



## 1. Edge (API Gateway)

### **Purpose:**

Entry point for all tenant requests. Performs authentication, extracts tenant ID, and routes to the correct workflow engine.

### **Responsibilities:**

- Verify JWT / entitlements
- Identify tenant from URL or host
- Ask Discovery for backend (pooled or isolated)
- Proxy request to target Engine
- Collect per-tenant metrics

### **Interfaces:**

```
type Edge struct {
    discovery Discovery
    auth       AuthZ
    proxy      *httputil.ReverseProxy
}

func (e *Edge) Handle(req *http.Request) error
```

## 2. Discovery

### **Purpose:**

Dynamic routing map that tells the Edge where each tenant's engine lives.

### **Responsibilities:**

- Maintain {tenant → backend endpoint} map
- Stream updates from Tenant Manager or Registry
- Hot-reload routing map in Edge memory

### **Interfaces:**

```
type Discovery interface {
    Lookup(tenant string) (Backend, bool)
    Subscribe(ctx context.Context) (<-chan
TenantRouteUpdate, error)
}
```

## 3. Tenant Manager (Control Plane Brain)

### **Purpose:**

Reconciles desired vs. actual tenant state. Creates/destroys isolated runtimes and publishes discovery updates.

### **Responsibilities:**

- Ensure pooled/isolated environments exist
- Promote/demote tenants (based on metrics)
- Update Registry with status
- Publish Discovery updates

### **Interfaces:**

```
type TenantManager struct {
    registry    Registry
    provisioner Provisioner
    metrics     Metrics
    discovery   DiscoveryPublisher
}

func (m *TenantManager) Reconcile(ctx context.Context, t
TenantSpec) error
func (m *TenantManager) PolicyTick(ctx context.Context)
```

## 4. Registry (Tenant Store)

### **Purpose:**

Persistent source of truth for tenant specs, limits, and placement.

### Responsibilities:

- Store tenant spec & status
- Handle CRUD for tenants
- Used by both Manager and Discovery

### Interfaces:

```
type Registry interface {
    Get(ctx context.Context, tenant string) (TenantSpec,
TenantStatus, error)
    Upsert(ctx context.Context, spec TenantSpec) error
    List(ctx context.Context) ([]TenantSpec, error)
    UpdateStatus(ctx context.Context, tenant string, st
TenantStatus) error
}
```



## 5. Provisioner (Platform Adapter)

### Purpose:

Creates runtime units (Deployments, namespaces, containers, etc.) for isolated tenants.

### Responsibilities:

- Deploy engines/loaders
- Configure resources and limits
- Destroy isolated tenants when demoted

### Interfaces:

```
type Provisioner interface {
    EnsurePooled(ctx context.Context) (Backend, error)
    EnsureIsolated(ctx context.Context, t TenantSpec)
(Backend, error)
    DeleteIsolated(ctx context.Context, tenant string)
error
    Health(ctx context.Context, backend Backend) error
}
```



## 6. Engines (Shared & Isolated)

### Purpose:

Execute workflow graphs (AST nodes). Shared engine serves pooled tenants; isolated engines serve heavy ones.

### Responsibilities:

- Execute workflow nodes
- Apply per-tenant rate limits
- Expose `/metrics` for Prometheus
- Receive workflow updates from Loader

### Endpoints:

```
POST /v1/{tenant}/execute
GET /metrics
POST /internal/workflows/refresh
```



## 7. Loader

### Purpose:

Compiles and synchronizes workflow definitions and plugin configurations from DB to engines.

### Responsibilities:

- Periodically fetch workflows from shared DB
- Push updates to engines
- Validate plugin configs

### Interfaces:

```
type Loader interface {
    SyncWorkflows(ctx context.Context)
    PushUpdates(ctx context.Context, backend Backend)
}
```



## 8. Metrics / Prometheus

### Purpose:

Collects per-tenant usage, powers promotion/demotion logic.

