Improving Developer Velocity with Simplicity

OSM: Refactoring

Authors:
seanteeling@microsoft.com
Reviewers: *add yourself*
Date Modified: **06/08/2022**
Status: Draft | In Review | **Approved** | Abandoned

Foreword

Before continuing with this doc I'd like to make it clear that any critiques of existing code, architecture designs or patterns are in no way intended to disparage any of the great work the maintainers have done. Quite the opposite; I believe that foreseeing correct lines of abstraction in a project of this magnitude, from an early stage is near impossible, and that any work of this size, particularly developed this quickly and with this many developers, will always benefit from a large refactoring. Hindsight is always 20/20.

Glossary

Data Objects - objects that only contain data. They do not perform any side effects, or reach out to other services. They contain little to no business logic[1]. OSM Objects, K8s Objects, and Envoy configs would all be an example of a data object. Examples of data objects include

- **K8s Objects** - I refer to types that represent the underlying configuration of a Kubernetes Object, ie: a Pod, as a K8s object.
- **OSM Object** - I refer to types that are strictly used within the OSM codebase as OSM objects. Examples of these include Service, Endpoint, and TrafficResource. These do not come from any underlying API, but are generated within the code, and should map to internal concepts within OSM, to decouple us from external API's.
- **Envoy Configs** - I refer to objects/types that are tied to the Envoy API as Envoy Configs.

Builders - object types that are responsible for generating output that may have substantial differences based on large numbers of inputs, or on permutations of those inputs.

Client Objects - objects that interact with other external processes via API calls

---

[1] A certificate object `IsExpired()` might check if the expiration time has elapsed. These methods should be helper methods that answer questions about the data, and nothing else.

Composer - (where the business logic is) a type, method, or function that may deal with any number of different types of objects. That is, a composer can call out to a client to get data, feed that data into a builder, and return the associated data object.

Motivation

Why should OSM refactor now? We are at a pivotal point, evaluating new features like Rate Limiting, MultiCluster, custom envoy filters, and VM integrations. The current code base lacks strict boundaries of responsibility among packages, the readability bar is high, and developer velocity slowed down. By dedicating a small portion of our headcount to the refactoring proposals described below, I believe we can clear a path for both more robust and quicker implementations of the aforementioned features, improve developer understanding of our code base, and promote software engineering best practices.

Goals & Guiding Principles

The number one goal of this doc is to improve developer velocity by creating intuitive layers in the code with clear lines of responsibility.

We believe we can create the simplest code base by leveraging the following principles:

- Everything has exactly 1 responsibility. Data objects, builders, clients, or composers all do 1 thing. No one type should have more than 1 responsibility from the above.
- Complex data objects, particularly those with a myriad of inputs that can result in different outputs/versions of that data object, should leverage a builder object to create it.
- Each package should have little to no concern or influence about what uses it. Each package is a stand alone entity.
    - This ties closely to the principle that interfaces should be defined where they are consumed not by where they are implemented.
- Each package should deal with 1 type of input, and 1 type of output.
    - Packages that contain business logic, should deal with a singular type of input/output (ie: composers).
    - In this instance, a “type of input” is a layer, ie: in a rest API, the layers may be: 1) API objects, 2) “internal objects”, 3) DAO (data access objects). This decouples the API from business logic, and from database.
    - Similarly, OSM has K8s, Internal, and Envoy. Any business logic should apply strictly on the Internal layer, meaning a composer has K8s/Envoy abstracted away from them.

