JSON policy attached to my role

The JSON policy I set up grants permissions to get an authorization token, retrieve repository endpoints, and read packages from CodeArtifact, which are essential for Maven to securely pull dependencies. It also allows sts:GetServiceBearerToken, but only when used with CodeArtifact, ensuring temporary elevated access is tightly scoped. These permissions are the minimum required for my EC2 instance to authenticate and interact with CodeArtifact while following the principle of least privilege.



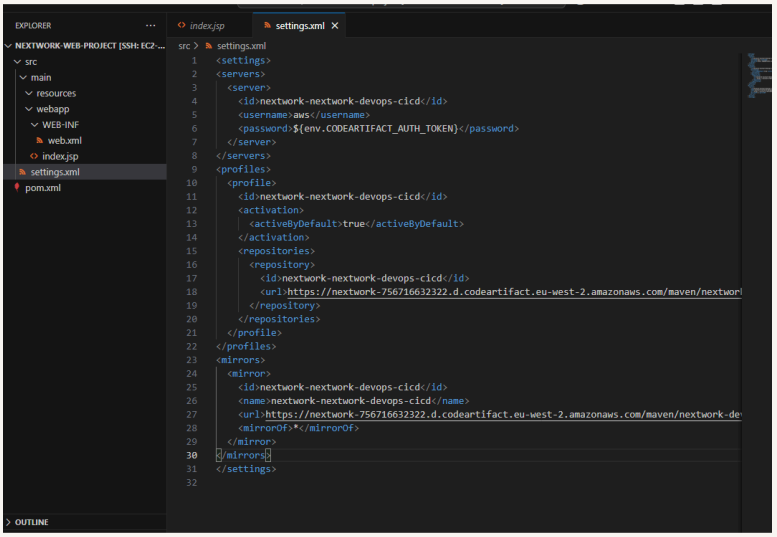
L
**Louis Moyo**
NextWork Student


# Maven and CodeArtifact

## To test the connection between Maven and CodeArtifact, I compiled my web app using settings.xml

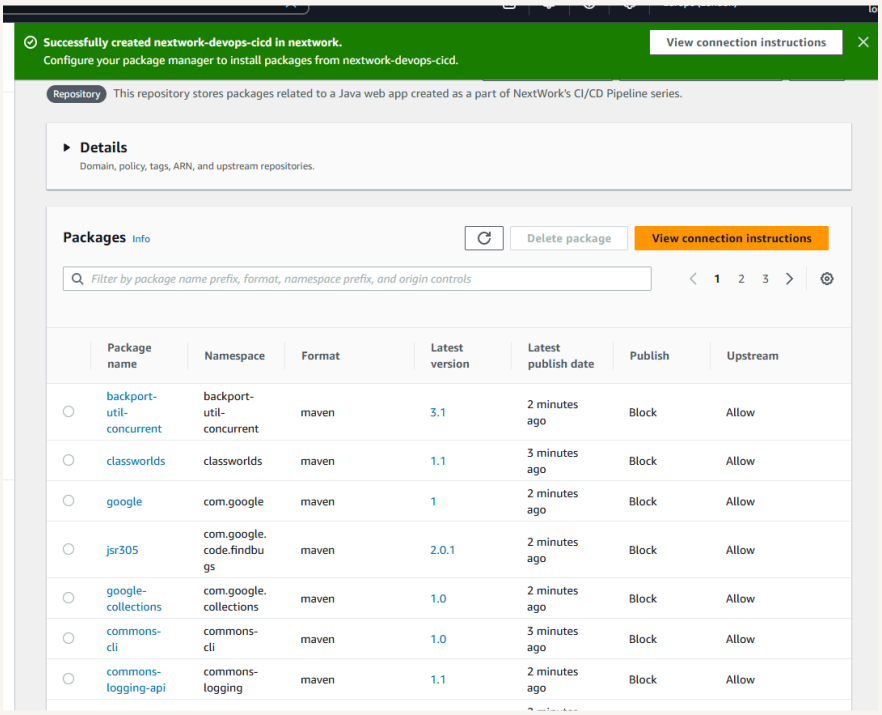
The settings.xml file configures Maven to authenticate and connect to my CodeArtifact repository. It stores the repository URL, the AWS authorization token, and the credentials needed so Maven knows where to download dependencies from and where to publish artifacts. By defining these settings, Maven automatically routes all package requests through CodeArtifact, ensuring secure and consistent access to my project's dependencies.

