


KubeCon | CloudNativeCon

North America 2023

Deep Dive Into Writing a Kubernetes Operator

Let’s Avoid Data Loss and Down Times!

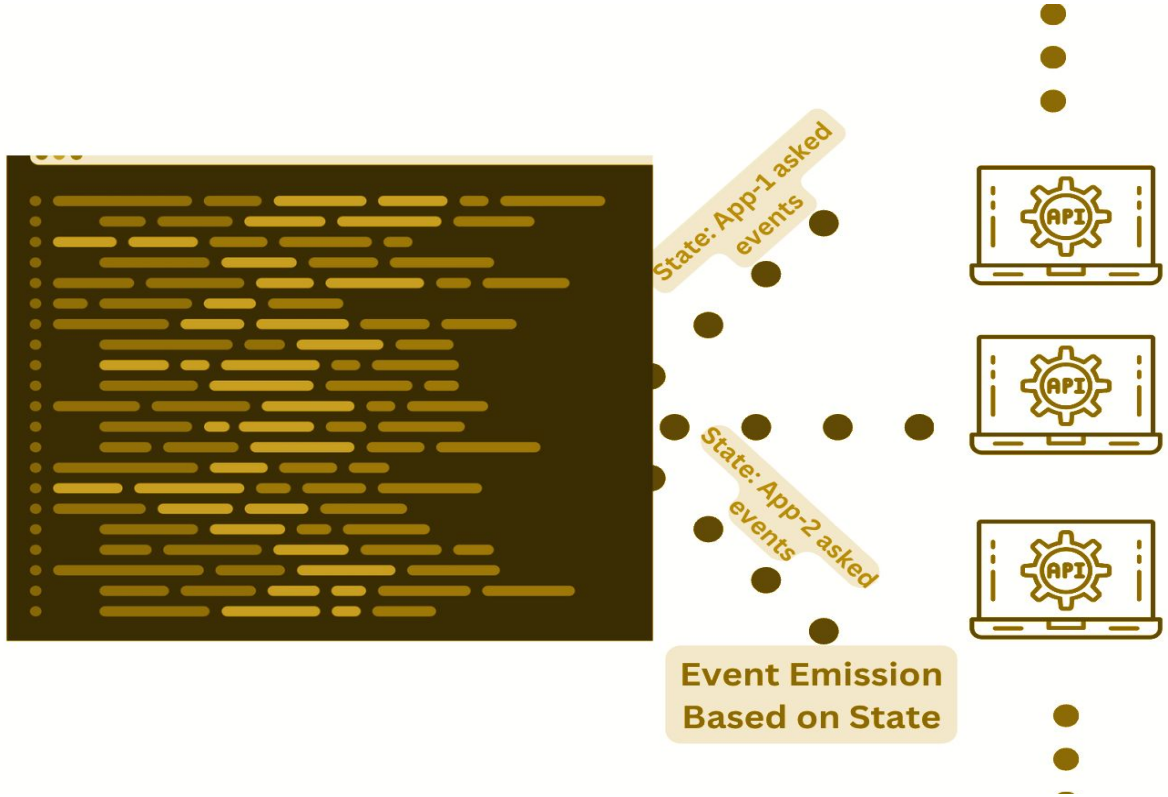
Shivangi Motwani

SpringTown AI

CONTENT

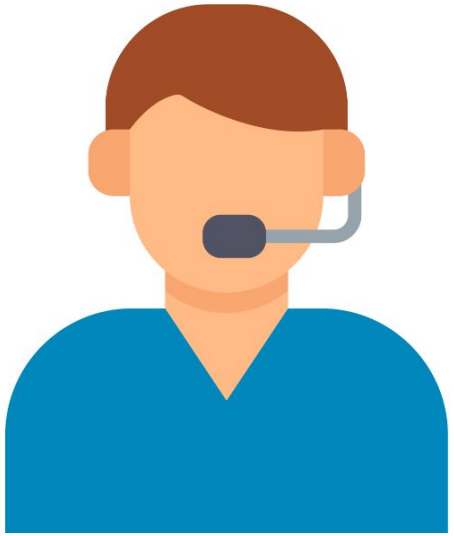
- Introduction and Problem Context
- Requirements & Attempted Solutions
- Transition to Kubernetes Operator
- Encountered Challenges and Solutions
- Additional Resources
- Recap and Conclusion
- Question and Answer Session

Problem Context- Understanding Application



- Complex Application
- Maintaining
  - Multiple States
  - For Multiple Tenants
    - In Multiple Zones

Who Is Managing These States?