- Each package should map to a real-world construct. A good litmus test is that each package is easy to describe in a single sentence.
- If a method does not access any fields of its instance, it should be a function.
    - This makes it simpler to determine the set of inputs to that function, and deeper function calls down the stack.
- Prefer unexported fields over exported fields.
- Avoid passing client objects as params, except when leveraging dependency injection. Instead, embed the client object into a separate client object. This allows us to use interfaces to abstract the client.
    - For client objects that are themselves fields on a struct, this makes it much easier to see where the client object is being used, through code navigation.
- Avoid passing a structs fields as parameters to downstream methods on that struct. This makes it easier for code navigation to tell you where each field is used.
- Avoid deeply nested function calls. Code should not branch out into millions of paths.
- Accept interfaces, return structs.
    - Assume your caller is smart. If they want the interface they can use that, but returning a struct provides more power.
- Prefer tests on exported functions/methods.
    - Unit testing internal package methods and functions belies too much importance on implementation details. Unit testing should test the inputs/outputs of the package.
- YAGNI - “You ain’t gonna need it”
- WET - “Write everything twice”
    - Similar to Rob Pike’s “A little copying is better than a little dependency”
- Prefer composition over inheritance, and think of code as layers of composition.
    - For instance, the MeshCatalog should not know anything about K8s. But it does need to call k8s clients. There may be a kubernetes client, that deals strictly with k8s objects, an OSM client, that converts from internal OSM calls, to the k8s client (this layer has OSM inputs, translates to k8s, and returns OSM outputs), and finally the mesh catalog, which may contain the business logic on the Internal types.
- Other interface best practices
- Other golang best practices

Happy Side Effects

While not explicit goals, we call out a few happy side effects of the proposed changes

- One big step towards letting OSM run outside of K8s.

    - By defining clear cut API's and removing existing leaky abstractions, we can allow for a single interface that users can define to integrate with a control plane and connect on-prem, or non-k8s clusters to their OSM mesh.
- Decoupled objects allow us to "think" in terms of OSM, while translation/builder objects reduce the burden of code when API's change.
- Increased performance
    - Due to the lack of builders, there's a lot of recomputation, particularly with nested for loops. By setting all of the inputs prior to building, we can optimize code to reduce this, improving both computational complexity, and intuitive (human) complexity.
- Improved Testing coverage
    - We will be able to remove substantial amounts of code. We will not consider this a success unless overall code coverage is increased.

Current Shortcomings & Proposed Solutions

The proposed summary of changes is as follows, with further details and motivation provided in the paragraphs below:

1. Methods in MeshCatalog and the xDS NewResponse's, will become **composers**, as defined above.
    a. This means they will not be responsible for building objects.
    b. They will instead, leverage builders, and provide all input up front to the builders. See the appendix below for an example
2. Builders will be used for both internal OSM objects, and envoy configs. They will never be passed any **client objects.**
3. OSM objects will become more normalized, as they are currently denormalized to more closely represent the envoy configs.
    a. Builders should save input as fields on their struct, and not perform any translation logic until `Build()` is called.
    b. However, as mentioned above, a package should not need to know how it is called. See the appendix below for an example of how this can be reduced.
    c. This will greatly reduce the amount of translation logic performed, as we currently have complex building in both MeshCatalog and xDS config packages.

Testing Plan

Maintain the same set of unit and e2e tests, by doing this piecemeal and incrementally.

Appendix

Further Motivation, and Examples of Inconsistency

- MeshCatalog acts as both a builder for OSM objects (traffic policy), and a client, with code switching between queries and constructing these structs.
    - For example, try tracing the calls from GetInboundMeshTrafficPolicy, and notice how we switch between constructing the objects, and issuing queries.
    - Some of these structs are also built outside of the MeshCatalog.
    - By leveraging builders we can clearly see what the inputs to a specific builder, such as an envoy config, are without having to drill too far down the “mental stack”
- The traffic policy structs are an attempt to decouple the K8s API from the Envoy API. While this is a great idea, the actual implementation is loosely coupled with the Envoy API, and suffers from the following:
    - These structs contain fields like `name` which are specific to envoy, and are denormalized in a way that more closely resembles the envoy configs, instead of the internal/intuitive OSM representation.
    - They don’t concretely map to the envoy API, so there is a lot of business logic to translate to these structs, and a lot of logic to translate from these to Envoy configs.
    - We can simplify this by keeping these constructs much more simple. See the appendix below for an example alternative.
    - Builders should have fewer code paths, as they can make smarter decisions on building since they have all the inputs at build time
- Similarly, the Envoy xDS response classes sometimes have 3 responsibilities: 1) building the OSM objects, 2) issuing queries, 3) Building the envoy configs. This should boil down to 1.
    - We will separate the xDS response classes from the builder logic. Each Envoy config will maintain a new builder struct, that is passed (data-only) input and knows how to generate the resulting Envoy configs, while the response handler is responsible for querying and feeding the resulting data to the builders.
- We do heavy lifting twice, by constructing denormalized data objects that are somewhat similar to envoy configs, but don’t map closely to our internal representation.
    - We will normalize the OSM internal objects, such that the envoy config builders are responsible for denormalization.
    - This will roughly halve the complexity across this set of code, while actually further decoupling these 2 types. See the appendix below for examples.