Compiling means converting the human-readable source code I wrote (in Java) into machine-readable bytecode that can run on the Java Virtual Machine (JVM). This step checks the code for errors and transforms it into .class files that the computer can execute. In other words, compiling is the process that turns my project's written instructions into a runnable program.


EXPLORER
index.jsp
settings.xml
NEXTWORK-WEB-PROJECT [SSH: EC2-...
src
main
resources
webapp
WEB-INF
web.xml
index.jsp
settings.xml
pom.xml
OUTLINE
src > settings.xml
<settings>
<servers>
<server>
<id>nextwork-nextwork-devops-cicd</id>
<username>aws</username>
<password>${env.CODEARTIFACT_AUTH_TOKEN}</password>
</server>
</servers>
<profiles>
<profile>
<id>nextwork-nextwork-devops-cicd</id>
<activation>
<activeByDefault>true</activeByDefault>
</activation>
<repositories>
<repository>
<id>nextwork-nextwork-devops-cicd</id>
<url>https://nextwork-756716632322.d.codeartifact.eu-west-2.amazonaws.com/maven/nextwork
</repository>
</repositories>
</profile>
</profiles>
<mirrors>
<mirror>
<id>nextwork-nextwork-devops-cicd</id>
<name>nextwork-nextwork-devops-cicd</name>
<url>https://nextwork-756716632322.d.codeartifact.eu-west-2.amazonaws.com/maven/nextwork-de
<mirrorOf>*</mirrorOf>
</mirror>
</mirrors>
</settings>
30

# Verify Connection

After compiling, I checked my CodeArtifact repository in the AWS Console. I noticed that it had automatically stored the dependencies Maven downloaded for my project, confirming that the connection between Maven and CodeArtifact was working correctly. This showed that my EC2 instance could now securely pull and cache packages in my repository.