### Responsibilities:

- Expose per-tenant CPU, memory, queue, latency
- Feed Alertmanager → Tenant Manager webhook
- Enable Grafana dashboards

### Interfaces:

```
type Metrics interface {
    TenantUsage(ctx context.Context, tenant string, window
    time.Duration)
    (cpuShare float64, memBytes uint64, queue int, err
    error)
}
```



## Flow Summary

1. Client sends `/v1/{tenant}/execute` → Edge → Discovery lookup → correct Engine
2. Engine runs workflow → exports metrics
3. Prometheus collects metrics → triggers Alert → Tenant Manager webhook
4. Tenant Manager calls Provisioner → deploys isolated engine → updates Registry
5. Discovery publishes routing update → Edge hot-reloads mapping instantly

Would you like me to generate a **labeled, colored version** of this diagram (with control plane and data plane boundaries marked)? It's perfect for presentation slides.

You said:

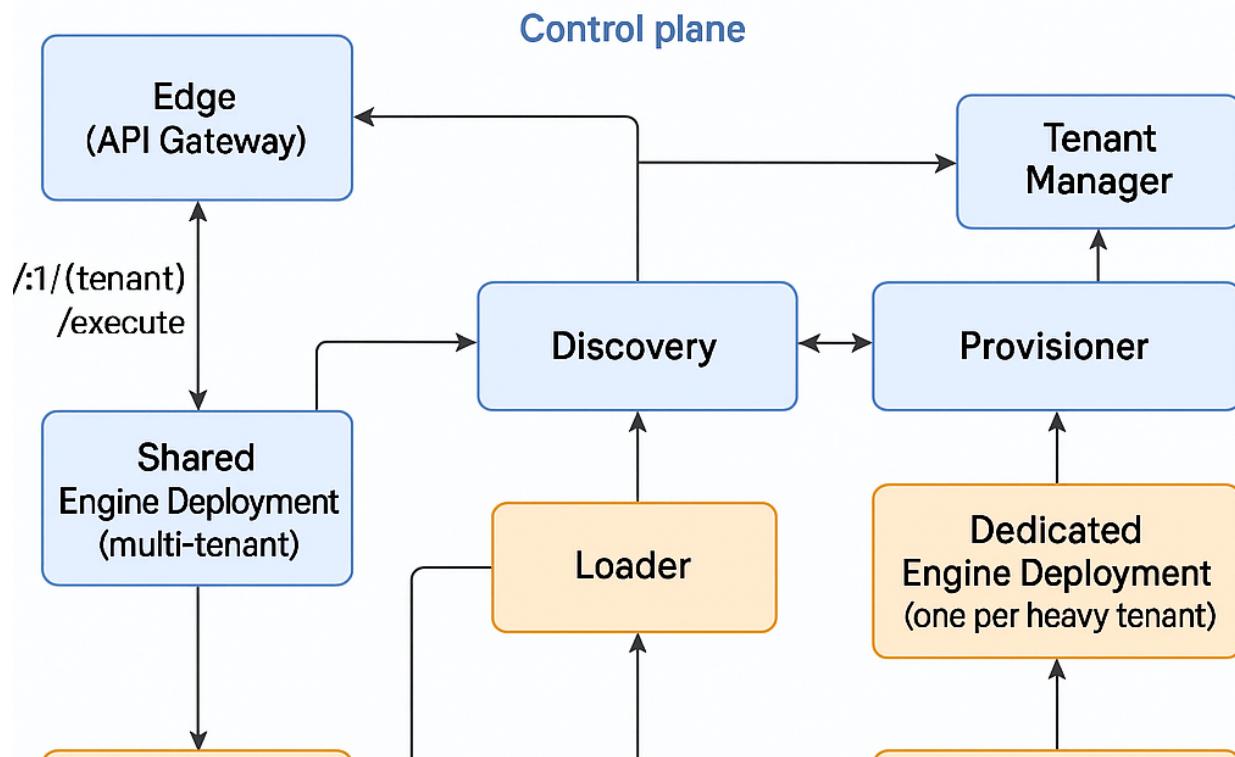
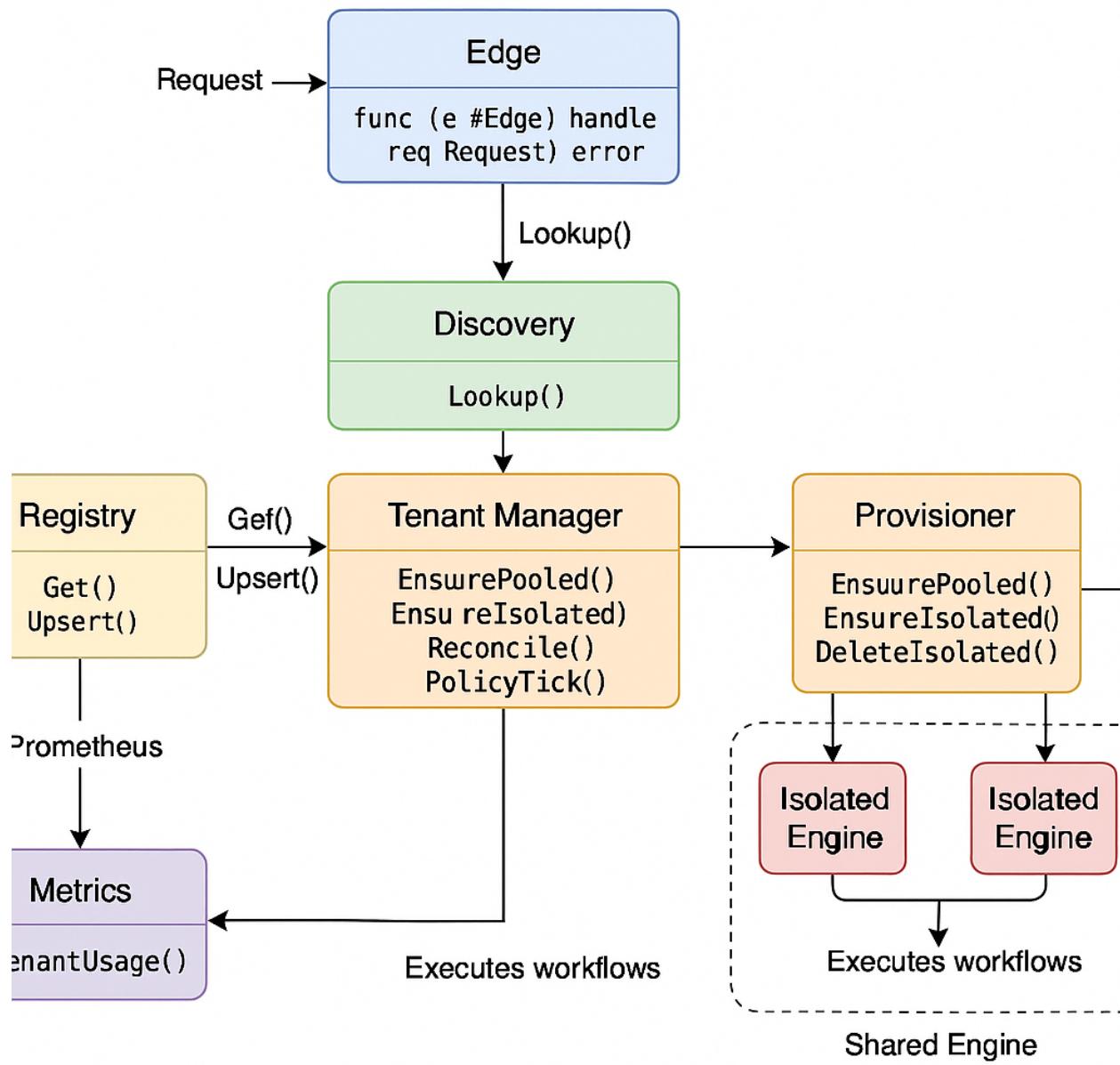


diagram is not full, I can see just part of it

ChatGPT said:

Image created



Share

✓ Here's your **final full architecture diagram** showing all components and control/data flows:



A\_diagram\_illustrates\_a\_multi-tenant\_architecture\_.png

It includes the full control plane (Edge, Discovery, Tenant Manager, Provisioner) and data plane (Engines, Loader, DB) with routing, provisioning, and metrics flows.