NewResponse Builders

See the current implementation for comparison

```
func NewResponse(meshCatalog catalog.MeshCataloger, proxy *envoy.Proxy, _
*xds_discovery.DiscoveryRequest, cfg configurator.Configurator, _
*certificate.Manager, proxyRegistry *registry.ProxyRegistry) ([]types.Resource, error)
{
   lb := newListenerBuilder(proxy.Identity)

   if featureflags := cfg.GetFeatureFlags(); featureflags.EnableWASMStats {
       lb.SetStatsHeaders(proxy.StatsHeaders())
   }

   svcList, err := proxyRegistry.ListProxyServices(proxy)
   if err != nil {
       return nil, err
   }
   lb.SetInboundServices(svcList)
   lb.SetEgressPolicy(meshCatalog.GetEgressTrafficPolicy())
   lb.SetOutboundEgressPolicy(meshCatalog.GetOutboundEgressTrafficPolicy())

   /*
       Set other inputs here
   */

   if pod, err := envoy.GetPodFromCertificate(proxy.GetCertificateCommonName(),
meshCatalog.GetKubeController()); err != nil {
       log.Warn().Str("proxy", proxy.String()).Msgf("Could not find pod for connecting
proxy, no metadata was recorded")
   } else {
       lb.SetMetricsEnabled(k8s.IsMetricsEnabled(pod))
   }

   return lb.Build()
}
```

Decoupled TrafficPolicy Objects

TrafficPolicy objects currently contain too much “knowledge” about the envoy config. They are a valiant attempt at making our objects easier to map to an envoy config, but ultimately end up

leaving envoy config logic in multiple places, and us performing similar mappings more than once.

Check out the code below for how we can simplify our objects.

