# Implementing RBAC in a Node.js Express API Gateway

In a microservices setup, the **API Gateway** (Express.js) acts as the single entry point for all clients[[1]](https://www.aegissofttech.com/insights/securing-microservices-in-node-js/#:~:text=API%20Gateway%20is%20the%20single,that%20come%20out%20of%20them). It handles JWT authentication and routes requests to downstream services (CMS, HR, etc.) or the RBAC service. You should first **design your RBAC data model**: treat each *Company* as a tenant, with its own Departments, Functions, Modules, Resources and Permissions. Define **Roles** (e.g. “Sales-Viewer”, “Sales-Editor”) that bundle specific resource-level actions (create, edit, view, delete). Each **User** belongs to one company and is assigned one or more roles (possibly with fine-grained boolean flags for each permission). For example, in a multi-tenant system “each company has its own users… users have specific roles (Admin, Agent, Customer) within each tenant”[[2]](https://www.permit.io/blog/implement-multi-tenancy-rbac-in-mongodb#:~:text=,its%20own%20users%20and%20tickets). Use a central RBAC microservice (Node+Express with a database) to store these relationships and answer permission queries[[3]](https://www.permit.io/blog/implement-multi-tenancy-rbac-in-mongodb#:~:text=Instead%20of%20embedding%20access%20rules,approach%20ensures%20three%20important%20things).

## Authentication with JWT

Use **JSON Web Tokens (JWT)** for authentication. On login (via your Auth/RBAC service), verify the user’s credentials and then generate a JWT that includes the user’s identity and context[[4]](https://api7.ai/learning-center/api-101/implementing-jwt-for-api-security#:~:text=%2F%2F%20Example%20payload%20in%20Node,user%27%2C%20%27editor%27%5D). For example:

* **Payload claims:** include sub (user ID), companyId (tenant), and any static role/permission data (e.g. an array of roles or flags). For instance:
* const payload = {  
   sub: userId,  
   company: companyId,  
   roles: ['sales\_viewer'], // user’s roles or permission flags  
   exp: Math.floor(Date.now()/1000) + 3600 // 1-hour expiry  
  };  
  const token = jwt.sign(payload, JWT\_SECRET, { algorithm: 'HS256' });
* (This example uses the [jsonwebtoken](https://www.npmjs.com/package/jsonwebtoken) library.)[[4]](https://api7.ai/learning-center/api-101/implementing-jwt-for-api-security#:~:text=%2F%2F%20Example%20payload%20in%20Node,user%27%2C%20%27editor%27%5D). JWTs are popular for this purpose because **user info is self-contained and signed**[[5]](https://www.aegissofttech.com/insights/securing-microservices-in-node-js/#:~:text=,between%20services%2C%20and%20verification%20securely).
* **Secure tokens:** Use a strong signing key (HS256/RS256) and store secrets securely (env vars or a vault)[[6]](https://api7.ai/learning-center/api-101/implementing-jwt-for-api-security#:~:text=,Rotate%20keys%20periodically). Always set an expiration (exp) and use HTTPS so tokens aren’t exposed in transit[[7]](https://api7.ai/learning-center/api-101/implementing-jwt-for-api-security#:~:text=,that%20could%20intercept%20the%20token)[[8]](https://api7.ai/learning-center/api-101/implementing-jwt-for-api-security#:~:text=meaning%20the%20payload%20is%20Base64Url,Refresh%20tokens%20should%20be). For extra security, implement refresh tokens or short TTLs to limit misuse[[8]](https://api7.ai/learning-center/api-101/implementing-jwt-for-api-security#:~:text=meaning%20the%20payload%20is%20Base64Url,Refresh%20tokens%20should%20be).
* **Gateway middleware:** In your Express API Gateway, add middleware to **verify** the JWT on every incoming request[[9]](https://api7.ai/learning-center/api-101/implementing-jwt-for-api-security#:~:text=,any%20other%20necessary%20claim%20validations). Extract the Authorization: Bearer <token> header, then call jwt.verify(token, JWT\_SECRET, options) to check the signature and claims (e.g. exp, iss, aud). If verification fails, return 401. On success, attach the decoded payload (user ID, roles, etc.) to req.user for downstream use[[10]](https://api7.ai/learning-center/api-101/implementing-jwt-for-api-security#:~:text=%2F%2F%20Example%20validation%20in%20Node,catch%20%28err%29). Example in pseudocode:
* function authenticateJWT(req, res, next) {  
   const token = req.header('Authorization')?.split(' ')[1];  
   if (!token) return res.status(401).send('Unauthorized');  
   try {  
   const decoded = jwt.verify(token, JWT\_SECRET, { algorithms: ['HS256'], audience: API\_AUDIENCE });  
   req.user = decoded;  
   next();  
   } catch {  
   return res.status(401).send('Invalid token');  
   }  
  }
* This ensures **authentication** is done once at the gateway.

