Caching

Caching dependencies is an effective way to make your GitHub Actions workflows faster and more efficient. GitHub Actions provides a caching feature that allows you to create and use caches for dependencies and other commonly reused files.

Caching Workflow Dependencies

Workflow runs often reuse the same outputs or downloaded dependencies from one run to another. For instance, package and dependency management tools like Maven, Gradle, npm, and Yarn keep a local cache of downloaded dependencies. To help speed up the time it takes to recreate files like dependencies, GitHub Actions can cache files you frequently use in workflows.

Best Practices and Considerations

- **Sensitive Information**: Be cautious about what you store in the cache. Do not include sensitive information like access tokens or login credentials, as anyone with read access can access the cache contents.
- **Caching Across Branches**: Workflow runs can restore caches created in the current branch or the default branch (usually `main`). Pull requests can also access caches created in the base branch. However, caches cannot be restored across child branches or sibling branches.
- **Matching Cache Keys**: The `cache` action uses a cache key to restore a cache. It is essential to understand how cache keys work, including using expressions and restore keys. Cache keys should be specific enough to avoid unnecessary cache misses, but not too broad as to lead to incorrect hits.
- **Cache Hits and Misses**: When a cache key exactly matches an existing cache, it's called a cache hit. When there is no exact match, it's a cache miss, and GitHub Actions automatically creates a new cache if the job completes successfully.

Managing Caches

- **Cache Eviction**: GitHub Actions will remove caches not accessed for over 7 days. Be aware of usage limits and the eviction policy. If necessary, workflows can be set up to delete specific caches.

Usage Limits and Storage Quotas

GitHub Actions has limits on the number of caches you can store and the total size of all caches in a repository. The default limit is 10 GB per repository, but this limit may vary depending on your organization's policies or repository administrators' settings.

Testing Caching in Workflow

1. Create Workflow File:

- Create a new file named `15-01-Caching.yml` in the `.github/workflows` directory of your repository.

```yaml
name: 15-01-Caching

on:
  workflow_dispatch

jobs:
  install-dependencies:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Setup Java
        uses: actions/setup-java@v4
        with:
          distribution: 'adopt'
          java-version: '11'
          cache: 'maven'

      - name: Install Dependencies
        run: mvn dependency:go-offline

  test:
    runs-on: ubuntu-latest
    needs: install-dependencies
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Setup Java
        uses: actions/setup-java@v4
        with:
          distribution: 'adopt'
          java-version: '11'
          cache: 'maven'

      - name: Test with Maven
        run: mvn -B test

  build:
    runs-on: ubuntu-latest
    needs: install-dependencies
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Setup Java
```

```
          uses: actions/setup-java@v4
          with:
            distribution: 'adopt'
            java-version: '11'
            cache: 'maven'

        - name: Build with Maven
          run: mvn clean install -Dmaven.test.skip=true
```

- Copy and paste the provided YAML configuration into this file.

2. Understanding the Workflow:

- This workflow is triggered manually via the GitHub UI (`workflow_dispatch`).
- It contains three jobs: `install-dependencies`, `test`, and `build`.
- Each job runs on an `ubuntu-latest` runner.
- The `install-dependencies` job installs Maven dependencies using the `mvn dependency:go-offline` command.
- The `test` job checks out the code, sets up Java, and runs Maven tests using the `mvn -B test` command.
- The `build` job checks out the code, sets up Java, and builds the project using the `mvn clean install -Dmaven.test.skip=true` command.

3. Testing the Workflow:

- Commit and push the workflow file (`15-01-Caching.yml`) to your repository.
- Navigate to the "Actions" tab in your GitHub repository.
- Manually trigger the workflow by clicking on the "Run workflow" button for the `15-01-Caching` workflow.
- Monitor the workflow run and verify that all three jobs (`install-dependencies`, `test`, and `build`) complete successfully.
- Check the logs of each job to ensure that the Maven dependencies are installed, tests are executed, and the project is built without errors.

4. Observing Caching:

- Upon triggering subsequent workflow runs, observe whether Maven dependencies are reinstalled in the `install-dependencies` job or if they are retrieved from cache.
- Note any differences in workflow execution time between the initial run and subsequent runs, as caching can significantly reduce build times for subsequent runs.