ChatGPT

OAuth 2.0 Dynamic Client Registration (RFC 7591) – Implementation Guide

Executive Summary: OAuth 2.0 Dynamic Client Registration (RFC 7591) allows clients (applications) to register themselves with an authorization server (AS) at runtime. This is essential in open environments like the Model Context Protocol (MCP), where clients and servers have no pre-established relationship. RFC 7591 (with its companion RFC 7592 for management) specifies a **client registration endpoint** to which a client (or developer) can send an HTTP **POST** request containing client metadata (e.g. `redirect_uris`, `client_name`, authentication methods, etc.) as a JSON object [1] [2]. The server **must** respond with an HTTP **201 Created** and a JSON body including a new `client_id` (required) and, if applicable, a `client_secret` (for confidential clients) along with all registered metadata [2] [3]. If client management is supported, the response also **must** include a `registration_access_token` (a bearer token for managing the client) and `registration_client_uri` (the URL of the client's configuration endpoint) [4] [5].

Following registration, the client (using the returned `registration_access_token`) can **read**, **update**, or **delete** its registration via standard REST methods on the configuration endpoint (`GET`, `PUT`, `DELETE` at `/register/{client_id}` for example) [6] [7]. Successful reads/updates return **200 OK** with the client metadata (possibly rotating the secret or token), while successful delete returns **204 No Content** [8] [9]. Error conditions (invalid input, unauthorized, etc.) use standard OAuth error formats (HTTP 400/401/403 with JSON `error` codes like `invalid_client_metadata` or `invalid_redirect_uri`) [10] [11].

Security is paramount: **TLS is mandatory** for all endpoints [12], registration tokens must have high entropy (to resist guessing) [13], and the AS must carefully validate input (redirect URIs, JSON fields, etc.) to avoid CSRF or injection attacks. Clients authenticate at the token endpoint using their chosen `token_endpoint_auth_method` (e.g. `client_secret_basic`, `none` for public clients) but that is separate from registration. Logging and auditing should record registration events without exposing secrets.

The following sections detail required endpoints, JSON schemas, parameters, token handling, validation rules, error handling, security concerns, lifecycle operations (register/read/update/delete) and their HTTP semantics, concurrency/idempotency concerns, and interoperability notes. **Implementation Checklist:**
- **Endpoints & Methods:** `/register` for POST (create), `/register/{client_id}` for GET/PUT/DELETE (read/update/delete) [1] [7].
- **JSON Schema:** Define models for *ClientRegistrationRequest* (metadata fields) and *ClientInformationResponse* (all client fields including issued IDs/secrets) [2] [4].
- **Required Fields:** At minimum, if using code/auth flows: `redirect_uris` (array of URIs) [14] and `token_endpoint_auth_method` (if omitted, defaults to `"client_secret_basic"` [15]). Always require a well-formed JSON object.
- **Optional Fields:** e.g. `client_name`, `client_uri`, `logo_uri`, `scope`, `contacts`, `tos_uri`, `policy_uri`, `jwks_uri` or `jwks` (but not both) [16] [17], `software_id`/`software_version` [18],