## Authorization: Push vs Pull

There are two common patterns for RBAC in microservices[[11]](https://www.osohq.com/post/microservices-authorization-patterns#:~:text=One%20clean%20solution%20to%20the,authorization%20decisions%20based%20on%20that)[[12]](https://www.osohq.com/post/microservices-authorization-patterns#:~:text=Another%20solution%20is%20to%20put,and%20ask%20the%20authorization%20service):

* **Push-based (Gateway-attached roles):** After authenticating the JWT, the gateway **retrieves or reads the user’s roles/permissions** and attaches them to each proxied request. For example, the gateway can add headers like X-User-Id and X-User-Roles, or simply trust the roles already in req.user. Downstream services (CMS, HR, etc.) then use these roles to check access. This is often implemented by middleware in the gateway: e.g.
* function authorize(requiredRole) {  
   return (req, res, next) => {  
   if (!req.user.roles || !req.user.roles.includes(requiredRole)) {  
   return res.status(403).send('Forbidden');  
   }  
   next();  
   };  
  }
* As Oso notes, this “gateway” pattern is simple: the gateway *attaches user info and role information* before passing the request along[[13]](https://www.osohq.com/post/microservices-authorization-patterns#:~:text=It%20is%20common%20for%20the,service%20in%20the%20example%20above). The service can then perform an immediate check without extra calls. It works best when each user has a **small number of roles**[[14]](https://www.osohq.com/post/microservices-authorization-patterns#:~:text=Note%20that%20using%20bare%20headers,often%20expressed%20as%20a%20JWT). Its upside is low latency (no extra network hop) and centralized enforcement at the gateway[[15]](https://www.osohq.com/post/microservices-authorization-patterns#:~:text=The%20main%20benefit%20of%20the,trips). The downside is potential staleness: if you change a user’s roles, you must reissue their token (or the new roles won’t appear in JWT claims). Also, **don’t trust clients to send their own roles** – only attach roles yourself after verifying the JWT.
* **Pull-based (Central RBAC service):** Alternatively, each service (or the gateway) can **query the RBAC microservice on-demand** to make an authorization decision. For example, when a user tries to create an invoice, the gateway could do something like POST /rbac/check with {userId, companyId, resource:'invoice', action:'create'} and wait for a true/false response. The Oso guide calls this a centralized authorization service: downstream simply asks “*can user X do Y?*”[[16]](https://www.osohq.com/post/microservices-authorization-patterns#:~:text=In%20this%20model%2C%20the%20documents,to%20make%20that%20decision). This ensures you always use the latest permission data, and you can implement very fine-grained or dynamic policies. Its drawbacks are added latency and operational complexity: the RBAC service must be highly available and may need to replicate user and resource data from other services[[17]](https://www.osohq.com/post/microservices-authorization-patterns#:~:text=Of%20course%2C%20this%20separation%20of,become%20a%20bottleneck%20for%20new)[[18]](https://www.osohq.com/post/microservices-authorization-patterns#:~:text=There%20are%20other%20factors%20that,queries%2C%20every%20request%20is%20slow).

