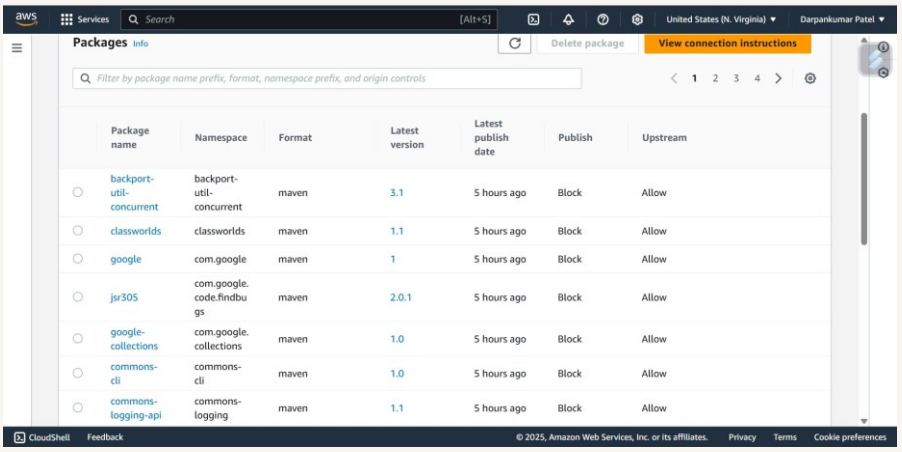
Darpan Patel

Secure Packages with CodeArtifact



Darpan Patel

Introducing Today's Project!

In this project, I will demonstrate how to set up and use AWS CodeArtifact to manage and secure Java dependencies for a web application. I'm doing this project to learn how to integrate a secure artifact repository into a CI/CD pipeline, ensure consistent dependency management across environments, and gain hands-on experience with AWS services such as EC2, IAM, and CodeArtifact. This project strengthens my understanding of DevOps best practices, particularly around package security, version control, and automated deployments.

Key tools and concepts

Services I used were AWS CodeArtifact, AWS CloudShell, and the AWS CLI. These tools helped me set up a secure and centralized system to manage dependencies and publish custom packages. I compiled a Java Maven project using CodeArtifact, published a custom tar.gz package, and downloaded it again to simulate real-world internal package distribution. Key concepts I learnt include the software package lifecycle, secure dependency management, and how to configure Maven with CodeArtifact. I also learned to generate and verify SHA256 hashes to ensure package integrity, use CloudShell for AWS operations, and understand the importance of a generic package format for custom file sharing. This experience highlighted how organizations manage proprietary code internally while ensuring speed, security, and consistency across teams. Now, I’m confident working with package repositories and understand the value of reproducible, secured software builds within enterprise environments.

Project reflection

This project took me approximately 4 to 5 hours to complete. The most challenging part was authenticating Maven with AWS CodeArtifact, especially resolving the 401 Unauthorized error which required careful checks of the `settings.xml` file, IAM role policies, and environment variables. It was most rewarding to see the full lifecycle of a custom package, from creation to publication, and finally downloading it back from the repository. That confirmed my setup worked

Darpan Patel

endto-end and gave me hands-on experience with secure dependency management—a key skill in DevOps and enterprise software development.

This project is part three of a series of DevOps projects where I'm building a CI/CD pipeline! I'll be working on the next project within the next few days to keep up the momentum and continue applying what I've learned. Each project builds on the last, so staying consistent is key to mastering the entire DevOps workflow—from source code management to automated deployment. I'm excited to see how the next challenge ties everything together!

CodeArtifact Repository

CodeArtifact is a fully managed artifact repository service from AWS that allows you to securely store, publish, and share software packages used in your application development. It supports popular package formats like Maven, npm, Python (PyPI), and NuGet, making it highly versatile for modern development environments. Engineering teams use artifact repositories because: 1. Security – Packages are stored in a centralized and trusted location, reducing the risk of pulling dependencies from unverified public sources. 2. Reliability – Applications don't break if public package repositories are down; your CodeArtifact repository serves as a reliable backup. 3. Version Control – Teams can control and manage which versions of packages are used, ensuring consistency across development, staging, and production environments. 4. Scalability & Access Control – AWS IAM policies can be used to manage who can access which packages, ideal for large, distributed teams working on shared projects.