custom extensions. Accept internationalized variants using `#lang` suffix [19]. Unknown fields must be ignored [20] or cause an `invalid_client_metadata` error if unacceptable.
- **Registration Access Token:** If management is supported, generate a high-entropy bearer token on successful registration [4]. Return it in the response as `registration_access_token` with the client info [4] [5]. This token is used as the sole authorization for subsequent GET/PUT/DELETE on the client's record [6]. It should **not expire** while the client is active (but may be rotated on use) [21]. Store it securely (e.g. hashed) and require TLS.
- `client_id`**/**`client_secret`**:** Generate a unique `client_id` (e.g. UUID or random string) and, if a confidential client (not `token_endpoint_auth_method = none`), a strong `client_secret`. Store these (preferably the secret hashed or encrypted) in your client database. Return `client_id_issued_at` and `client_secret_expires_at` (epoch times) in the response if using client_secret [22].
- **Validation Rules:** Validate all fields: e.g. each `redirect_uri` must exactly match a valid absolute URI and (for code flows) begin with `https` except localhost [14] [23]. Enforce consistency: if `grant_types` includes `"authorization_code"`, `response_types` must include `"code"` (and vice versa) [24] [25]. If `jwks_uri` is provided, it must point to a valid JWK Set; if `jwks` is used, it must be well-formed JSON [17]. Reject or sanitize missing or malformed required metadata with an `invalid_client_metadata` error [26].
- **Software Statement:** Support signed software statements (JWTs) if needed. Accept them via the `software_statement` parameter (JWT string) in the JSON request [27] [28]. Verify the JWS signature and issuer (using a trusted key), extract claims like `client_name`, `client_uri`, etc. Any metadata in the statement **overrides** the corresponding JSON fields [27] [28]. Return the (unchanged) `software_statement` back in the response if accepted [29]. If the statement is invalid or unapproved, return error `invalid_software_statement` or `unapproved_software_statement` [30].
- **Error Handling:** On malformed request or invalid metadata, respond with HTTP **400 Bad Request** and a JSON `{ "error": "...", "error_description": "..." }` [10]. Use the defined error codes (`invalid_redirect_uri`, `invalid_client_metadata`, etc.) [31]. On unauthorized (missing/invalid bearer token at config endpoint), return **401 Unauthorized** (per RFC 6750) [32] [33]. On forbidden (authenticated but not permitted), return **403 Forbidden** [34] [35]. If HTTP methods other than allowed are used, respond **405 Method Not Allowed** [36]. Always include caching headers (`Cache-Control: no-store, Pragma: no-cache`) on sensitive endpoints.
- **Lifecycle Operations & HTTP Semantics:** - **Register (Create):** `POST /register` with JSON body. On success: **201 Created** with full client info JSON [37]. Include `Location: {registration_client_uri}` header if possible.
- **Read:** `GET /register/{client_id}` with `Authorization: Bearer {registration_access_token}`. On success: **200 OK** with JSON of client info (same schema as create response, including possibly new `client_secret` or `registration_access_token` if rotated) [38] [39]. If token invalid or client not found: **401** (and revoke token) [40].
- **Update:** `PUT /register/{client_id}` with JSON body containing the *entire* client metadata set to retain (omitting a field deletes it) [41]. The request *must* include `client_id` (and `client_secret` if used) in the body, matching the current values [42]. On success: **200 OK** with updated client info JSON (per 3.2.1) [43]. New secret or reg token may be returned, in which case the client must discard the old [43]. On invalid input: **400** with error as above. On auth failure or missing client: **401/403** as above [44].
- **Delete:** `DELETE /register/{client_id}` with `Authorization: Bearer {registration_access_token}`. On success: **204 No Content**, invalidating `client_id`, secret and reg

token [45] . If not supported: **405**. If token invalid or client missing: **401** and revoke token [46] ; if forbidden: **403**.

- **Concurrency & Idempotency:** The protocol does not define idempotency keys. You must handle concurrent requests safely (e.g. use transactions or optimistic locks) to avoid duplicated or conflicting client entries. If two identical registration requests arrive, you might respond with the same `client_id` (idempotent) or create duplicates. Consider rejecting a second create with HTTP **409 Conflict** if a duplicate `software_id` or initial token is detected. For update, use an `If-Match` header with a version or `client_id_issued_at` to ensure changes aren't lost. Document behavior clearly.

- **Logging & Auditing:** Log all registration attempts and outcomes (success, failure) with timestamps, client_id (if assigned), requestor IP/user-agent, but never log `client_secret` or bearer tokens in plaintext. Audit trails should record who created/updated/deleted each client registration, and when. Ensure logs comply with privacy (don't log end-user PII, etc.) and are protected.