ANAND

- Anand on the day time had to look around
  - **15 tickets** on good days,
  - **50+ changes** on the bad days

- If there is service deploying at night
  - Work ad hoc hours

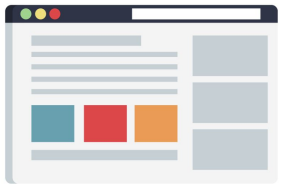
What We Needed As a Solution?

1. Address Anand State Change Tickets:
   a. Clearly define ownership responsibilities for efficient handling.
   b. Assign ownership to specific service teams.
2. Ensure Security:
   a. Implement measures to prevent compromise of security.
   b. Regularly update access controls and permissions.
3. Self-Serve Solution:
   a. Introduce a user-friendly self-service system.
   b. Develop a comprehensive knowledge base for issue resolution.
   c. Encourage teams to independently troubleshoot common problems.
4. Automated Resource Provisioning:
   a. Resource provisioning to be automated based on tenant creation.
5. Decentralized Resource Management:
   a. Each application team responsible for managing their own resources independently.

Attempted Solutions

1. Human Resource Hiring
    a. Attempts at knowledge sharing for the complex application were ineffective.
    b. Time invested did not yield satisfactory results.

2. Exploration of Jenkins Automation
    a. Encountered security concerns with authentication and auditing in the Jenkins solution.
    b. Deemed it not secure for the application's needs.



3. UI-based State Creation:
    a. Auditing problems persisted with the implemented solutions.
    b. Faced challenges, particularly in handling **duplication issues**

Transition to Kubernetes Operator

1. Managing States:
   a. Identified the need to efficiently manage application states.
2. Recognition of Repetitive Task:
   a. Acknowledged the repetitive nature of the task, prompting the decision to automate.
3. Custom Resource Definition (CRD) Conversion for a declarative model
4. Gap in Built-in Controllers:
   a. Found that our unique use case was not accommodated by existing built-in controllers.
5. Challenges Encountered:
   a. **Resource Overhead:** Faced challenges related to resource utilization.
   b. **Learning Curve:** Encountered difficulties in managing the learning curve associated with the chosen approach.

Phase 1 - Basic SetUp

1. CRD Writing and Finalization: 

    a. Utilized Kube-builder framework for initial boilerplate.
    b. Developed custom **logic** and **validation** tailored to our use case.
2. Local Testing:
    a. **Tested** the CRD and custom logic in a **local** environment.

Custom Resource Definition Considerations


```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: mycustomresources.example.com
spec:
  group: example.com
  version: v1
  scope: Namespaced  # or Cluster
  names:
    plural: mycustomresources
    singular: mycustomresource
    kind: MyCustomResource
    shortNames:
    - mcr
```

Custom Resource Definition Considerations


```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: mycustomresources.example.com
spec:
  group: example.com
  version: v1
  scope: Namespaced  # or Cluster
  names:
    plural: mycustomresources
    singular: mycustomresource
    kind: MyCustomResource
    shortNames:
    - mcr
```


```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: mycustomresources.example.com
spec:
  group: example.com
  version: v1
  scope: Namespaced  # or Cluster
  names:
    plural: mycustomresources
    singular: mycustomresource
    kind: MyCustomResource
    shortNames:
    - mcr
```

