Making on-prem infrastructure feel Cloud-Native with the Java Operator SDK

Valentin Delaye

valentin.delaye@elca.ch

KCD Suisse Romande

DECEMBER 4 & 5, 2025 – CERN, GENEVA

jonesbusy
valentin.delaye

- Valentin Delaye
- Platform Engineer at ELCA
- Jenkins maintainer and Governance Board
- ORAS Java SDK maintainer
- Google Summer of Code Mentor (2024 & 2025)

Agenda

- `What` vs `How` and `Developer Experience`
- `K8S Operators` and the `KRM` model
- `Java Operator SDK` and it's `Quarkus` extension
- A `GitOps` and (`BackstageOps`) approach via `FluxCD`
- Examples with K8S internal and external resources
- Q&A

What vs How (1)

- I need to connect my app to a database
- I need persistent storage for my service
- I need my app to be highly available

- I need to know how to deploy a database
- I need to know which storage class to use and how to provision it
- I need to know how many replicas to choose, cluster topology, etc.

> **Tip**
> Developers only define intent in manifests. Platform handles the rest.

Helm to the rescue?

- Package your Kubernetes resources as single deployable unit
- Define values to customize your deployments per environment

What vs How (2)

```yaml
apiVersion: helm.toolkit.fluxcd.io/v2
kind: HelmRelease
metadata:
  name: my-app
spec:
  chart:
      ...
  values:
    ...
    tag: latest
    replicas: 1
    database:
      external: false
      resources:
         limits:
           memory: "512Mi"
```

> ⚠ **Warning**
> Different values for different environments (dev/prod)
> More abstraction needed

```yaml
apiVersion: helm.toolkit.fluxcd.io/v2
kind: HelmRelease
metadata:
  name: my-app
spec:
  chart:
      ...
  values:
    ...
    tag: 0.0.1
    replicas: 3
    affinity:
      podAntiAffinity:
        ...
    database:
      external: true
      host: db.prod.internal
      port: 5432
      name: myappdb
      user: myappuser
```


What vs How (3)

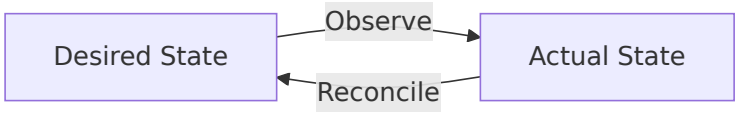
```yaml
apiVersion: app.elca.ch/v1alpha1
kind: MySuperApp
metadata:
  name: todo-app-1
spec:
  tag: latest
```

```yaml
apiVersion: app.elca.ch/v1alpha1
kind: MySuperApp
metadata:
  name: todo-app-1
spec:
  tag: 0.0.1
```

A Kubernetes Operator could

- Deploy a Database on K8S or provision a new schema on an external cluster (external resource)
- Drop a `ConfigMap` and `Secret` with required connection info
- Set the correct number of replicas, affinity rules, monitoring, etc.