- **Interoperability Pitfalls & Edge Cases:** Different implementations vary on:

- **Redirect URI strictness:** RFC 7591 expects exact URI matches. Beware of clients omitting trailing slashes, using ports, or unusual schemes. Reject invalid URIs with `invalid_redirect_uri` [47] .
- **Language tags:** JSON keys with `#` (e.g. `client_name#ja`) are valid JSON but need special handling in many languages [48] . In .NET use `Dictionary<string,string>` for dynamic keys or JSON patch.
- `jwks` **vs** `jwks_uri`**:** Ensure mutual exclusion as mandated [17] .
- **Public vs Confidential:** A client setting `"token_endpoint_auth_method":"none"` (public client) should not receive a `client_secret`.
- **Initial Access Tokens:** If your server requires an initial token for registration (out of band), ensure it is validated. This is AS-specific and out of scope of RFC 7591, but must be documented. Clients typically present `Authorization: Bearer {token}`.
- **Backward Compatibility:** Clients or identity servers may expect the OpenID Connect variant of DCR. RFC 7591 is compatible with OIDC Dynamic Registration, but some fields differ (OIDC may include `redirect_uri` vs `redirect_uris`, etc.). Test against known providers. If you had pre-existing static clients, you may seed them into the dynamic store or support fixed registrations alongside dynamic.

The **diagram below** illustrates the registration and management flows:

```
sequenceDiagram
    participant Client
    participant AuthServer
    Client->>AuthServer: POST /register (metadata JSON)
    AuthServer-->>Client: 201 Created (JSON with client_id, [client_secret],
registration_access_token, registration_client_uri, etc.)
    Client->>AuthServer: GET /register/{client_id} (Bearer
registration_access_token)
    AuthServer-->>Client: 200 OK (client info JSON)
    Client->>AuthServer: PUT /register/{client_id} (Bearer
registration_access_token, updated metadata JSON)
    AuthServer-->>Client: 200 OK (updated client info JSON)
    Client->>AuthServer: DELETE /register/{client_id} (Bearer
registration_access_token)
    AuthServer-->>Client: 204 No Content
```

Client Metadata: Required vs. Optional Fields

RFC 7591 defines a set of standard client metadata fields. All are *optional* unless noted, but some clients require certain fields (e.g. `redirect_uris` for code flow) [14]. Unknown fields **must be ignored or removed** [20] unless a policy forbids them.

Field	Required (Registration)	Type	Description
`redirect_uris`	*Conditionally* (see note)	Array of URI	Array of allowed redirection URIs. **Required** if the client will use a redirect-based flow (authorization code or implicit) [14]. Must be valid absolute URIs (HTTPS recommended) [14] [47].
`token_endpoint_auth_method`	Optional (default `client_secret_basic`)	String	Authentication method for the token endpoint. Standard values: `"client_secret_basic"`, `"client_secret_post"`, or `"none"` for public clients [15]. If omitted, defaults to `"client_secret_basic"` [49] Extensions via IANA registry or absolute URIs are allowed.
`grant_types`	Optional (default `["authorization_code"]`)	Array of String	OAuth 2.0 grant types the client will use (e.g. `"authorization_code"`, `"refresh_token"`, `"client_credentials"`, `"password"`, or JWT/SAML assertion types) [50]. If omitted, defaults to `["authorization_code"]`. Values must be consistent with `response_types`.