Note that the below doesn't use a builder, so can be further improved, although it does pull up all of the querying into the top level method, leveraging functions below.

```
// GetInboundMeshTrafficPolicy returns the inbound mesh traffic policy for the given
upstream identity and services
func (mc *MeshCatalog) GetInboundRules(upstreamIdentity identity.ServiceIdentity)
[]*trafficpolicy.InboundRule {
   destinationFilter :=
smi.WithTrafficTargetDestination(upstreamIdentity.ToK8sServiceAccount())
   trafficTargets := mc.meshSpec.ListTrafficTargets(destinationFilter)

   // Create a map of maps to speed up lookups.
   routePolicies := make(map[string]map[string]trafficpolicy.HTTPRouteMatch)
   for _, trafficSpecs := range mc.meshSpec.ListHTTPTrafficSpecs() {
       specKey := getTrafficSpecName(smi.HTTPRouteGroupKind, trafficSpecs.Namespace,
trafficSpecs.Name)
       routePolicies[specKey] = make(map[string]trafficpolicy.HTTPRouteMatch)
       for _, trafficSpecsMatches := range trafficSpecs.Spec.Matches {
           serviceRoute := trafficpolicy.HTTPRouteMatch{
               Path:          trafficSpecsMatches.PathRegex,
               PathMatchType: trafficpolicy.PathMatchRegex,
               Methods:       trafficSpecsMatches.Methods,
               Headers:       trafficSpecsMatches.Headers,
           }

           // When pathRegex or/and methods are not defined, they will be wildcarded
           if serviceRoute.Path == "" {
               serviceRoute.Path = constants.RegexMatchAll
           }
           if len(serviceRoute.Methods) == 0 {
               serviceRoute.Methods = []string{constants.WildcardHTTPMethod}
           }
           routePolicies[specKey][trafficSpecsMatches.Name] = serviceRoute
       }
   }
```

```
   // From each TrafficTarget and HTTPRouteGroup configuration associated with this
service, build routes for it.
   var routingRules []*trafficpolicy.InboundRule
   for _, trafficTarget := range trafficTargets {
       rules := getRoutingRulesFromTrafficTarget(trafficTarget, routePolicies)
       // Multiple TrafficTarget objects can reference the same route, in which case
such routes
       // need to be merged to create a single route that includes all the downstream
client identities
       // this route is authorized for.
       routingRules = trafficpolicy.MergeRules(routingRules, rules)
   }

   return routingRules
}

func getRoutingRulesFromTrafficTarget(trafficTarget *access.TrafficTarget,
routePolicies map[string]map[string]trafficpolicy.HTTPRouteMatch)
[]*trafficpolicy.InboundRule {
   // Compute the allowed downstream service identities for the given TrafficTarget
object
   allowedDownstreamIdentities := mapset.NewSet()
   for _, source := range trafficTarget.Spec.Sources {

allowedDownstreamIdentities.Add(trafficTargetIdentityToSvcAccount(source).ToServiceIde
ntity())
   }

   var routingRules []*trafficpolicy.InboundRule
   for _, rule := range trafficTarget.Spec.Rules {
       trafficSpecName := getTrafficSpecName(smi.HTTPRouteGroupKind,
trafficTarget.Namespace, rule.Name)
       // TODO(steeling): I think there is a bug, where the SMI spec says if no
matches are specified, then all should be applied.
       for _, match := range rule.Matches {
           if matchedRoute, exists := routePolicies[trafficSpecName][match]; exists {
               rule := &trafficpolicy.InboundRule{
                   HTTPRouteMatch:           matchedRoute,
```

```
                    AllowedServiceIdentities: allowedDownstreamIdentities,
                }
                routingRules = append(routingRules, rule)
            } else {
                log.Debug().Msgf("No matching trafficpolicy.HTTPRoute found for match
name %s in Traffic Spec %s (in namespace %s)", match, trafficSpecName,
trafficTarget.Namespace)
            }
        }
    }

    return routingRules
}

func getTrafficSpecName(trafficSpecKind string, trafficSpecNamespace string,
trafficSpecName string) string {
    return fmt.Sprintf("%s/%s/%s", trafficSpecKind, trafficSpecNamespace,
trafficSpecName)
}
```

