Aggregate CRD Design Options

OpenAPI Operator Generator

This document outlines design options for adding an aggregate CRD feature to the OpenAPI operator generator. An aggregate CRD takes one or more generated CRDs as input and combines them for unified management or observation.

Option 1: Composition CRD (Reference-Based)

A CRD that references existing Custom Resources by name/namespace and aggregates their statuses.

```
apiVersion: petstore.example.com/v1alpha1
kind: PetStoreAggregate
metadata:
  name: my-store-health
spec:
  resources:
    - kind: Pet
      name: my-pet
      namespace: default
    - kind: Order
      name: my-order
      namespace: default
  aggregationStrategy: AllHealthy  # or AnyHealthy, Quorum
status:
  state: Synced  # Aggregated state
  resourceStatuses:
    - kind: Pet
      name: my-pet
      state: Synced
      externalID: "123"
    - kind: Order
      name: my-order
      state: Failed
      message: "API returned 404"
  conditions: [...]
```

Pros:

- Clean separation - resources managed independently
- Leverages existing AllHealthy pattern in pkg/endpoint/resolver.go
- Easy to understand ownership model

Cons:

- Requires watching multiple resource types
- Resources must exist before aggregate

Option 2: Inline Composition CRD (Embedded Specs)

An aggregate CRD that embeds the specs of child resources directly, creating multiple API resources from a single CR.

```
apiVersion: petstore.example.com/v1alpha1
kind: PetBundle
metadata:
  name: full-pet-setup
spec:
  pet:
    name: "Fido"
    status: available
  order:
    petId: "{{.pet.externalID}}"  # Template reference
    quantity: 1
status:
  pet:
    state: Synced
    externalID: "123"
  order:
    state: Synced
    externalID: "456"
```

Pros:

- Single CR creates multiple API resources
- Can express dependencies between resources (order references pet ID)
- Familiar pattern (similar to Helm charts, Argo Application)

Cons:

- More complex reconciliation logic (ordering, dependency resolution)
- Harder to manage partial failures
- Spec becomes large

Option 3: Declarative Bundle (Template-Based)

A meta-CRD that declares a bundle template, instantiated by a separate CR. Provides reusable templates with parameters.

```
# Template definition
apiVersion: petstore.example.com/v1alpha1
kind: PetBundleTemplate
metadata:
  name: pet-with-order
spec:
  parameters:
    - name: petName
      type: string
    - name: quantity
      type: integer
      default: 1
  resources:
    - apiVersion: petstore.example.com/v1alpha1
      kind: Pet
      spec:
        name: "{{ .petName }}"
    - apiVersion: petstore.example.com/v1alpha1
      kind: Order
      spec:
        petId: "{{ .resources.Pet.status.externalID }}"
        quantity: "{{ .quantity }}"
---
# Instance
apiVersion: petstore.example.com/v1alpha1
kind: PetBundle
metadata:
  name: my-bundle
spec:
  templateRef: pet-with-order
  parameters:
    petName: "Buddy"
    quantity: 2
```

Pros:

- Reusable templates
- Clear separation of template definition and instantiation
- Parameters enable customization

Cons:

- Most complex to implement
- Two new CRD types needed
- Template language adds complexity

Option 4: Status Aggregator (Read-Only)

A lightweight CRD that only observes and aggregates status from existing resources. Similar to the existing ReadOnly mode.

```
apiVersion: petstore.example.com/v1alpha1
kind: ResourceHealthCheck
metadata:
  name: store-health
spec:
  selector:
    matchLabels:
      app: petstore
  # OR explicit list
  resources:
    - group: petstore.example.com
      kind: Pet
    - group: petstore.example.com
      kind: Order
  aggregation:
    type: Summary  # or Detailed
status:
  summary:
    total: 10
    synced: 8
    failed: 2
  conditions:
    - type: AllHealthy
      status: "False"
      reason: "2 resources failed"
```

Pros:

- Simplest to implement - read-only aggregation
- Builds on existing ReadOnly mode pattern
- Good for dashboards/monitoring

Cons:

- Cannot create/manage resources
- Limited to status observation

Option 5: Workflow/Pipeline CRD

An ordered sequence of operations with dependencies. Steps execute in order based on dependency graph.

```
apiVersion: petstore.example.com/v1alpha1
kind: PetWorkflow
metadata:
  name: onboard-pet
spec:
  steps:
    - name: create-pet
      resource:
        kind: Pet
        spec:
          name: "Fido"
          status: pending
    - name: upload-image
      dependsOn: [create-pet]
      action:
        kind: PetUploadImage
        spec:
          petId: "{{ steps.create-pet.externalID }}"
          file: "..."
    - name: mark-available
      dependsOn: [upload-image]
      resource:
        kind: Pet
        spec:
          name: "Fido"
          status: available
status:
  phase: Running  # Pending, Running, Succeeded, Failed
  steps:
    create-pet: Completed
    upload-image: Running
    mark-available: Pending
```

Pros:

- Expresses ordered operations naturally
- Good fit for Action endpoints (one-shot operations)
- Clear execution semantics

Cons:

- Most complex reconciliation logic
- Need DAG evaluation
- State machine complexity

Implementation Considerations

Aspect	Current Support	What's Needed
Multi-resource status	Yes (Responses map)	Extend to cross-CRD

Type references	Yes (UsesSharedType)	Cross-CRD references
Health aggregation	Yes (AllHealthy strategy)	Apply to CRs not endpoints
Watches	Single type	Multi-type watches
Templates	Go templates	Add aggregate templates

Recommendation

For a first iteration, **Option 1 (Composition CRD)** or **Option 4 (Status Aggregator)** are recommended:

- **Option 1 -** If you need the aggregate to manage lifecycle (create/delete child resources)
- **Option 4 -** If you only need observability across resources

Both leverage existing patterns (Responses map, AllHealthy strategy, ReadOnly mode) and require minimal new concepts.