Field	Required (Registration)	Type	Description
`response_types`	Optional (default `["code"]`)	Array of String	OAuth 2.0 response types (e.g. `"code"`, `"token"` for implicit) [51] . If omitted, defaults to `["code"]`. Must align with `grant_types` (e.g. if `grant_types` includes `"implicit"`, `response_types` should include `"token"`) [24] .
`client_name`	Optional (RECOMMENDED)	String	Human-readable name of the client (displayed on consent screens) [16] . May include Unicode. Multiple languages allowed via `client_name#lang` variants [19] . If omitted, the AS may display the `client_id`.
`client_uri`	Optional (RECOMMENDED)	URL (string)	URL for client information page [52] . Must be a valid web page URL.
`logo_uri`	Optional	URL (string)	URL to the client's logo image [53] (valid image file).
`scope`	Optional	String (space-separated)	Default scopes for client (service-specific semantics) [54] . If omitted, AS may assign defaults.
`contacts`	Optional	Array of Strings	Contact email addresses for client owner [55] . For end-user support.
`tos_uri`	Optional	URL (string)	URL to the client's terms-of-service [56] .
`policy_uri`	Optional	URL (string)	URL to the client's privacy policy [57] .
`jwks_uri`	Optional	URL (string)	URL referencing JSON Web Key Set with client's public keys [17] . Preferred to `jwks`.

Field	Required (Registration)	Type	Description
`jwks`	Optional	JSON object	JWK Set object containing client's public keys [58]. Mutually exclusive with `jwks_uri`.
`software_id`	Optional	String	Identifier for the client software/application (e.g. UUID) [59]. Remains constant across versions.
`software_version`	Optional	String	Version identifier for `software_id` [60]. Changes with software updates.
`token_endpoint_auth_signing_alg` (OAuth2.1)	Optional	String (JWT alg)	(*Not in RFC 7591; in OAuth 2.1 draft or OIDC*) Preferred algorithm for token endpoint JWT client auth (e.g. RS256, ES256). Clients can propose this if using JWT auth.
`software_statement`	Optional	JWT string	A signed JWT containing client metadata claims [27]. If present, metadata inside it overrides JSON fields [27]. Must be JWS-signed with trusted key.

- **Notes:**
- RFC 7591 **does not** register a discovery mechanism for finding `/register`; in practice this URL is out-of-band or documented by the AS. Common implementations use `/register` or `/connect/dcr`.
- The **Authorization** header (Bearer token) is *not* part of the JSON schema but is used to send the initial access token (if registration requires one) and the `registration_access_token` on subsequent calls.
- `client_id`, `client_secret`, `registration_access_token`, `registration_client_uri`, `client_id_issued_at`, `client_secret_expires_at` are **not** sent by the client in a create/update request; they are returned by the server in responses [4] [2].

Example JSON Payloads

- **Registration (POST /register) – Request Example:**

```
{
  "redirect_uris": [
    "https://client.example.org/callback",
    "https://client.example.org/alt"
  ],
  "client_name": "My Example Client",
  "token_endpoint_auth_method": "client_secret_basic",
  "grant_types": ["authorization_code", "refresh_token"],
  "response_types": ["code"],
  "logo_uri": "https://client.example.org/logo.png",
  "jwks_uri": "https://client.example.org/keys.json",
  "scope": "read write",
  "contacts": ["admin@client.example.org"],
  "client_name#ja-Jpan-JP": "クライアント名"
}
```

All above fields except `redirect_uris` and `client_name` are optional.

- **Registration – Success Response (201 Created):**
  The server returns **201** with JSON including all registered metadata. Required fields:

```
{
  "client_id": "s6BhdRkqt3",
  "client_secret": "cf136dc3c1fc93f31185e5885805d",
  "client_id_issued_at": 1610078132,
  "client_secret_expires_at": 0,
  "registration_access_token": "reg-23410913-abewfq.123483",
  "registration_client_uri": "https://server.example.com/register/
s6BhdRkqt3",
  "redirect_uris": [
    "https://client.example.org/callback",
    "https://client.example.org/alt"
  ],
  "grant_types": ["authorization_code","refresh_token"],
  "response_types": ["code"],
  "token_endpoint_auth_method": "client_secret_basic",
  "client_name": "My Example Client",
  "logo_uri": "https://client.example.org/logo.png",
  "jwks_uri": "https://client.example.org/keys.json",
  "scope": "read write",
  "contacts": ["admin@client.example.org"]
}
```

This contains the issued `client_id`, optional `client_secret`, issuance timestamps, **and** the `registration_access_token` and `registration_client_uri` for management [4] [61]. The server echo the metadata fields, possibly substituting defaults if it modified any.