Kubernetes reconcile loop

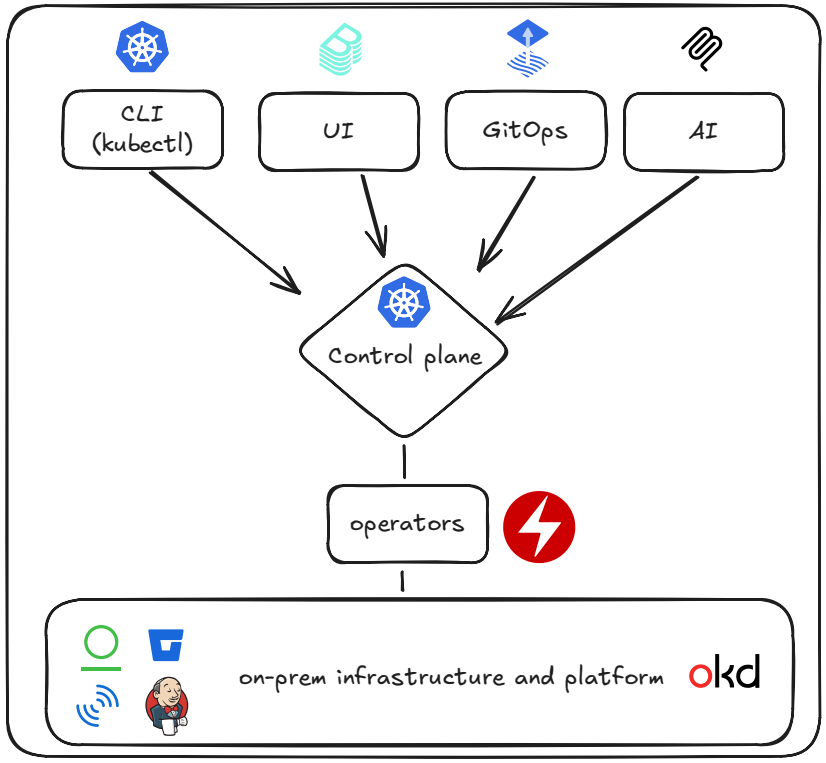


- Declarative (YAML manifests)
- Immutable
- Composable
- Versioned (GitOps)
- Extensible (KRM)

KRM and developer platform

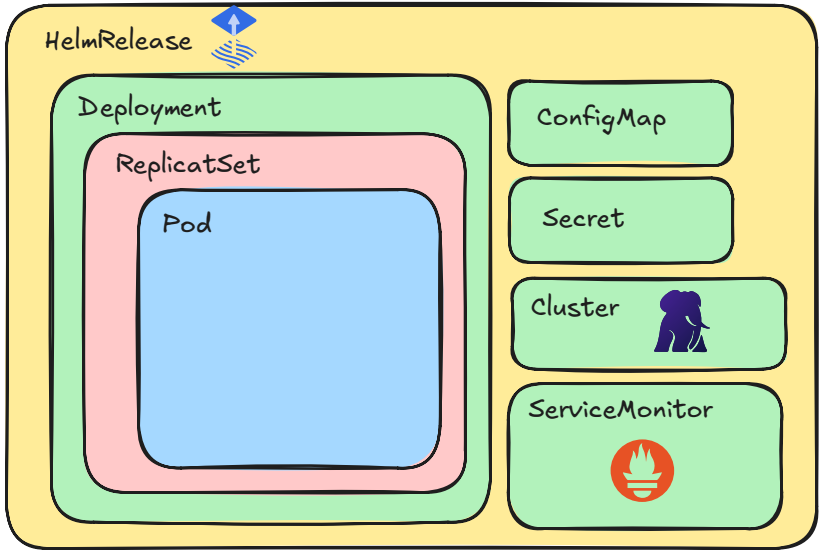
Everything is a resource

- An application
- A database (inside or outside K8S)
- A firewall rule
- An image repository
- A DNS entry
- A service account token
- etc.



Dependent resource

- A `Deployment` managing `ReplicaSet` which in turn manages `Pods` (High level vs low level resource)
- High level abstractions or workflows
- `owner` references and garbage collection (across same namespace only)
- `Helm` is some kind of abstraction too! (On client side, no permanent reconciliation, no drift detection, etc.)
- `Flux CD` resource `HelmRelease` fix some of the issues with `Helm` (server side



A custom resource

```java
@Group(TodoApp.GROUP)
@Version(TodoApp.VERSION)
@Plural(TodoApp.PLURAL)
@Kind(TodoApp.KIND)
public class TodoApp
    extends CustomResource<TodoAppSpec, TodoAppStatus>
    implements Namespaced {

    public static final String KIND = "TodoApp";
    public static final String LOWER_NAME = "todoapp";
    public static final String PLURAL = "todoapps";
    public static final String GROUP = "apps.sso.elca.ch";
    public static final String VERSION = "v1";

}
```

> ⓘ **Note**
> Fabric8 CRDGenerator to the rescue!

```yaml
apiVersion: "apiextensions.k8s.io/v1"
kind: "CustomResourceDefinition"
metadata:
  name: "todoapps.samples.elca.ch"
spec:
  group: "samples.elca.ch"
  names:
    kind: "TodoApp"
    plural: "todoapps"
    singular: "todoapp"
  scope: "Namespaced"
  versions:
    name: "v1"
    schema:
      openAPIV3Schema:
        properties:
          spec:
            properties:
              ...
          status:
            ...
```


