GitHub Actions

3rd semester @ Erhvervsakademi København

Our Goal

> Automate tests runs and production of artifacts (docker image) using GitHub Actions

Question:

What have you done so far to automate tests, builds and deployments?

What is GitHub Actions?

> GitHub Actions lets you automate workflows like testing, building, or deploying code directly from events in a GitHub repository.

High level overview of GitHub Actions

> An event triggers a workflow defined in a YAML file, which consists of jobs that run on GitHub-hosted or self-hosted runners, executing steps that can be actions or shell commands.

```
Event (triggers) -> Workflow (YAML) -> Job -> Step (Action or Command)
```

Workflow

> A workflow is a YAML file that defines the automation.

Key facts:

- Stored in `.github/workflows/` directory
- One repository can have many workflows
- Each workflow responds to one or more events or triggers

Events

> An event is something that happens in your repository that starts a workflow.

Common examples:

- `push`
- `pull_request`
- `workflow_dispatch` (manual trigger)
- `workflow_run` (triggered by another workflow)
- `schedule` (triggered at specific times)

> **Note:**
> Events answer the question: *"When should this workflow run?"*

Events example

```yaml
name: CI
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
  workflow_dispatch:
```

The triggering events for this workflow are:

- Code is pushed to the `main` branch
- A pull request is opened against the `main` branch
- Manually triggered (`workflow_dispatch`) from the GitHub UI

Jobs

> A job is a set of steps that execute on the same runner.

Key facts:

- Each workflow can have multiple jobs running in parallel or sequentially
- Each job runs in its own virtual environment (runner)
- Jobs can depend on the completion of other jobs using `needs`

> **Note:**
> Jobs answer the question: *"What tasks should be performed when the workflow runs?"* i.e. build, test, deploy, etc.

Example of a job

```yaml
# ...
jobs:
  build:
    runs-on: ubuntu-latest
  # ...
  deploy:
    runs-on: ubuntu-latest
    needs: build
  # ...
```

In this example:

- The `build` job runs on the latest Ubuntu runner
- The `deploy` job also runs on the latest Ubuntu runner but only starts after the `build` job has successfully completed (due to `needs: build`)

Steps

> A step is a single task within a job, which can be an action or a shell command.

Key facts:

- A job consists of multiple steps that are executed sequentially within a job
- Each step can use an **action** (`uses`) or run a command directly in the shell (`run`)
- Steps can share data using outputs and environment variables

> **Note:**
> Steps are the instructions that define the actions to be performed within a job.

Example of steps

```yaml
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code # The code is available in the runner
        uses: actions/checkout@v5

      - name: Set up Java # The Java environment is set up in the runner
        uses: actions/setup-java@v5
        with:
          java-version: '25'

      - name: Run tests
        run: mvn -B test
```

> **Note:**
>
> The `run` executes a shell command in the runner.

Runnning multiple shell commands in a step

```
# ...
- name: Run multiple commands
  run: |
    echo "This is the first command"
    echo "This is the second command"
    echo "This is the third command"
```

> **Note:**
>
> The `|` allows you to run multiple commands in a single step, and they will be executed in the same shell session.

Actions - reusable units of code

> An action is a reusable unit of code that can be shared and used in workflows.

Key facts:

- Actions are just repositories that contain code to perform a specific task => these repositories live on GitHub and can be public or private
- You can use actions created by the GitHub community or create your own custom actions
- Actions can be used in steps with the `uses` keyword

Common used GitHub Actions

- actions/checkout - Checks out your repository, so your workflow can access it.
- actions/setup-java - Sets up a specific version of Java in your GitHub Actions runner environment.
- actions/cache - Caches dependencies and build outputs to improve workflow execution time.
- actions/upload-artifact - Uploads artifacts from a workflow for later use.
- actions/download-artifact - Downloads artifacts that were uploaded in a workflow.
- docker/build-push-action - Builds and pushes Docker images.
- docker/login-action - Logs into a Docker registry.
- docker/metadata-action - Generates metadata for Docker images.

SemVer (Semantic Versioning)

> Semantic versioning is a versioning scheme that uses a three-part version number: `MAJOR.MINOR.PATCH`.

Example: `v2.1.1`:

- **Major version**: Breaking changes, not backward compatible (e.g., `v2`)
- **Minor version**: New features, backward compatible (e.g., `v2.1`)
- **Patch version**: Bug fixes, backward compatible (e.g., `v2.1.1`)

Actions versioning

> When using an action, you can specify the version you want to use. This is important for stability and security.

Ways to specify action versions:

- **Tag**: `uses: actions/checkout@v5` (recommended for stability)
- **Branch**: `uses: actions/checkout@main` (not recommended for production workflows)
- **SHA**: `uses: actions/checkout@<commit-sha>` (most stable, but less readable)

Hashpinning

> Hashpinning is the practice of using a specific **commit SHA** to reference an action, ensuring that you are using the exact code you expect.