The current implementation is provided below for comparison:

```
// GetInboundMeshTrafficPolicy returns the inbound mesh traffic policy for the given
upstream identity and services
func (mc *MeshCatalog) GetInboundMeshTrafficPolicy(upstreamIdentity
identity.ServiceIdentity, upstreamServices []service.MeshService)
*trafficpolicy.InboundMeshTrafficPolicy {
    var trafficMatches []*trafficpolicy.TrafficMatch
    var clusterConfigs []*trafficpolicy.MeshClusterConfig
    var trafficTargets []*access.TrafficTarget
    routeConfigPerPort := make(map[int][]*trafficpolicy.InboundTrafficPolicy)

    permissiveMode := mc.configurator.IsPermissiveTrafficPolicyMode()
    if !permissiveMode {
        // Pre-computing the list of TrafficTarget optimizes to avoid repeated
```

```
        // cache lookups for each upstream service.
        destinationFilter :=
smi.WithTrafficTargetDestination(upstreamIdentity.ToK8sServiceAccount())
        trafficTargets = mc.meshSpec.ListTrafficTargets(destinationFilter)
    }

    // A policy (traffic match, route, cluster) must be built for each upstream
service. This
    // includes apex/root services associated with the given upstream service.
    allUpstreamServices := mc.getUpstreamServicesIncludeApex(upstreamServices)

    // Build configurations per upstream service
    for _, upstreamSvc := range allUpstreamServices {
        // ---
        // Create local cluster configs for this upstram service
        clusterConfigForSvc := &trafficpolicy.MeshClusterConfig{
            Name:    upstreamSvc.EnvoyLocalClusterName(),
            Service: upstreamSvc,
            Address: constants.LocalhostIPAddress,
            Port:    uint32(upstreamSvc.TargetPort),
        }
        clusterConfigs = append(clusterConfigs, clusterConfigForSvc)

        // ---
        // Create a TrafficMatch for this upstream servic.
        // The TrafficMatch will be used by LDS to program a filter chain match
        // for this upstream service on the upstream server to accept inbound
        // traffic.
        trafficMatchForUpstreamSvc := &trafficpolicy.TrafficMatch{
            Name:                upstreamSvc.InboundTrafficMatchName(),
            DestinationPort:     int(upstreamSvc.TargetPort),
            DestinationProtocol: upstreamSvc.Protocol,
        }
        trafficMatches = append(trafficMatches, trafficMatchForUpstreamSvc)

        // Build the HTTP route configs for this service and port combination.
        // If the port's protocol corresponds to TCP, we can skip this step
        if upstreamSvc.Protocol == constants.ProtocolTCP || upstreamSvc.Protocol ==
constants.ProtocolTCPServerFirst {
```

```
            continue
        }
        // ---
        // Build the HTTP route configs per port
        // Each upstream service accepts traffic from downstreams on a list of allowed
routes.
        // The routes are derived from SMI TrafficTarget and TrafficSplit policies in
SMI mode,
        // and are wildcarded in permissive mode. The downstreams that can access this
upstream
        // on the configured routes is also determined based on the traffic policy
mode.
        inboundTrafficPolicies := mc.getInboundTrafficPoliciesForUpstream(upstreamSvc,
permissiveMode, trafficTargets)
        routeConfigPerPort[int(upstreamSvc.TargetPort)] =
append(routeConfigPerPort[int(upstreamSvc.TargetPort)], inboundTrafficPolicies)
    }

    return &trafficpolicy.InboundMeshTrafficPolicy{
        TrafficMatches:          trafficMatches,
        ClustersConfigs:         clusterConfigs,
        HTTPRouteConfigsPerPort: routeConfigPerPort,
    }
}

func (mc *MeshCatalog) getInboundTrafficPoliciesForUpstream(upstreamSvc
service.MeshService, permissiveMode bool, trafficTargets []*access.TrafficTarget)
*trafficpolicy.InboundTrafficPolicy {
    var inboundPolicyForUpstreamSvc *trafficpolicy.InboundTrafficPolicy

    if permissiveMode {
        // Add a wildcard HTTP route that allows any downstream client to access the
upstream service
        hostnames := k8s.GetHostnamesForService(upstreamSvc, true /* local namespace
FQDN should always be allowed for inbound routes*/)
        inboundPolicyForUpstreamSvc =
trafficpolicy.NewInboundTrafficPolicy(upstreamSvc.FQDN(), hostnames)
        localCluster := service.WeightedCluster{
            ClusterName: service.ClusterName(upstreamSvc.EnvoyLocalClusterName()),