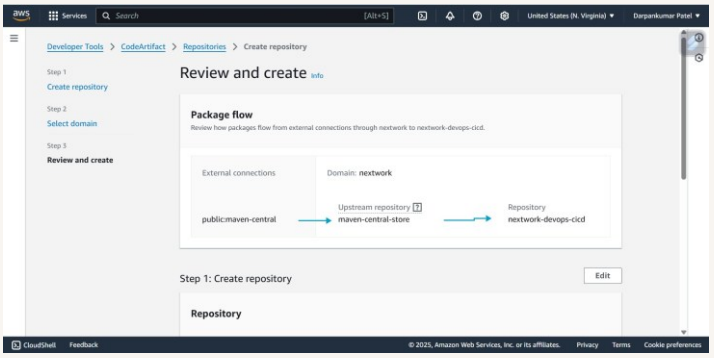
A domain is a top-level container in AWS CodeArtifact that helps manage multiple repositories under a single security and access policy framework. It allows you to centrally control who can access or modify packages across all repositories within the domain, which is especially valuable for large teams or projects with many

Darpan Patel

dependencies. My domain is `nextwork`. By using this domain, I can apply consistent permissions, manage package lifecycles, and monitor activity across all repositories in the NextWork CI/CD pipeline project. This makes security management more efficient and scalable.

A CodeArtifact repository can have an upstream repository, which means it can pull packages from another external source when the package isn't available locally.

This helps ensure that builds don’t fail due to missing dependencies and allows commonly used packages to be cached for faster and more reliable future access. My repository's upstream repository is [Maven Central] — the most widely used public repository for Java libraries. By connecting to Maven Central, my CodeArtifact repository can automatically fetch and cache Java packages needed by my application, improving speed, reliability, and security across builds.



Darpan Patel

CodeArtifact Security

Issue

To access CodeArtifact, we need an authorization token because CodeArtifact is a secure, managed service that controls who can read or publish packages. The token acts like a temporary password that proves your identity and permissions, allowing tools like Maven to safely fetch or upload packages without exposing permanent credentials. I ran into an error when retrieving a token because my EC2 instance didn’t have the necessary AWS permissions to request that token. By default, AWS resources have no access to other services until explicitly granted, so without assigning the right IAM role and policy, the instance can’t

authenticate with CodeArtifact—hence the "Unable to locate credentials" error. This security measure protects your repositories and ensures only authorized entities can access them.

Resolution

To resolve the error with my security token, I attached the correct IAM role (`EC2instance-nextwork-cicd`) to my EC2 instance. This resolved the error because the IAM role included the necessary permissions—defined in the `codeartifact-nextworkconsumer-policy`—to access AWS CodeArtifact. Once the role was attached, AWS automatically provided temporary credentials to the instance, allowing the `export` token command to successfully retrieve a valid authorization token for CodeArtifact.

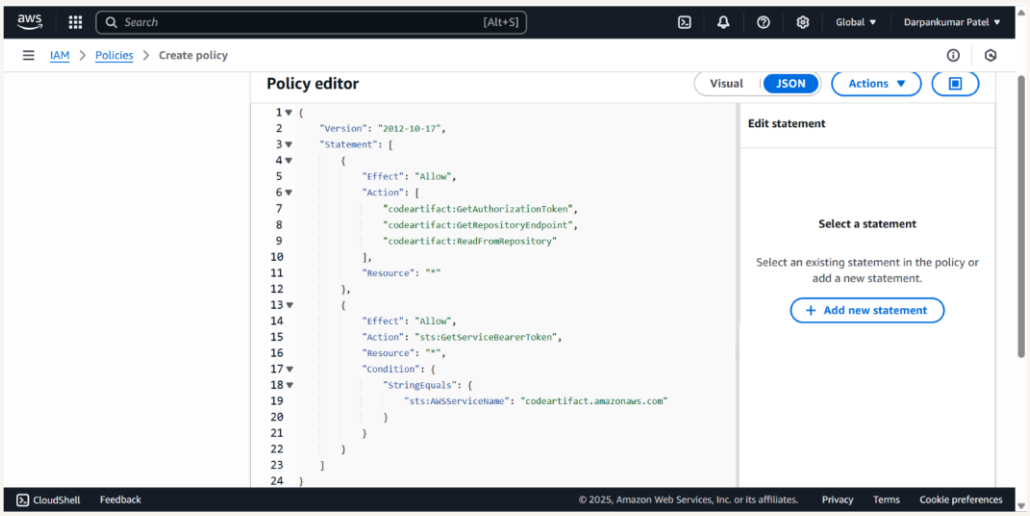
It's security best practice to use IAM roles because they allow AWS services like EC2 to access other AWS resources securely without the need to hard-code credentials or manage access keys manually. Roles provide temporary, automatically rotated credentials, reducing the risk of credential exposure. They also support fine-grained access control through attached policies, enforce the principle of least privilege, and

Darpan Patel

simplify auditing and permissions management across your infrastructure. This makes them a more secure and scalable solution for managing access in AWS environments.