**Louis Moyo**
NextWork Student



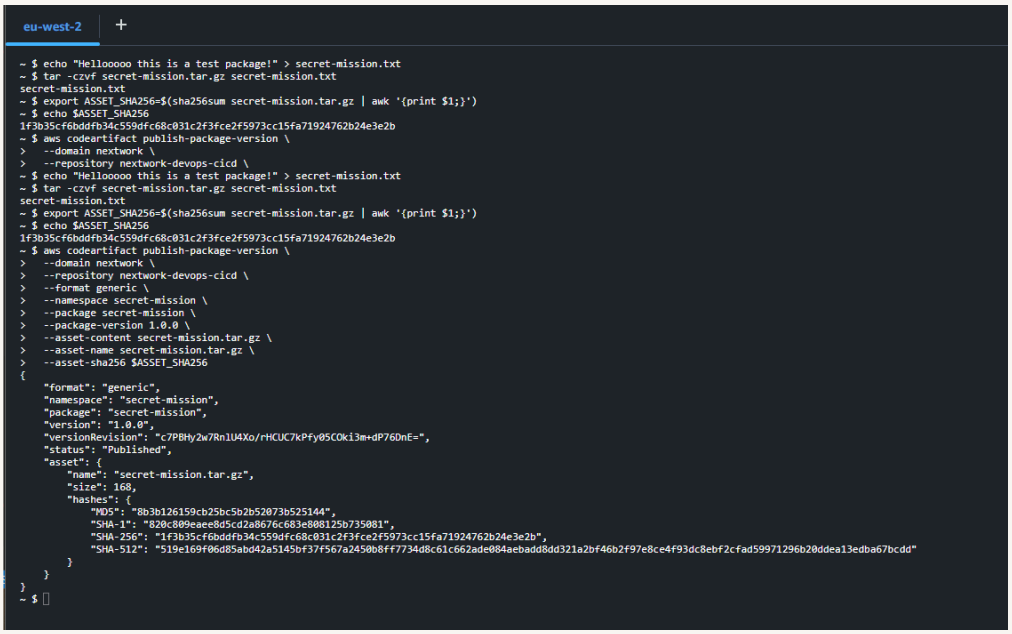
# Uploading My Own Packages

In a project extension, I also decided to create and publish my own custom package to CodeArtifact. This is useful in situations where companies want to securely share internal code libraries-like common utilities, frameworks, or proprietary business logic-without exposing them to the public. It demonstrates the full package lifecycle of not just consuming, but also publishing and reusing in-house software.

To create my own package, I wrote a simple message file, bundled it into a tar.gz archive, and prepared it for upload to CodeArtifact. I also generated a SHA256 security hash, which acts like a digital fingerprint to prove the package wasn't altered or corrupted in transit. This ensures integrity and security when publishing and sharing custom packages.

To publish the package, I uploaded my secret-mission.tar.gz file along with its SHA256 hash to my CodeArtifact repository. When I look at the package details in CodeArtifact, I can see the version I just created, the timestamp of publication, and the calculated security hashes under the Assets section. This confirms the package was stored successfully and can now be downloaded or shared securely within the repository.

To validate my packages, I downloaded the secret-mission.tar.gz file back from CodeArtifact using the AWS CLI. I then extracted it with tar -xzvf and opened the secret-mission.txt file. I saw my original secret message inside, which confirmed the package had been published, stored, and retrieved correctly without corruption.


eu-west-2
~ $ echo "Hellooooo this is a test package!" > secret-mission.txt
~ $ tar -czvf secret-mission.tar.gz secret-mission.txt
secret-mission.txt
~ $ export ASSET_SHA256=$(sha256sum secret-mission.tar.gz | awk '{print $1;}')
~ $ echo $ASSET_SHA256
1f3b35cf6bddfb34c559dfc68c031c2f3fce2f5973cc15fa71924762b24e3e2b
~ $ aws codeartifact publish-package-version \
> --domain nextwork \
> --repository nextwork-devops-cicd \
~ $ echo "Hellooooo this is a test package!" > secret-mission.txt
~ $ tar -czvf secret-mission.tar.gz secret-mission.txt
secret-mission.txt
~ $ export ASSET_SHA256=$(sha256sum secret-mission.tar.gz | awk '{print $1;}')
~ $ echo $ASSET_SHA256
1f3b35cf6bddfb34c559dfc68c031c2f3fce2f5973cc15fa71924762b24e3e2b
~ $ aws codeartifact publish-package-version \
> --domain nextwork \
> --repository nextwork-devops-cicd \
> --format generic \
> --namespace secret-mission \
> --package secret-mission \
> --package-version 1.0.0 \
> --asset-content secret-mission.tar.gz \
> --asset-name secret-mission.tar.gz \
> --asset-sha256 $ASSET_SHA256
{
"format": "generic",
"namespace": "secret-mission",
"package": "secret-mission",
"version": "1.0.0",
"versionRevision": "c7PBHy2w7Rn1U4Xo/rHCUC7kPfy05COki3m+dP76DnE=",
"status": "Published",
"asset": {
"name": "secret-mission.tar.gz",
"size": 168,
"hashes": {
"MD5": "8b3b126159cb25bc5b2b52073b525144",
"SHA-1": "820c809eaee8d5cd2a8676c683e80125b735081",
"SHA-256": "1f3b35cf6bddfb34c559dfc68c031c2f3fce2f5973cc15fa71924762b24e3e2b",
"SHA-512": "519e169f06d85abd42a5145bf37f567a2450b8ff7734d8c61c662ade084aebadd8dd321a2bf46b2f97e8ce4f93dc8ebf2cfad59971296b20ddea13edba67bcdd"
}
}
}
~ $

# The place to learn & showcase your skills

Check out nextwork.org for more projects