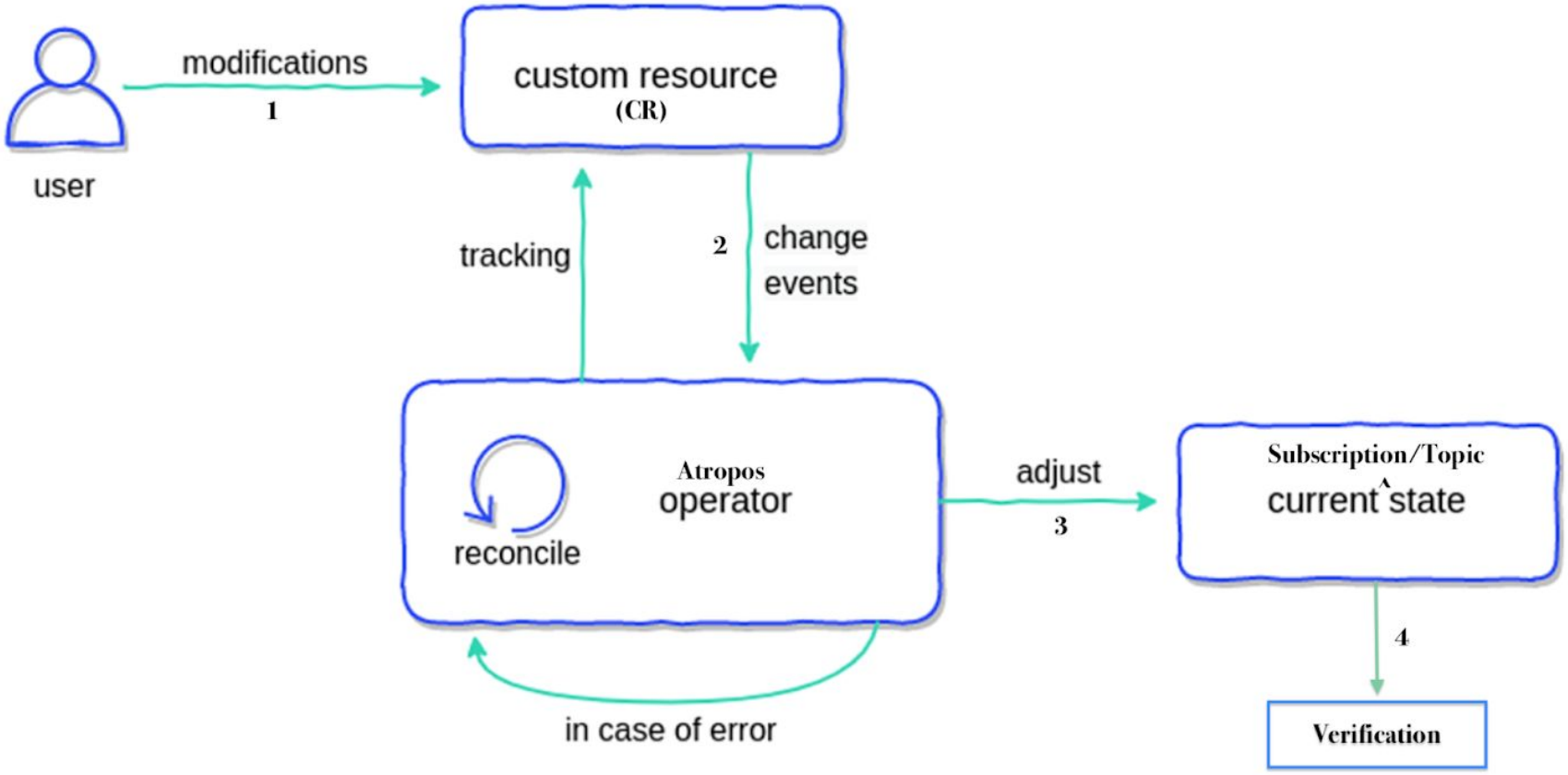
Custom Resource Definition Considerations


```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: mycustomresources.example.com
spec:
  group: example.com
  version: v1
  scope: Namespaced  # or Cluster
  names:
    plural: mycustomresources
    singular: mycustomresource
    kind: MyCustomResource
    shortNames:
    - mcr
```


```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: mycustomresources.example.com
spec:
  group: example.com
  version: v1
  scope: Namespaced  # or Cluster
  names:
    plural: mycustomresources
    singular: mycustomresource
    kind: MyCustomResource
    shortNames:
    - mcr
```


```yaml
status:
  acceptedNames:
    kind: ""
    plural: ""
  conditions: []
  storedVersions: []
```

How a Kubernetes Operator Functions?