Which approach is best? It depends on complexity[[19]](https://www.osohq.com/post/microservices-authorization-patterns#:~:text=When%20speaking%20with%20engineering%20teams%2C,into%20a%20dedicated%20authorization%20service). If each user has only a few static roles that easily fit in a token, **push (gateway pattern)** is efficient[[14]](https://www.osohq.com/post/microservices-authorization-patterns#:~:text=Note%20that%20using%20bare%20headers,often%20expressed%20as%20a%20JWT)[[19]](https://www.osohq.com/post/microservices-authorization-patterns#:~:text=When%20speaking%20with%20engineering%20teams%2C,into%20a%20dedicated%20authorization%20service). If you have many dynamic permissions or very fine-grained checks, **pull (centralized service)** ensures accuracy[[20]](https://www.osohq.com/post/microservices-authorization-patterns#:~:text=This%20can%20be%20quite%20appealing%3A,rest%20of%20your%20overall%20system)[[17]](https://www.osohq.com/post/microservices-authorization-patterns#:~:text=Of%20course%2C%20this%20separation%20of,become%20a%20bottleneck%20for%20new). A hybrid is common: embed core roles in the JWT for quick checks and call the RBAC service for sensitive actions. For example, the gateway can attach req.user = decodedJWT with roles, and then in an authorization middleware either check req.user.roles or call RBACService.checkPermission(...) as needed. In your scenario (centralized RBAC service, Node/Express stack), a **pull-based check at the gateway** is natural: upon each request, after verifying the JWT, call your RBAC service to verify that the user’s role allows the requested operation. Alternatively, you can push the roles in the JWT at login time if updates are infrequent.

## Step-by-Step Implementation

1. **Design the RBAC service and data schema.** Define database tables/collections for Company, Department, Function, Module, Resource, Permission, Role, and User (with User–Role mappings). For each Role, store which permissions (actions on resources) it grants within a company. For each User, store assigned Role(s) (and possibly individual permission flags). This follows a typical multi-tenant RBAC model, where “tenant-specific roles are managed dynamically” and logic is separated from app DB[[3]](https://www.permit.io/blog/implement-multi-tenancy-rbac-in-mongodb#:~:text=Instead%20of%20embedding%20access%20rules,approach%20ensures%20three%20important%20things).
2. **Build the RBAC microservice.** Use Node.js/Express (since you use Express already) and connect to your chosen database. Expose endpoints to create roles, assign permissions, and crucially to *check* permissions. For example, an endpoint POST /rbac/check that takes { userId, companyId, resource, action } and returns { allowed: true/false } after querying the database. Ensure this service itself is protected (e.g. it trusts only requests from the gateway, perhaps via an internal API key or mutual TLS).
3. **Implement user login and JWT issuance.** In your Auth or RBAC service, create a login endpoint. When a user logs in, verify credentials and then query the RBAC database for that user’s roles (or permissions). Create a JWT with the user’s sub (ID), companyId, and their role or permission claims. For example, you might include a roles: ['sales\_viewer'] array in the token[[4]](https://api7.ai/learning-center/api-101/implementing-jwt-for-api-security#:~:text=%2F%2F%20Example%20payload%20in%20Node,user%27%2C%20%27editor%27%5D). Sign it with your secret key. Return this token to the client. This token now serves as proof of identity/role until it expires.
4. **Set up the Express API Gateway.** In your Node.js gateway, import middleware to verify JWT on each request. For example, use express-jwt or custom middleware as shown above. Configure it with your JWT secret/public key and validation parameters (issuer, audience). Attach the decoded token to req.user. Then route requests to your services (e.g. app.use('/cms', authenticateJWT, cmsRouter)). The gateway can either proxy (using HTTP client or http-proxy) or mount routers that forward requests to downstream URLs. Because you share the same JWT secret across services, you could also allow services to independently verify tokens.
5. **Enforce authorization (RBAC) at the gateway.** For each protected route, apply an authorization middleware. If you chose **push-based**, this middleware will check the roles/permissions in req.user. For example:

* function requireRole(role) {  
   return (req, res, next) => {  
   if (!req.user.roles || !req.user.roles.includes(role)) {  
   return res.status(403).send('Forbidden');  
   }  
   next();  
   };  
  }  
  // Usage:  
  app.get('/cms/invoices', authenticateJWT, requireRole('sales\_viewer'), proxyToCMS);
* If you chose **pull-based**, the middleware will call your RBAC service instead: e.g.
* async function requirePermission(resource, action) {  
   return async (req, res, next) => {  
   const { sub: userId, company } = req.user;  
   const { data } = await axios.post(`${RBAC\_URL}/check`,   
   { userId, companyId: company, resource, action });  
   if (data.allowed) next();  
   else res.status(403).send('Forbidden');  
   };  
  }  
  // Usage:  
  app.post('/cms/invoices', authenticateJWT, requirePermission('invoice', 'create'), proxyToCMS);
* This middleware approach pushes the Policy Enforcement Point into the gateway as recommended[[21]](https://www.alexanderlolis.com/authorization-in-a-microservices-world#:~:text=Authorization%20flow%20overview)[[22]](https://stackoverflow.com/questions/59786338/how-to-implement-role-based-security-in-microservices-architecture#:~:text=,as%20directly%20to%20your%20microservices). (As one StackOverflow answer noted, “You don’t need to validate the JWT twice… Then [the] API gateway propagates the roles. In your microservice, you initialise… using the roles passed in the header”[[22]](https://stackoverflow.com/questions/59786338/how-to-implement-role-based-security-in-microservices-architecture#:~:text=,as%20directly%20to%20your%20microservices).)

1. **Integrate downstream services.** Configure your CMS, HR, etc. to trust requests from the gateway. You can either let the gateway handle all authorization and have services skip checks, or have each service also verify the JWT and (if push-based) check req.user.roles. In practice it’s safest if each service also validates the JWT signature (so they trust the gateway) and then uses its own logic or re-queries RBAC. In any case, the gateway serves as the common entry point, so clients never call services directly.
2. **Secure the system.** Use HTTPS/TLS for all client-to-gateway and inter-service communication (to protect JWTs and data in transit)[[7]](https://api7.ai/learning-center/api-101/implementing-jwt-for-api-security#:~:text=,that%20could%20intercept%20the%20token). Store secrets (JWT keys, API keys) in environment variables or a secure vault[[6]](https://api7.ai/learning-center/api-101/implementing-jwt-for-api-security#:~:text=,Rotate%20keys%20periodically). Keep JWT lifetimes short and use refresh tokens if needed[[8]](https://api7.ai/learning-center/api-101/implementing-jwt-for-api-security#:~:text=meaning%20the%20payload%20is%20Base64Url,Refresh%20tokens%20should%20be). Apply rate limiting and throttling on the gateway to mitigate abuse[[1]](https://www.aegissofttech.com/insights/securing-microservices-in-node-js/#:~:text=API%20Gateway%20is%20the%20single,that%20come%20out%20of%20them). Log all auth decisions for auditing. Finally, enforce the principle of least privilege: give each role (and each microservice) only the permissions it needs[[23]](https://www.aegissofttech.com/insights/securing-microservices-in-node-js/#:~:text=Microservices%20need%20to%20be%20kept,controlling%20and%20monitoring%20access%20rights).

## Best Practices

* **Centralize security at the gateway.** By validating JWTs in one place, you reduce attack surface[[1]](https://www.aegissofttech.com/insights/securing-microservices-in-node-js/#:~:text=API%20Gateway%20is%20the%20single,that%20come%20out%20of%20them). You can centrally manage CORS, throttling, and API keys at the gateway.
* **Keep payloads lean.** Only put non-sensitive, essential claims in JWTs (user ID, roles, tenant)[[8]](https://api7.ai/learning-center/api-101/implementing-jwt-for-api-security#:~:text=meaning%20the%20payload%20is%20Base64Url,Refresh%20tokens%20should%20be). Avoid packing large permission lists into tokens.
* **Use strong cryptography.** Sign tokens with robust algorithms (HS256 with a long secret, or RS256 with rotated keys)[[6]](https://api7.ai/learning-center/api-101/implementing-jwt-for-api-security#:~:text=,Rotate%20keys%20periodically). Never allow alg: none.
* **Verify all token claims.** On each request, check exp and optionally iss/aud[[9]](https://api7.ai/learning-center/api-101/implementing-jwt-for-api-security#:~:text=,any%20other%20necessary%20claim%20validations). If a token is expired or malformed, reject it.
* **Enforce least privilege.** Design roles and permissions narrowly so that each user or service only accesses needed resources[[23]](https://www.aegissofttech.com/insights/securing-microservices-in-node-js/#:~:text=Microservices%20need%20to%20be%20kept,controlling%20and%20monitoring%20access%20rights). For example, a “Sales Viewer” role should have only read/view permissions on sales modules.
* **Use HTTPS everywhere.** Transmit JWTs and API calls over TLS to prevent interception[[7]](https://api7.ai/learning-center/api-101/implementing-jwt-for-api-security#:~:text=,that%20could%20intercept%20the%20token).
* **Plan for revocation.** Since JWTs are stateless, use short expirations and, if needed, a token blacklist for urgent revocations.
* **Document and test policies.** Clearly document which roles map to which permissions, and write automated tests to verify that your middleware blocks unauthorized actions.