You said:

okay and give me description and interfaces for all those components now, and describe interaction contract for them describing specifically which part is the infrastructure agnostic and which needs to have integration with some OC api(which one)

ChatGPT said:

Thought for 15s

Perfect—here's a clean, implementation-oriented spec of **all components**, with **interfaces**, **interaction contracts**, and a clear split between **infrastructure-agnostic** parts vs. **where OpenShift (OC/K8s) APIs are required**.

## Components, Interfaces, and Contracts

### 1) Edge (API Gateway) – *infra-agnostic*

**Role:** AuthN/Z, extract `{tenant}`, look up backend, reverse-proxy to the right Engine, emit metrics.

**Interface (Go):**

```

type Backend struct {
    Scheme string // "http"|"https"
    Host   string // "engine-
pooled.workflows.svc.cluster.local:8080"
    Timeouts struct{ Dial, Read, Write time.Duration }
}

type TenantRouteUpdate struct {
    Tenant string
    Backend Backend
    Version int64 // monotonic; ignore stale
}

type Discovery interface {
    Lookup(tenant string) (Backend, bool) // O(1) read
    Subscribe(ctx context.Context) (<-chan TenantRouteUpdate, error) // push updates
}

type AuthZ interface {
    VerifyJWT(r *http.Request) (claims map[string]any, err error)
    CheckEntitlements(claims map[string]any, tenant string) error
}

type Edge struct {
    Disc Discovery
    Auth AuthZ
    RP   *httputil.ReverseProxy // one proxy; backend chosen per request
}

Contract:

```

- **Input:** HTTP 1.1/2 request POST /v1/{tenant}/... with JWT.
- **Behavior:** Validate JWT → Disc.Lookup(tenant) → proxy to Backend.Host.

- **Errors:** 401/403 on auth failure; 503 if no backend or backend unhealthy.
- **SLOs:** Stateless, horizontally scalable; hot-reload routes from **Subscribe**.

**Infra:** None required (works on VMs, K8s, OpenShift, etc.).

## 2) Discovery — *two implementations*

**Role:** Provide Edge with an always-fresh `{tenant → backend}` map.

**Agnostic (recommended first): Registry-backed Discovery**

- Edge **subscribes** to a gRPC/SSE/WebSocket stream exposed by Manager/Registry.

**K8s/OC aware (optional): CRD watcher**

- Edge runs a read-only informer on a **Tenant** CRD and builds the map from `status.endpoint`.

**Interface (Edge consumes):** see above.

**Contracts:**

- **Transport:** gRPC stream or SSE/WebSocket; JSON payload:

```
{ "tenant": "acme", "backend":  
  { "scheme": "http", "host": "engine-acme.t-acme.svc:8080" },  
  "version": 1730660000000 }
```

- 

- **Semantics:** At-least-once delivery; idempotent via **Version**.

**Infra:**

- Agnostic mode: none.
- CRD watcher mode: needs K8s list/watch on the **Tenant** CRD (custom API group).

### 3) Registry (Tenant Store) — *infra-agnostic*

**Role:** Persist **spec** (desired placement/limits) and **status** (actual endpoint, readiness).

**Interface (Go):**

```
type TenantSpec struct {
    Name      string      // "acme"
    SKU       string      // "POOLED" | "ISOLATED"
    CPULimit  string      // "500m"
    MemLimit  string      // "512Mi"
    Labels    map[string]string // optional
}

type TenantStatus struct {
    Placement  string // "POOLED" | "ISOLATED"
    Namespace  string // ns for isolated
    Endpoint   Backend // resolved service endpoint
    Ready      bool
    LastChange time.Time
}

type Registry interface {
    Get(ctx context.Context, name string) (TenantSpec, TenantStatus, error)
    Upsert(ctx context.Context, spec TenantSpec) error
    UpdateStatus(ctx context.Context, name string, st TenantStatus) error
    List(ctx context.Context) ([]TenantSpec, error)
}
```