Phase 2 - Handling Upgrades

- We were testing:
  - In **PreProd** Environment
  - For **5000+ CR** Instances
  - With **Scope** as **Cluster**

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: mycustomresources.example.com
spec:
  group: example.com
  version: v1
  scope: Namespaced  # or Cluster
  names:
    plural: mycustomresources
    singular: mycustomresource
    kind: MyCustomResource
    shortNames:
    - mcr
```

Phase 2 - Handling Upgrades

1. Security and Permission Challenges:
    a. Identified security and permission concerns, especially concerning different developer roles, within the Cluster Scope.
2. **Forced** Migration:
    a. The identified concerns compelled us to migrate from Cluster Scope to Namespace Scoped operations.

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: mycustomresources.example.com
spec:
  group: example.com
  version: v1
  scope: Namespaced  # or Cluster
  names:
    plural: mycustomresources
    singular: mycustomresource
    kind: MyCustomResource
    shortNames:
    - mcr
```

- We were testing:
    - In **PreProd** Environment
    - For **5000+ CR** Instances
    - With **Scope** as **Cluster**

Phase 2 - Handling Upgrades

- Understanding relationship between your k8s package manager and CRD

Phase 2 - Handling Upgrades

- Understanding relationship between your k8s package manager and CRD
- Only Option was to Migrate 

SPLIT BRAIN due to state difference 

Phase 3 - Handling Failures, BackUps, Recovery

SPLIT BRAIN due to state difference

- Avoid(Frequent) Migration
- Plan Migrations well
- Don’t play with CR Finaliser

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: finalizer-test
  namespace: myproject
  labels:
    app: finalizer-test
  finalizers:
    - finalizer.extensions/v1beta1
spec:
  selector:
    matchLabels:
      app: finalizer-test
  replicas: 3
  template:
    metadata:
      labels:
        app: finalizer-test
    spec:
      containers:
        - name: hieveryone
          image: openshiftkatacoda/blog-django-py
          imagePullPolicy: Always
          ports:
            - name: helloworldport
              containerPort: 8080
```

HELM Docs

Home Docs Charts Blog Community

Search...

Docs Home

Introduction

installation step, you can pass the `--skip-crds` flag.

Some caveats (and explanations)

There is no support at this time for upgrading or deleting CRDs using Helm.

- Understanding relationship between your k8s package manager and CRD
- Only Option was to Migrate 

Phase 4 - Insights

Understanding the new solution and process is difficult

1. Documentation Improvement
   a. External Teams
   b. Internal Team
2. Latency Challenges:
   a. Efficient Concurrency Handling

      `max-concurrent-evals`
3. Debugging Challenges
   a. Add Proper **Error Logs**
   b. Adding Prometheus **Metrics**
   c. Setting Up **Alerts**

Phase 4 - Insights

Understanding the new solution and process is difficult

1. Downstream Application Overload:
   a. Configurable Concurrency
   b. Rate Limiting Implementation
2. Stale Resource Cleanup:
   a. Systematic cleanup of stale or unused resources
3. Maintaining Resource State:
   a. Helped us cleanup state and implement retry strategy properly
4. E2E Testing Suite

Additional Resources

1. Stop Writing Operators from KubeCon + CloudNativeCon Europe 2021 Virtual from May 4–7, 2021
2. When Not to Write Kubernetes Operator
3. To Crd, or Not to Crd, That is the Question by Ed King & Sam Gunaratne, Pivotal
4. Stop Messing With Kubernetes Finalizers by Martin Heinz

RECAP

1. Explore Alternatives
    a. Investigate options beyond writing a Kubernetes Operator
2. Accurate Estimations
3. CRD Considerations
    a. Prioritize avoiding migrations when working with Custom Resource Definitions (CRD).
4. Thorough Documentation
    a. Emphasize comprehensive documentation for clarity and reference.
5. Concurrency Management
6. Effective Observability
7. Resource Cleanup
8. End-to-End (E2E) Testing

THANK YOU...

KubeCon CloudNativeCon North America 2023

Access Slides Here:

Send You Feedback hear:

Connect With Speaker: https://shivangim.github.io/contact