```

```
            Weight:      constants.ClusterWeightAcceptAll,
        }

inboundPolicyForUpstreamSvc.AddRule(*trafficpolicy.NewRouteWeightedCluster(trafficpoli
cy.WildCardRouteMatch, []service.WeightedCluster{localCluster}),
identity.WildcardServiceIdentity)
    } else {
        // Build the HTTP routes from SMI TrafficTarget and HTTPRouteGroup
configurations
        inboundPolicyForUpstreamSvc =
mc.buildInboundHTTPPolicyFromTrafficTarget(upstreamSvc, trafficTargets)
    }

    return inboundPolicyForUpstreamSvc
}

func (mc *MeshCatalog) buildInboundHTTPPolicyFromTrafficTarget(upstreamSvc
service.MeshService, trafficTargets []*access.TrafficTarget)
*trafficpolicy.InboundTrafficPolicy {
    hostnames := k8s.GetHostnamesForService(upstreamSvc, true /* local namespace FQDN
should always be allowed for inbound routes*/)
    inboundPolicy := trafficpolicy.NewInboundTrafficPolicy(upstreamSvc.FQDN(),
hostnames)
    localCluster := service.WeightedCluster{
        ClusterName: service.ClusterName(upstreamSvc.EnvoyLocalClusterName()),
        Weight:      constants.ClusterWeightAcceptAll,
    }

    var routingRules []*trafficpolicy.Rule
    // From each TrafficTarget and HTTPRouteGroup configuration associated with this
service, build routes for it.
    for _, trafficTarget := range trafficTargets {
        rules := mc.getRoutingRulesFromTrafficTarget(*trafficTarget, localCluster)
        // Multiple TrafficTarget objects can reference the same route, in which case
such routes
        // need to be merged to create a single route that includes all the downstream
client identities
        // this route is authorized for.
        routingRules = trafficpolicy.MergeRules(routingRules, rules)
```

```
    }
    inboundPolicy.Rules = routingRules

    return inboundPolicy
}

func (mc *MeshCatalog) getRoutingRulesFromTrafficTarget(trafficTarget
access.TrafficTarget, routingCluster service.WeightedCluster) []*trafficpolicy.Rule {
    // Compute the HTTP route matches associated with the given TrafficTarget object
    httpRouteMatches, err := mc.routesFromRules(trafficTarget.Spec.Rules,
trafficTarget.Namespace)
    if err != nil {
        log.Error().Err(err).Str(errcode.Kind,
errcode.GetErrCodeWithMetric(errcode.ErrFetchingSMIHTTPRouteGroupForTrafficTarget)).
            Msgf("Error finding route matches from TrafficTarget %s in namespace %s",
trafficTarget.Name, trafficTarget.Namespace)
        return nil
    }

    // Compute the allowed downstream service identities for the given TrafficTarget
object
    allowedDownstreamIdentities := mapset.NewSet()
    for _, source := range trafficTarget.Spec.Sources {
        sourceSvcIdentity :=
trafficTargetIdentityToSvcAccount(source).ToServiceIdentity()
        allowedDownstreamIdentities.Add(sourceSvcIdentity)
    }

    var routingRules []*trafficpolicy.Rule
    for _, httpRouteMatch := range httpRouteMatches {
        rule := &trafficpolicy.Rule{
            Route:
*trafficpolicy.NewRouteWeightedCluster(httpRouteMatch,
[]service.WeightedCluster{routingCluster}),
            AllowedServiceIdentities: allowedDownstreamIdentities,
        }
        routingRules = append(routingRules, rule)
    }
```

```
    return routingRules
}

// routesFromRules takes a set of traffic target rules and the namespace of the
traffic target and returns a list of
//  http route matches (trafficpolicy.HTTPRouteMatch)
func (mc *MeshCatalog) routesFromRules(rules []access.TrafficTargetRule,
trafficTargetNamespace string) ([]trafficpolicy.HTTPRouteMatch, error) {
    var routes []trafficpolicy.HTTPRouteMatch

    specMatchRoute, err := mc.getHTTPPathsPerRoute() // returns
map[traffic_spec_name]map[match_name]trafficpolicy.HTTPRoute
    if err != nil {
        return nil, err
    }

    if len(specMatchRoute) == 0 {
        log.Trace().Msg("No elements in map[traffic_spec_name]map[match
name]trafficpolicyHTTPRoute")
        return routes, nil
    }

    for _, rule := range rules {
        trafficSpecName := mc.getTrafficSpecName(smi.HTTPRouteGroupKind,
trafficTargetNamespace, rule.Name)
        for _, match := range rule.Matches {
            matchedRoute, found :=
specMatchRoute[trafficSpecName][trafficpolicy.TrafficSpecMatchName(match)]
            if found {
                routes = append(routes, matchedRoute)
            } else {