Dependent resource (1)

```java
public final class DatabaseResource {

    private final String name;

    public DatabaseResource(String name) { this.name = name; }

    public String getName() { return name; }

    @Override
    public boolean equals(Object o) {
        if (o == null || getClass() != o.getClass()) return false;
        DatabaseResource that = (DatabaseResource) o;
        return Objects.equals(name, that.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name);
    }
}
```

⚠ **Warning**

Dependent resource (2)

```java
@KubernetesDependent
public class ExternalDatabaseDependent extends PerResourcePollingDependentResource<DatabaseResource, TodoApp>
        implements Creator<DatabaseResource, TodoApp> { // Traits

    private DatabaseService databaseService;

    public ExternalDatabaseDependent(DatabaseService databaseService) {
        super(DatabaseResource.class, Duration.ofHours(1)); // Polling interval
        this.databaseService = databaseService;
    }

    @Override
    protected DatabaseResource desired(TodoApp primary, Context<TodoApp> context) {
        return new DatabaseResource(primary.getSpec().group());
    }

    @Override
    public DatabaseResource create(DatabaseResource resource, TodoApp primary, Context<TodoApp> context) {
        return databaseService.createDatabase(resource);
    }

    @Override
    public Set<DatabaseResource> fetchResources(TodoApp primary) {
```


Dependent resource (3)

```java
@KubernetesDependent
public class KubernetesDatabaseDependent extends CRUDKubernetesDependentResource<Deployment, TodoApp> {

    @Override
    protected Deployment desired(TodoApp primary, Context<TodoApp> context) {
            return new DeploymentBuilder()
                .withNewMetadata()
                .withName(primary.getMetadata().getName() + "-db")
                ...
                .withNewSpec()
                .addNewContainer()
                .withName("postgres")
                .withImage("postgres:18")
                .addNewPort().withContainerPort(5432).endPort()
                .endContainer()
                .endSpec()
                .endTemplate()
                .endSpec()
                .build();
    }
}
```


Dependent resource (4)

```java
@ControllerConfiguration(
        dependents = {
            @Dependent(type = ExternalDatabaseDependent.class, reconcileCondition = IsProductionCondition.class),
            @Dependent(type = KubernetesDatabaseDependent.class, reconcileCondition = IsDevelopmentCondition.class
            @Dependent(type = ConfigMapDependent.class, dependsOn = {ExternalDatabaseDependent.class, DatabaseDepe
        }
)
public class TodoAppReconciler implements Reconciler<TodoApp> {

    @Override
    public UpdateControl<TodoApp> reconcile(TodoApp resource, Context<TodoApp> context) {

        final var result = context.managedWorkflowAndDependentResourceContext()
                .getWorkflowReconcileResult()
                .orElseThrow()
                .allDependentResourcesReady();

        if (!result) {
            return UpdateControl.patchStatus(resource.withStatus("NotReady")).rescheduleAfter(5L, TimeUnit.SECONDS
        }
        if (resource.getStatus().equals("Ready")) {
            return UpdateControl.noUpdate();
        }
```

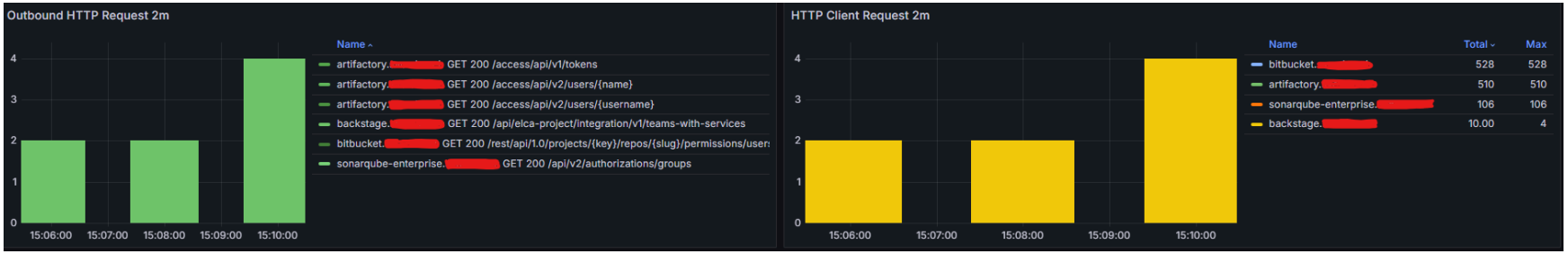

What we learned (1)