**Storage:** Postgres (portable).

**API (optional):** REST/gRPC for admin/Edge read.

**Infra:** None (DB only).

### 4) Tenant Manager (Control Plane) — *core logic; pluggable adapters*

**Role:** Reconcile desired → actual, run promotion/demotion policy, publish Discovery updates.

## Interfaces (Go):

```
type Metrics interface {
    TenantUsage(ctx context.Context, tenant string, window
time.Duration) (cpuShare float64, memBytes uint64, queue
int, err error)
}

type DiscoveryPublisher interface {
    Publish(ctx context.Context, upd TenantRouteUpdate) error
    // fan-out to all Edges
}

type Provisioner interface {
    EnsurePooled(ctx context.Context) (Backend, error)
    EnsureIsolated(ctx context.Context, spec TenantSpec)
    (Backend, error)
    DeleteIsolated(ctx context.Context, tenant string) error
    Health(ctx context.Context, b Backend) error
}

type Manager struct {
    Reg    Registry
    Prov   Provisioner
    Met    Metrics
    Pub    DiscoveryPublisher
    Now    func() time.Time
}

func (m *Manager) Reconcile(ctx context.Context, name
string) error // idempotent
func (m *Manager) PolicyTick(ctx context.Context)
// periodic/push
Contracts:
```

- **Promotion rule (example):** `cpuShare > 0.70` for 5m → SKU=ISOLATED.
- **Demotion rule:** `cpuShare < 0.20` for 60m + cooldown.
- **Reconcile:** Must be **idempotent** (safe to retry); publish new Backend on success.

- **Security:** Write access to Registry; publish rights to Discovery; platform rights via **Provisioner**.

**Infra:** Manager itself is agnostic; **Provisioner** decides OC/K8s integration (see below).

## 5) Provisioner (Platform Adapter) — *OC/K8s integration lives here*

**Role:** Create/update/delete runtime for pooled/isolated tenants.

**Interface:** see above.

**OpenShift/K8s implementation calls:**

- **Pooled** (ensure shared engine exists):
  - `apps/v1.Deployments` (engine-pooled)
  - `v1.Services` (clusterIP)
  - Optional `autoscaling/v2.HorizontalPodAutoscaler`
  - `v1.ResourceQuota + v1.LimitRange` at namespace `workflows` (once)
- **Isolated** (per tenant):
  - `v1.Namespaces` → `t-<tenant>`
  - `v1.ResourceQuota, v1.LimitRange` (ns caps)
  - `apps/v1.Deployments` (engine-tenant, loader CronJob: `batch/v1.CronJob`)
  - `v1.Services` (engine-tenant)
  - **OpenShift Route** `route.openshift.io/v1.Route` (*only if you want direct external ingress; not required if all goes through Edge*)
  - `networking.k8s.io/v1.NetworkPolicy` (allow DB + observability only)
  - `v1.ConfigMap, v1.Secret` (if you keep some config outside DB)

- Optional `rbac.authorization.k8s.io/v1` SA/Role/RoleBinding (scoped)

**Health check:** `HTTP GET /healthz` on engine service; also `Endpoints` readiness.

**Agnostic implementations:**

- `ecsProvisioner` (ECS service/task), `vmProvisioner` (systemd+containerd) with cgroups.

## 6) Engines (Shared & Isolated) — *infra-agnostic*

**Role:** Execute AST workflows, enforce soft quotas, expose metrics, accept live updates.

**Process contracts:**

- **Execute:** `POST /v1/{tenant}/execute` (JSON; idempotency key recommended)
- **Refresh:** `POST /internal/workflows/refresh` (signed internal call)
- **Metrics:** `GET /metrics` (Prometheus; add OTel exporter for portability)

**Per-tenant soft limits (in pooled mode):**

- Concurrency semaphore + token-bucket rate limiter
- Bounded per-tenant caches (LRU)
- Return `429` with `Retry-After` when throttled

**Security:** mTLS or network-internal; JWT from Edge is trusted claim for `{tenant}`.

## 7) Loader — *infra-agnostic*

**Role:** Compile/validate workflows & plugin configs from DB, push diffs to Engines.

**Contracts:**

- **Pull:** SQL (Postgres) by `tenant_id, version`
- **Push:** `POST /internal/workflows/refresh` to pooled and/or isolated engine

- **Schedule:** Cron (K8s batch/v1.CronJob) or internal ticker (VMs)