By following these steps and practices, you ensure that your Node.js API Gateway authenticates users via JWT and enforces fine-grained RBAC either by carrying role claims (push pattern) or consulting the centralized RBAC service (pull pattern). The choice depends on your needs: for simple, low-latency checks, embedding roles in the JWT at login is convenient[[14]](https://www.osohq.com/post/microservices-authorization-patterns#:~:text=Note%20that%20using%20bare%20headers,often%20expressed%20as%20a%20JWT); for dynamic or very granular policies, querying the RBAC service on each request is safer[[20]](https://www.osohq.com/post/microservices-authorization-patterns#:~:text=This%20can%20be%20quite%20appealing%3A,rest%20of%20your%20overall%20system). In all cases, proper middleware placement (as a Policy Enforcement Point) and secure JWT handling are key to a robust solution[[13]](https://www.osohq.com/post/microservices-authorization-patterns#:~:text=It%20is%20common%20for%20the,service%20in%20the%20example%20above)[[22]](https://stackoverflow.com/questions/59786338/how-to-implement-role-based-security-in-microservices-architecture#:~:text=,as%20directly%20to%20your%20microservices).

**Sources:** We leveraged best-practice guides on microservice authorization patterns[[11]](https://www.osohq.com/post/microservices-authorization-patterns#:~:text=One%20clean%20solution%20to%20the,authorization%20decisions%20based%20on%20that)[[19]](https://www.osohq.com/post/microservices-authorization-patterns#:~:text=When%20speaking%20with%20engineering%20teams%2C,into%20a%20dedicated%20authorization%20service), Node.js security practices[[5]](https://www.aegissofttech.com/insights/securing-microservices-in-node-js/#:~:text=,between%20services%2C%20and%20verification%20securely)[[23]](https://www.aegissofttech.com/insights/securing-microservices-in-node-js/#:~:text=Microservices%20need%20to%20be%20kept,controlling%20and%20monitoring%20access%20rights), and JWT usage[[4]](https://api7.ai/learning-center/api-101/implementing-jwt-for-api-security#:~:text=%2F%2F%20Example%20payload%20in%20Node,user%27%2C%20%27editor%27%5D)[[9]](https://api7.ai/learning-center/api-101/implementing-jwt-for-api-security#:~:text=,any%20other%20necessary%20claim%20validations) to inform this design. These principles ensure a comprehensive, maintainable RBAC implementation in a Node.js ecosystem.

[[1]](https://www.aegissofttech.com/insights/securing-microservices-in-node-js/#:~:text=API%20Gateway%20is%20the%20single,that%20come%20out%20of%20them) [[5]](https://www.aegissofttech.com/insights/securing-microservices-in-node-js/#:~:text=,between%20services%2C%20and%20verification%20securely) [[23]](https://www.aegissofttech.com/insights/securing-microservices-in-node-js/#:~:text=Microservices%20need%20to%20be%20kept,controlling%20and%20monitoring%20access%20rights) Top 10 Best Practices for Securing Microservices in Node.js

