D'accord, je vais fournir un exemple plus complet de l'implémentation du contrôleur de workflow CI/CD en Java, en utilisant l'Operator SDK. L'exemple inclura également des étapes pour le déploiement et le test du contrôleur dans Kubernetes.

Fichier CRD (config/crd/bases/cicd.example.com_workflows.yaml)

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: workflows.cicd.example.com
spec:
  group: cicd.example.com
  versions:
    - name: v1alpha1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                pipeline:
                  type: string
                repository:
                  type: string
                branch:
                  type: string
            status:
              type: object
              properties:
                status:
                  type: string
                message:
                  type: string
  scope: Namespaced
  names:
    plural: workflows
    singular: workflow
    kind: Workflow
    shortNames:
    - wf
```

Fichier Custom Resource et Contrôleur en Java

Workflow.java

```java
package com.example.cicd;

import io.fabric8.kubernetes.api.model.Namespaced;
import io.fabric8.kubernetes.client.CustomResource;

public class Workflow extends CustomResource<WorkflowSpec,
WorkflowStatus> implements Namespaced {}
```

WorkflowSpec.java

```java
package com.example.cicd;

public class WorkflowSpec {
    private String pipeline;
    private String repository;
    private String branch;

    // Getters and Setters
    public String getPipeline() {
        return pipeline;
    }

    public void setPipeline(String pipeline) {
        this.pipeline = pipeline;
    }

    public String getRepository() {
        return repository;
    }

    public void setRepository(String repository) {
        this.repository = repository;
    }

    public String getBranch() {
        return branch;
    }

    public void setBranch(String branch) {
        this.branch = branch;
    }
}
```

WorkflowStatus.java

```java
package com.example.cicd;
```

```
public class WorkflowStatus {
    private String status;
    private String message;

    // Getters and Setters
    public String getStatus() {
        return status;
    }

    public void setStatus(String status) {
        this.status = status;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

WorkflowReconciler.java

```
package com.example.cicd;

import io.fabric8.kubernetes.api.model.Pod;
import io.fabric8.kubernetes.api.model.PodBuilder;
import io.fabric8.kubernetes.client.KubernetesClient;
import io.javaoperatorsdk.operator.api.reconciler.Context;
import
io.javaoperatorsdk.operator.api.reconciler.ControllerConfiguration;
import io.javaoperatorsdk.operator.api.reconciler.Reconciler;
import io.javaoperatorsdk.operator.api.reconciler.UpdateControl;

import java.util.List;

@ControllerConfiguration(namespaces = "default")
public class WorkflowReconciler implements Reconciler<Workflow> {

    private final KubernetesClient client;

    public WorkflowReconciler(KubernetesClient client) {
        this.client = client;
    }

    @Override
    public UpdateControl<Workflow> reconcile(Workflow workflow, Context
context) {
        String pipeline = workflow.getSpec().getPipeline();
```

```
        String repository = workflow.getSpec().getRepository();
        String branch = workflow.getSpec().getBranch();

        System.out.println("Running pipeline " + pipeline + " for repo "
+ repository + " on branch " + branch);

        try {
            // Trigger the CI/CD pipeline by creating a Pod
            Pod pod = new PodBuilder()
                    .withNewMetadata()
                    .withName("cicd-pipeline-" +
workflow.getMetadata().getName())
                    .endMetadata()
                    .withNewSpec()
                    .addNewContainer()
                    .withName("pipeline-container")
                    .withImage("cicd-pipeline-image")

.addNewEnv().withName("PIPELINE").withValue(pipeline).endEnv()

.addNewEnv().withName("REPOSITORY").withValue(repository).endEnv()

.addNewEnv().withName("BRANCH").withValue(branch).endEnv()
                    .endContainer()
                    .endSpec()
                    .build();

            client.pods().inNamespace("default").createOrReplace(pod);

            // Update status
            workflow.setStatus(new WorkflowStatus());
            workflow.getStatus().setStatus("Running");
            workflow.getStatus().setMessage("Pipeline started
successfully");

        } catch (Exception e) {
            workflow.getStatus().setStatus("Failed");
            workflow.getStatus().setMessage(e.getMessage());
        }

        return UpdateControl.updateStatusSubResource(workflow);
    }
}
```

Configuration et Déploiement

Dockerfile

```
FROM adoptopenjdk/openjdk11:alpine
COPY target/java-operator-1.0-SNAPSHOT.jar /usr/src/myapp/java-
```

```
operator.jar
WORKDIR /usr/src/myapp
CMD ["java", "-jar", "java-operator.jar"]
```

Construction et Pousser l'Image Docker

```
./mvnw clean package
docker build -t your-operator-image:tag .
docker push your-operator-image:tag
```

Déploiement de l'Opérateur

Créez les fichiers nécessaires pour le déploiement de l'opérateur dans Kubernetes.

manager.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: java-operator
spec:
  replicas: 1
  selector:
    matchLabels:
      name: java-operator
  template:
    metadata:
      labels:
        name: java-operator
    spec:
      containers:
      - name: java-operator
        image: your-operator-image:tag
        imagePullPolicy: Always
        ports:
        - containerPort: 8080
        env:
        - name: WATCH_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
        - name: POD_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        - name: OPERATOR_NAME
          value: "java-operator"
```

Déployez les CRD et l'opérateur :

```
kubectl apply -f config/crd/bases/cicd.example.com_workflows.yaml
kubectl apply -f config/manager/manager.yaml
```

Création d'une Instance de Workflow

Créez un fichier `workflow.yaml` pour définir une instance de votre CRD :

```
apiVersion: cicd.example.com/v1alpha1
kind: Workflow
metadata:
  name: example-workflow
spec:
  pipeline: "example-pipeline"
  repository: "https://github.com/example/repo.git"
  branch: "main"
```

Appliquez cette configuration :

```
kubectl apply -f workflow.yaml
```

Avec cela, vous aurez un opérateur Kubernetes complet en Java, capable de gérer un workflow CI/CD en utilisant l'Operator SDK. Vous pouvez adapter et étendre cette configuration en fonction de vos besoins spécifiques.