Benefits of hashpinning:

- **Security**: Prevents malicious code from being introduced if the action's repository is compromised later on
- **Stability**: Ensures that your workflow continues to work as expected, even if the action's repository is updated with breaking changes (deterministic builds)
- **Reproducibility**: Allows you to reproduce the same workflow behavior in the future, as the action's code will not change

See GitHub Actions reference guide for more details.

Caching nomenclature

> **Caching is the practice of storing a copy of frequently accessed or expensive-to-compute data in a temporary storage location (a cache) so it can be retrieved faster the next time it's needed.**

- **Cache key**: A unique identifier for the cache, often based on the contents of files.
- **Cache value**: The stored data associated with a cache key.
- **Cache hit**: A cache entry matching the requested key is found and successfully restored.
- **Cache miss**: No matching cache entry is found; the data must be recomputed or refetched.

Caching maven dependencies

> Caching dependencies can significantly speed up your workflow by reusing previously downloaded or built dependencies.

The `actions/cache` action allows you to cache files and directories in your workflow, which can be used to store dependencies like Maven's `.m2` directory.

Example of caching Maven dependencies

```yaml
- name: Cache Maven dependencies
  uses: actions/cache@cdf6c1fa76f9f475f3d7449005a359c84ca0f306
  with:
    path: ~/.m2/repository
    key: ${{ runner.os }}-maven-${{ hashFiles('**/pom.xml') }}
    restore-keys: |
      ${{ runner.os }}-maven-
```

- The cache key is generated based on the operating system and a hash of all `pom.xml` files, ensuring that the cache is updated whenever dependencies change.
- The `restore-keys` allows for partial matches, so if an exact cache key is not found, it can still restore a cache that matches the OS and Maven prefix.
- The `hashFiles()`: computes a hash based on the contents of the specified files, creating a unique cache key that changes when depdendency files change.

Permissions for a job

> Permissions define what actions a job can perform on the repository and other resources.

Key facts:

- By default, GitHub Actions jobs have read and write permissions to the repository, but you can customize these permissions for security reasons.
- You can specify permissions at the workflow or job level using the `permissions` key in your YAML file.
- Common permissions include `contents`, `actions`, `packages`, `issues`, `pull-requests`, etc.

Example of setting permissions for a job

```yaml
jobs:
  build:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: write
```

In this example:

- `contents: read` : The job has read access to the repository contents, allowing it to check out code and read files.
- `packages: write` : The job has write access to GitHub Packages, allowing it to publish packages or Docker images to GitHub Container Registry (GHCR).

Pushing docker images to GHCR

> GitHub Container Registry (GHCR) is a service provided by GitHub. It allows you to store and manage Docker container images directly within GitHub.

Steps in a job to push a Docker image to GHCR:

1. Log in to GHCR using `docker/login-action`
2. Extract metadata for the Docker image using `docker/metadata-action`
3. Build and push the Docker image using `docker/build-push-action`

Logging in to GHCR

```yaml
- name: Log in to GHCR
  uses: docker/login-action@v3
  with:
    registry: ghcr.io
    username: ${{ github.actor }}
    password: ${{ secrets.GITHUB_TOKEN }}
```

In this example:

- The `registry` is set to `ghcr.io`, which is the endpoint for GitHub Container Registry.
- The `username` is set to `${{ github.actor }}`, which is the GitHub username of the person or bot that triggered the workflow.
- The `password` is set to `${{ secrets.GITHUB_TOKEN }}`, which is a special token automatically generated by GitHub Actions.

Extracting metadata for the Docker image

```
- name: Extract metadata for Docker image
  id: meta
  uses: docker/metadata-action@v5
  with:
    images: ghcr.io/${{ github.repository }}
    tags: |
      type=ref,event=branch
      type=sha,prefix={{branch}}-
      type=raw,value=latest,enable={{is_default_branch}}
```

In this example:

- The `images` input specifies the name of the Docker image to be built and pushed, using the repository name as part of the image name.
- The `tags` input defines how the Docker image should be tagged based on the Git reference. Prodcues two tags - `latest` and `main-sha`.