## 8) Metrics & Alerts — *infra-agnostic (Prometheus/OTel)*

**Role:** Expose app metrics; Alertmanager notifies Manager.

**Contracts:**

- **App metrics (examples):**
  - tenant\_cpu\_seconds\_total{tenant} (counter / proxy)
  - tenant\_workflows\_active{tenant} (gauge)
  - tenant\_queue\_depth{tenant} (gauge)
  - tenant\_rate\_limited\_total{tenant} (counter)
- **Alert → Manager:** POST /alerts (JSON from Alertmanager)

**Example alert (illustrative):**

```
- alert: TenantOveruse
  expr: sum(rate(tenant_cpu_seconds_total[5m])) by (tenant) >
        0.70 *
  sum(kube_pod_container_resource_limits_cpu_cores{namespace="workflows"})
  for: 5m
  labels: {severity: "warn"}
  annotations: {summary: "Tenant {{ $labels.tenant }} exceeds pooled CPU"}
```

## Interaction Matrix (who talks to whom)

From → To	Transport	Payload	Notes
<b>Client → Edge</b>	HTTP(S)	API calls + JWT	Auth at Edge
<b>Edge → Discovery</b>	in-proc API	<code>Lookup()</code> + <code>Subscribe()</code>	No network if Discovery client is embedded
<b>Edge → Engine</b>	HTTP(S)	Proxied request	Adds tenant headers
<b>Loader → Engine</b>	HTTP(S)	<code>refresh</code> payload	Signed internal call
<b>Engine → Metrics</b>	HTTP/OTLP	<code>/metrics</code> & traces	Scraped/exported
<b>Prom/AM → Manager</b>	HTTP(S)	Alert JSON	Webhook
<b>Manager → Provisioner</b>	Go iface → OC API	K8s objects	Idempotent
<b>Manager → Registry</b>	gRPC/SQL	Spec/Status	Single source of truth
<b>Manager → DiscoveryPublisher</b>	gRPC/SSE	<code>TenantRouteUpdate</code>	Fan-out; versioned

## Infra-Agnostic vs. OC/K8s-Specific

### Infra-agnostic (portable anywhere):

- Edge (router)
- Discovery (registry-stream variant)
- Registry (Postgres)
- Engines (shared/isolated binaries)
- Loader (cron or ticker)
- Metrics/Tracing (OpenTelemetry + optional Prom scrape)
- Manager core logic (policy, reconcile orchestration)

### OpenShift/K8s integration (lives ONLY in **Provisioner** and optional **Discovery**/CRD):

- `core/v1`: Namespace, Service, ConfigMap, Secret, ResourceQuota, LimitRange
- `apps/v1`: Deployment
- `batch/v1`: CronJob
- `networking.k8s.io/v1`: NetworkPolicy

- `rbac.authorization.k8s.io/v1`: SA/Role/RoleBinding
- `autoscaling/v2`: HPA (optional)
- `route.openshift.io/v1`: Route (only if you want direct external ingress to isolated engines)
- **(Optional)** Discovery via K8s watch on a Tenant CRD (`apiextensions.k8s.io/v1`)

All OC/K8s specifics are isolated behind **Provisioner** (and optional CRD-based Discovery). Swap the adapter to run on ECS/VMs without touching Edge/Engine/Loader/Manager core.

## Minimal DTOs (shared across services)

```

type TenantSpec    {...} // as above
type TenantStatus {...} // as above
type Backend      {...} // as above

// Admin/Onboarding API (Registry or Manager HTTP)
POST /tenants
{
  "name": "acme",
  "sku": "POOLED",
  "cpuLimit": "500m",
  "memLimit": "512Mi"
}

PATCH /tenants/acme
{
  "sku": "ISOLATED"
}

GET /tenants/acme
{
  "spec": {...},
  "status": {...}
}

```

## Operational Guarantees

- **Idempotency:** `Reconcile()` and all `Ensure*()` operations safe to retry.
- **Eventual consistency:** Edge sees new endpoints after `DiscoveryPublisher.Publish()`; keep TTLs short and use versions.
- **Security:** mTLS internal; Edge has read-only Discovery; Manager SA has minimal RBAC; Engines trust Edge JWT.