                log.Debug().Msgf("No matching trafficpolicy.HTTPRoute found for match
name %s in Traffic Spec %s (in namespace %s)", match, trafficSpecName,
trafficTargetNamespace)
            }
        }
    }

    return routes, nil
```

```
}

func (mc *MeshCatalog) getHTTPPathsPerRoute()
(map[trafficpolicy.TrafficSpecName]map[trafficpolicy.TrafficSpecMatchName]trafficpolic
y.HTTPRouteMatch, error) {
   routePolicies :=
make(map[trafficpolicy.TrafficSpecName]map[trafficpolicy.TrafficSpecMatchName]trafficp
olicy.HTTPRouteMatch)
   for _, trafficSpecs := range mc.meshSpec.ListHTTPTrafficSpecs() {
       log.Debug().Msgf("Discovered TrafficSpec resource: %s/%s",
trafficSpecs.Namespace, trafficSpecs.Name)
       if trafficSpecs.Spec.Matches == nil {
           log.Error().Str(errcode.Kind,
errcode.GetErrCodeWithMetric(errcode.ErrSMIHTTPRouteGroupNoMatch)).
               Msgf("TrafficSpec %s/%s has no matches in route; Skipping...",
trafficSpecs.Namespace, trafficSpecs.Name)
           continue
       }

       // since this method gets only specs related to HTTPRouteGroups added
HTTPTraffic to the specKey by default
       specKey := mc.getTrafficSpecName(smi.HTTPRouteGroupKind,
trafficSpecs.Namespace, trafficSpecs.Name)
       routePolicies[specKey] =
make(map[trafficpolicy.TrafficSpecMatchName]trafficpolicy.HTTPRouteMatch)
       for _, trafficSpecsMatches := range trafficSpecs.Spec.Matches {
           serviceRoute := trafficpolicy.HTTPRouteMatch{
               Path:          trafficSpecsMatches.PathRegex,
               PathMatchType: trafficpolicy.PathMatchRegex,
               Methods:       trafficSpecsMatches.Methods,
               Headers:       trafficSpecsMatches.Headers,
           }

           // When pathRegex or/and methods are not defined, they will be wildcarded
           if serviceRoute.Path == "" {
               serviceRoute.Path = constants.RegexMatchAll
           }
           if len(serviceRoute.Methods) == 0 {
               serviceRoute.Methods = []string{constants.WildcardHTTPMethod}
```

```
            }

routePolicies[specKey][trafficpolicy.TrafficSpecMatchName(trafficSpecsMatches.Name)] =
serviceRoute
        }
    }
    log.Debug().Msgf("Constructed HTTP path routes: %+v", routePolicies)
    return routePolicies, nil
}

func (mc *MeshCatalog) getTrafficSpecName(trafficSpecKind string, trafficSpecNamespace
string, trafficSpecName string) trafficpolicy.TrafficSpecName {
    specKey := fmt.Sprintf("%s/%s/%s", trafficSpecKind, trafficSpecNamespace,
trafficSpecName)
    return trafficpolicy.TrafficSpecName(specKey)
}

// getUpstreamServicesIncludeApex returns a list of all upstream services associated
with the given list
// of services. An upstream service is associated with another service if it is a
backend for an apex/root service
// in a TrafficSplit config. This function returns a list consisting of the given
upstream services and all apex
// services associated with each of those services.
func (mc *MeshCatalog) getUpstreamServicesIncludeApex(upstreamServices
[]service.MeshService) []service.MeshService {
    svcSet := mapset.NewSet()
    var allServices []service.MeshService

    // Each service could be a backend in a traffic split config. Construct a list
    // of all possible services the given list of services is associated with.
    for _, svc := range upstreamServices {
        if newlyAdded := svcSet.Add(svc); newlyAdded {
            allServices = append(allServices, svc)
        }

        for _, split := range
mc.meshSpec.ListTrafficSplits(smi.WithTrafficSplitBackendService(svc)) {
```

```
            svcName := k8s.GetServiceFromHostname(mc.kubeController,
split.Spec.Service)
            subdomain := k8s.GetSubdomainFromHostname(mc.kubeController,
split.Spec.Service)
            apexMeshService := service.MeshService{
                Namespace:  svc.Namespace,
                Name:       svcName,
                Port:       svc.Port,
                TargetPort: svc.TargetPort,
                Protocol:   svc.Protocol,
            }

            if subdomain != "" {
                apexMeshService.Name = fmt.Sprintf("%s.%s", subdomain, svcName)
            }

            if newlyAdded := svcSet.Add(apexMeshService); newlyAdded {
                allServices = append(allServices, apexMeshService)
            }
        }
    }

    return allServices
}
```