The JSON policy attached to my role

The JSON policy I set up grants these key permissions: codeartifact\:GetAuthorizationToken: Lets the EC2 instance request a temporary token to authenticate with CodeArtifact. codeartifact\:GetRepositoryEndpoint: Allows the instance to get the repository URL to know where to fetch packages. codeartifact\:ReadFromRepository: Permits downloading packages from CodeArtifact. sts\:GetServiceBearerToken: Enables the instance to get temporary credentials via AWS STS specifically for CodeArtifact operations, restricted by condition to improve security. These permissions are necessary because they allow the EC2 instance to securely authenticate, locate, and retrieve Java packages from CodeArtifact. This ensures that Maven on the EC2 instance can access dependencies without granting excessive permissions, following the principle of least privilege for secure, reliable builds.



Darpan Patel

Maven and CodeArtifact

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

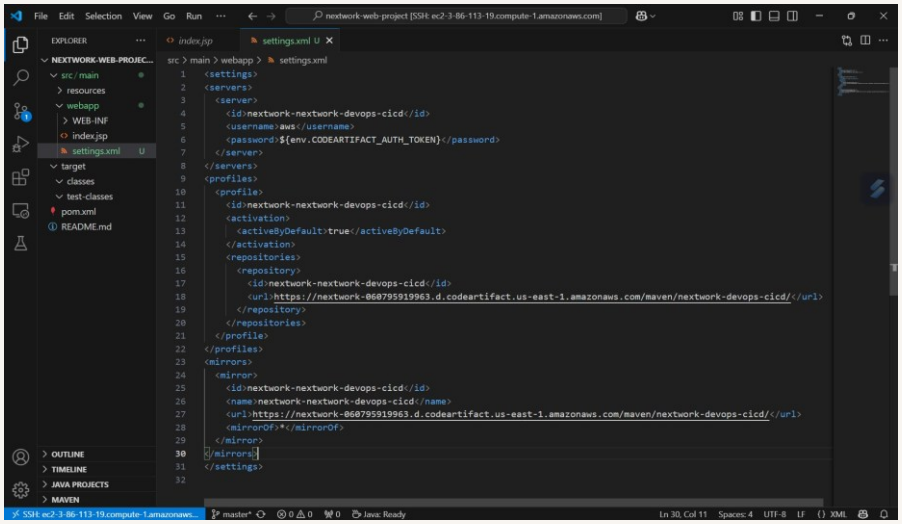
The `settings.xml` file configures Maven to work with AWS CodeArtifact by defining how Maven authenticates, where it should look for dependencies, and which repository to prioritize. Specifically, it includes: 1. Authentication credentials in the `<servers>` section, where Maven is given the AWS access token and username to securely connect to the CodeArtifact repository. 2. Repository usage rules in the `<profiles>` section, which instruct Maven to use the specific CodeArtifact URL as the source for project dependencies. 3. Fallback or default behavior in the `<mirrors>` section, which ensures that

all dependency requests (even if unspecified) are directed to the CodeArtifact repository by default. Together, these configurations ensure that Maven can automatically authenticate and fetch (or deploy) packages from your private CodeArtifact repository without manual intervention during each build.

Compiling means transforming source code written by developers into a format that a computer can understand and execute—typically bytecode or machine code. For Java projects, this involves converting `.java` files into `.class` files using tools like the Java Compiler or Maven. During compilation, the system checks for syntax errors, verifies data types, and ensures all dependencies are correctly referenced. It also validates that your code follows the rules of the programming language. In a Maven project, compiling not only builds your code but also fetches necessary dependencies from repositories like AWS CodeArtifact. This step ensures that all required packages are available locally and securely, making the build process reproducible and reliable.