- Use `conditions` in `status` to reflect the progress of reconciliation (even for simple resources)

```yaml
status:
  conditions:
  - lastTransitionTime: "2025-10-07T11:49:44.177461113Z"
    message: Deployed TodoApp
    observedGeneration: 1
    status: "True"
    type: Done
  - status: "False"
    type: Started
  - status: "False"
    type: HasErrors
  deployedAppVersion: 0.0.11
  deployedVersion: 0.16.1
  lastHandledReconcileAt: "2025-10-07T11:49:44.192938122Z"
  location: my-todo-app.apps.my-domain.elca.ch
  observedGeneration: 1
```


What we learned (2)

- Monitor closely external resource polling intervals (equality, outbound HTTP requests, etc.)

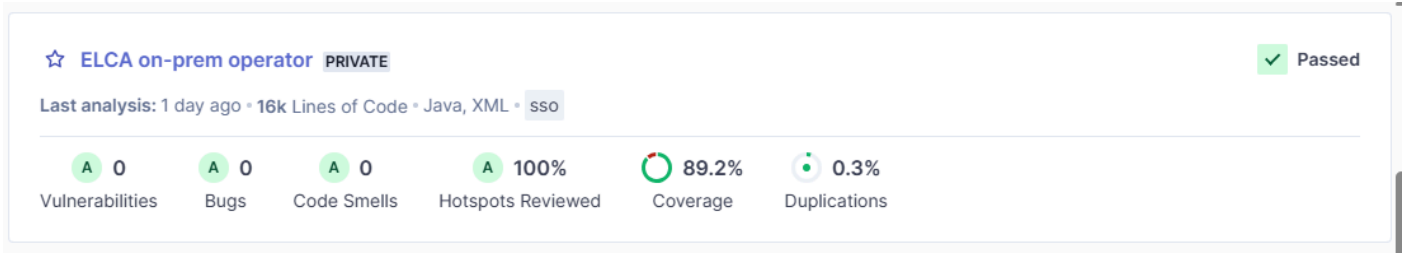


- Controller logs are not very useful for end-users, use events

```
Events:
  Type    Reason  Age    From              Message
  ----    ------  ----   ----              -------
  Normal  Ready   177m   on-prem-operator  Synchronized Artifactory groups from Backstage
  Normal  Ready   172m   on-prem-operator  Synchronized Artifactory groups from Backstage
```


What we learned (3)

- Test your controllers!



- Additional columns are useful to give instant feedback to consumers

```
kubectl get todoapps.samples.elca.ch

NAME          STATUS             DONE   APP VERSION   VERSION   LOCATION
my-todo-app   Deployed TodoApp   True   0.0.11        0.16.1    my-todo-app.apps.my-domain.elca.ch
```


Challenges we faced

Technical

- Learning curve
- Many best practices to follow to implement a good operator
- Expertise in Kubernetes internals
- Migrate existing workflows/automation to custom resources

Non-technical

- Culture change and mindset for (Dev vs Ops)
- Non-cloud-native apps and legacy systems

Current state

- ~ `7500` CR managed (and growing!)
- ~ `15` minutes to check all CRs state with max 5 reconciliations in parallel (restart)
- ~ Less that `half second` start time with Quarkus native image
- ~ `512 MB` memory for the operator

Thank you!

valentin.delaye@elca.ch
valentin.delaye
jonesbusy

References

- Java Operator SDK Documentation
- API conditions and conventions
- KRM-Native GitOps: Yes — Without Flux, No. (FluxCD or Nothing.)