<https://www.aegissofttech.com/insights/securing-microservices-in-node-js/>

[[2]](https://www.permit.io/blog/implement-multi-tenancy-rbac-in-mongodb#:~:text=,its%20own%20users%20and%20tickets) [[3]](https://www.permit.io/blog/implement-multi-tenancy-rbac-in-mongodb#:~:text=Instead%20of%20embedding%20access%20rules,approach%20ensures%20three%20important%20things) Implement Multi-Tenancy Role-Based Access Control (RBAC) in MongoDB

<https://www.permit.io/blog/implement-multi-tenancy-rbac-in-mongodb>

[[4]](https://api7.ai/learning-center/api-101/implementing-jwt-for-api-security#:~:text=%2F%2F%20Example%20payload%20in%20Node,user%27%2C%20%27editor%27%5D) [[6]](https://api7.ai/learning-center/api-101/implementing-jwt-for-api-security#:~:text=,Rotate%20keys%20periodically) [[7]](https://api7.ai/learning-center/api-101/implementing-jwt-for-api-security#:~:text=,that%20could%20intercept%20the%20token) [[8]](https://api7.ai/learning-center/api-101/implementing-jwt-for-api-security#:~:text=meaning%20the%20payload%20is%20Base64Url,Refresh%20tokens%20should%20be) [[9]](https://api7.ai/learning-center/api-101/implementing-jwt-for-api-security#:~:text=,any%20other%20necessary%20claim%20validations) [[10]](https://api7.ai/learning-center/api-101/implementing-jwt-for-api-security#:~:text=%2F%2F%20Example%20validation%20in%20Node,catch%20%28err%29) Implementing JWT for API Security - API7.ai

<https://api7.ai/learning-center/api-101/implementing-jwt-for-api-security>

[[11]](https://www.osohq.com/post/microservices-authorization-patterns#:~:text=One%20clean%20solution%20to%20the,authorization%20decisions%20based%20on%20that) [[12]](https://www.osohq.com/post/microservices-authorization-patterns#:~:text=Another%20solution%20is%20to%20put,and%20ask%20the%20authorization%20service) [[13]](https://www.osohq.com/post/microservices-authorization-patterns#:~:text=It%20is%20common%20for%20the,service%20in%20the%20example%20above) [[14]](https://www.osohq.com/post/microservices-authorization-patterns#:~:text=Note%20that%20using%20bare%20headers,often%20expressed%20as%20a%20JWT) [[15]](https://www.osohq.com/post/microservices-authorization-patterns#:~:text=The%20main%20benefit%20of%20the,trips) [[16]](https://www.osohq.com/post/microservices-authorization-patterns#:~:text=In%20this%20model%2C%20the%20documents,to%20make%20that%20decision) [[17]](https://www.osohq.com/post/microservices-authorization-patterns#:~:text=Of%20course%2C%20this%20separation%20of,become%20a%20bottleneck%20for%20new) [[18]](https://www.osohq.com/post/microservices-authorization-patterns#:~:text=There%20are%20other%20factors%20that,queries%2C%20every%20request%20is%20slow) [[19]](https://www.osohq.com/post/microservices-authorization-patterns#:~:text=When%20speaking%20with%20engineering%20teams%2C,into%20a%20dedicated%20authorization%20service) [[20]](https://www.osohq.com/post/microservices-authorization-patterns#:~:text=This%20can%20be%20quite%20appealing%3A,rest%20of%20your%20overall%20system) Best Practices for Authorization in Microservices

<https://www.osohq.com/post/microservices-authorization-patterns>

[[21]](https://www.alexanderlolis.com/authorization-in-a-microservices-world#:~:text=Authorization%20flow%20overview) Authorization in a microservices world | Alexander's Blog

<https://www.alexanderlolis.com/authorization-in-a-microservices-world>

[[22]](https://stackoverflow.com/questions/59786338/how-to-implement-role-based-security-in-microservices-architecture#:~:text=,as%20directly%20to%20your%20microservices) spring boot - how to implement role-based security in microservices architecture - Stack Overflow

<https://stackoverflow.com/questions/59786338/how-to-implement-role-based-security-in-microservices-architecture>