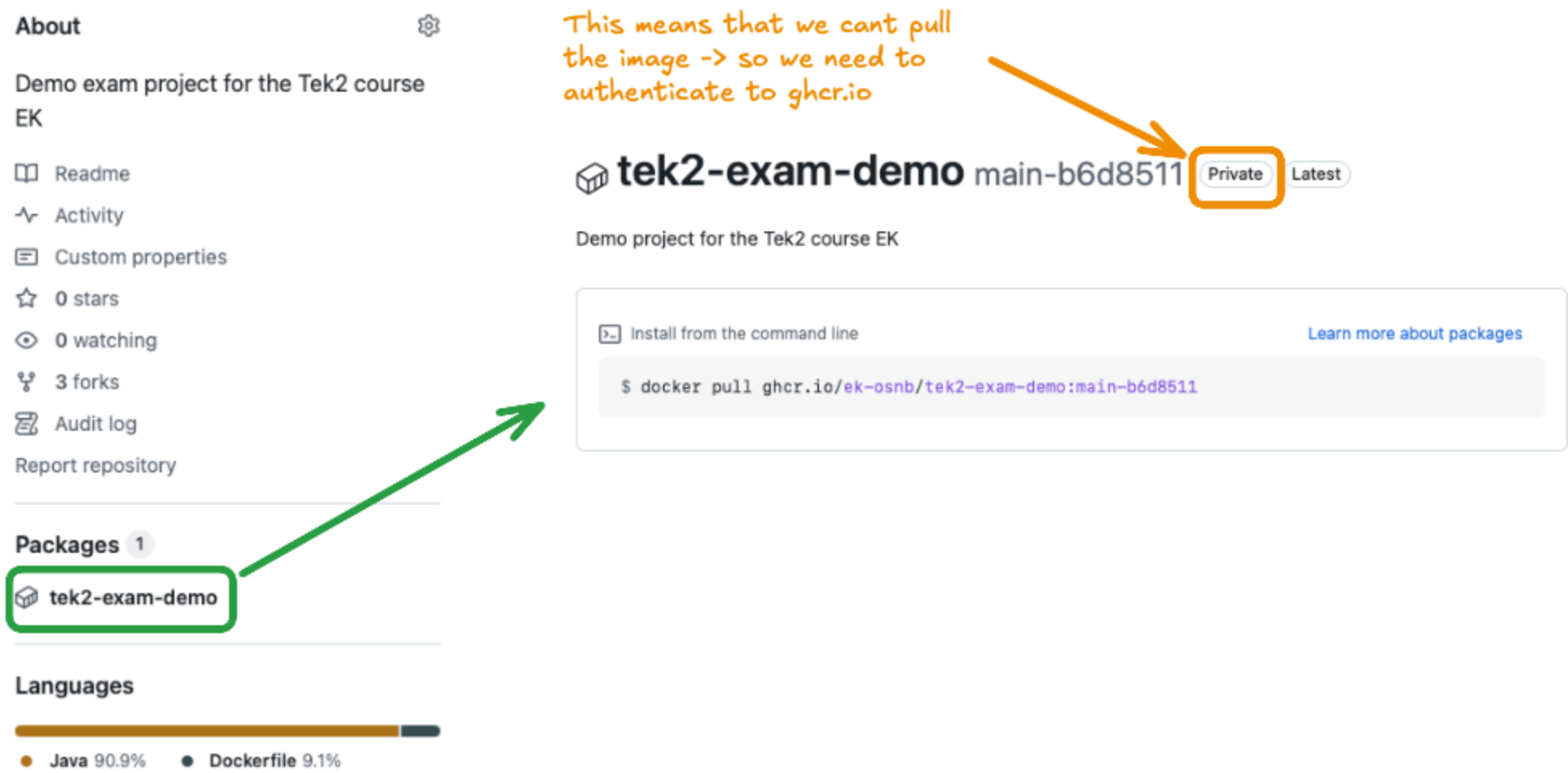
Building and pushing the Docker image

```
- name: Build and push Docker image
  uses: docker/build-push-action@v6
  with:
    context: .
    push: true
    tags: ${{ steps.meta.outputs.tags }}
    labels: ${{ steps.meta.outputs.labels }}
```

In this example:

- The `context` is set to `.`, indicating that the Docker build context is the root.
- The `push` input is set to `true`, indicating that the built Docker image should be pushed to the specified registry (GHCR in this case).
- The `tags` input uses the tags generated by the previous `docker/metadata-action` step.

Accessing the pushed Docker image



Creating a PAT (Personal Access Token) for GHCR

> A Personal Access Token (PAT) is a token that you can create in your GitHub account to authenticate with GitHub services, including GHCR.

Steps to create a PAT for GHCR:

1. Navigate to setting -> Developer settings -> Tokens (classic)
2. Generate a new token (name it `GHCR_PAT`) with the following scope:
    - `read:packages` (to pull packages)
3. Generate the token and copy it. Store it securely as you won't be able to see it again.

Authenticating with GHCR using a PAT

Authenticate with GHCR on your local machine:

```
export GHCR_PAT=your_personal_access_token_here
echo $GHCR_PAT | docker login ghcr.io -u your_github_username --password-stdin
```

Pull the Docker image:

```
docker pull ghcr.io/your_github_username/repository_name:latest
```

Logging out from GHCR:

```
docker logout ghcr.io
```

Exercises