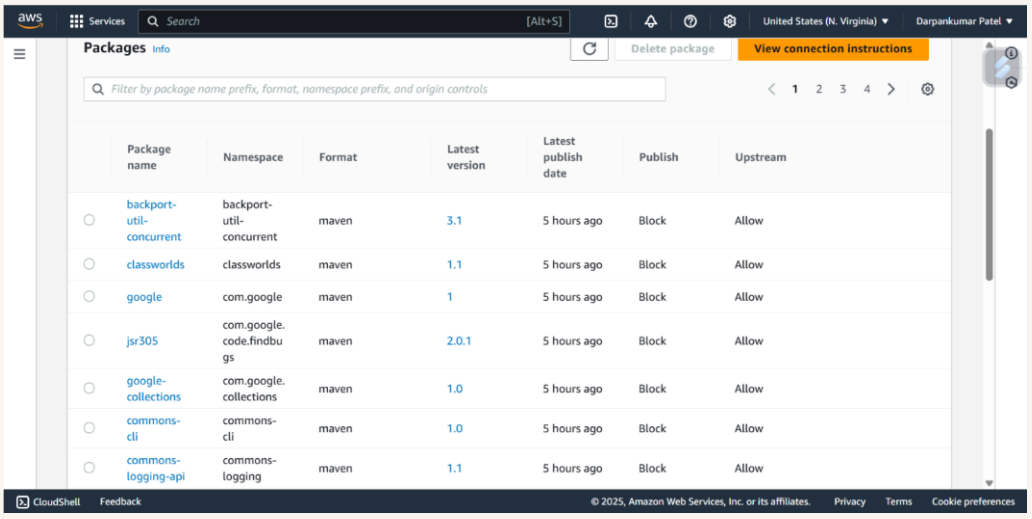
A successful compilation means your code is structurally sound and ready for testing or deployment. It's an essential step that bridges development and execution by ensuring your app is valid, optimized, and ready to run.

Darpan Patel



Verify Connection

After compiling, I checked the CodeArtifact repository in the AWS Management Console. I noticed that several Maven packages had appeared in the Packages pane, which confirmed that the CodeArtifact integration was working correctly. These packages were dependencies listed in my `pom.xml` file, and they were fetched from Maven Central via CodeArtifact since it was the first time they were requested. Their presence showed that Maven successfully pulled the dependencies through CodeArtifact, cached them in the repository, and used them to build the project — exactly as expected for a secure, centralized dependency management setup.



Darpan Patel

Uploading My Own Packages

In a project extension, I also decided to create and publish my own custom Maven package to AWS CodeArtifact. This is useful in situations where organizations need to share internal libraries — like utility functions, shared components, or proprietary code — securely within their teams without exposing them to the public. By uploading my own package, I simulated how real-world development teams manage private, reusable code across multiple projects. It allowed me to experience the full package lifecycle: authoring, publishing, and consuming. This practice enhances security, promotes consistency, and streamlines dependency management in enterprise environments.

To create my own package, I started by creating a simple text file using the command `echo "Hellooooo this is a test package!" > secret-mission.txt`. Then, I bundled that file into a compressed archive using `tar -czvf secret-mission.tar.gz secretmission.txt`, which created a `.tar.gz` file — a common format for packaging files in Linux environments. I also generated a security hash because CodeArtifact requires a SHA256 hash to verify the integrity of any uploaded package. I used the command `sha256sum secret-mission.tar.gz | awk '{print $1;}'` and stored the result in the environment variable `ASSET_SHA256`. This hash ensures that the file has not been modified or corrupted during upload, helping maintain a secure and reliable package publishing process.

To publish the package, I first created a simple text file containing a message, then bundled it into a `tar.gz` archive. I generated a SHA256 security hash for the archive and used the AWS CLI to publish the package to my CodeArtifact repository. When I look at the package details in CodeArtifact, I can see metadata such as the package name (`secret-mission`), the time it was published, the version number I assigned, and the origin marked as my own repository since it's a custom package.

Darpan Patel

Additionally, under the version details, I can view the security hash CodeArtifact calculated and compare it to my submitted hash to confirm the integrity of the package. This verification ensures the file hasn't been tampered with and confirms a successful and secure publication.

How did you validate your packages? Start your response with: "To validate my packages, I then... I saw" --- To validate my packages, I then downloaded the `secret-mission.tar.gz` file from my CodeArtifact repository using the AWS CLI. After the download completed, I extracted the contents with the `tar -xzvf` command and opened the `secret-mission.txt` file using `cat`. I saw the original message I had written earlier, confirming that the package was successfully uploaded, stored, and retrieved without any corruption or modification. This verification step ensured that my package's integrity and functionality were fully preserved throughout its journey in the CodeArtifact system.