- **Read (GET /register/{client_id}) – Success Response (200 OK):**

```
{
  "client_id": "s6BhdRkqt3",
  "client_secret": "cf136dc3c1fc93f31185e5885805d",
  "client_id_issued_at": 1610078132,
  "client_secret_expires_at": 0,
  "registration_access_token": "reg-23410913-abewfq.123483",
  "registration_client_uri": "https://server.example.com/register/
s6BhdRkqt3",
  "redirect_uris": [
    "https://client.example.org/callback",
    "https://client.example.org/alt"
  ],
  "grant_types": ["authorization_code","refresh_token"],
  "token_endpoint_auth_method": "client_secret_basic",
  "client_name": "My Example Client",
  "logo_uri": "https://client.example.org/logo.png",
  "jwks_uri": "https://client.example.org/keys.json"
}
```

Same schema as above. Note the presence of `registration_*` fields. [62]

- **Update (PUT /register/{client_id}) – Request Example:**
  The client must send *all* metadata fields it wants to retain. For example, to update the name and logo:

```
{
  "client_id": "s6BhdRkqt3",
  "client_secret": "cf136dc3c1fc93f31185e5885805d",
  "redirect_uris": [
    "https://client.example.org/callback",
    "https://client.example.org/alt"
  ],
  "grant_types": ["authorization_code","refresh_token"],
  "token_endpoint_auth_method": "client_secret_basic",
  "jwks_uri": "https://client.example.org/keys.json",
  "client_name": "My Renamed Client",
  "client_name#fr": "Mon Nouveau Client",
  "logo_uri": "https://client.example.org/newlogo.png"
}
```

Observe that `client_id` and (if present) `client_secret` must match the current values, and fields omitted (e.g. `scope`) are removed [42] [63].

- **Update – Success Response (200 OK):** Returns updated client info (see Read response). If a new secret or reg token was issued, it must be returned (then the client discards the old) [43].

- **Delete (DELETE /register/{client_id}) – Request:** No JSON payload. On success, **204 No Content**.
- **Error Example (400 Bad Request):**

```
{
  "error": "invalid_redirect_uri",
  "error_description": "The redirection URI http://bad.example.com is not
allowed."
}
```

(HTTP 400) 64 .

Sample .NET Web API Implementation

Below is a sketch of how one might implement these endpoints in ASP.NET Core. *Note: database and hosting are unspecified – the code shows patterns, not a full app.*

Controller & Models

```
// Model for registration request (JSON fields as C# properties)
public class ClientRegistrationRequest
{
    public List<string> RedirectUris { get; set; }
    public string ClientName { get; set; }
    public string TokenEndpointAuthMethod { get; set; } = "client_secret_basic";
    public List<string> GrantTypes { get; set; }
    public List<string> ResponseTypes { get; set; }
    public string LogoUri { get; set; }
    public string JwksUri { get; set; }
    public string Scope { get; set; }
    public List<string> Contacts { get; set; }
    // ... other fields (client_uri, tos_uri, etc) ...
    // Supports dynamic language tags via JSON extension if needed.
}

// Model for client info (to store in DB)
public class ClientRecord
{
    public string ClientId { get; set; }
    public string HashedSecret { get; set; } // store hash, not plaintext
    public DateTimeOffset ClientIdIssuedAt { get; set; }
    public DateTimeOffset? ClientSecretExpiresAt { get; set; }
    public string RegistrationAccessToken { get; set; } // hashed or raw
    public DateTimeOffset CreatedAt { get; set; }
    // Metadata fields:
    public List<string> RedirectUris { get; set; }
    public string ClientName { get; set; }
    public string TokenEndpointAuthMethod { get; set; }
    public List<string> GrantTypes { get; set; }
    public List<string> ResponseTypes { get; set; }
    public string LogoUri { get; set; }
    public string JwksUri { get; set; }
    public string Scope { get; set; }
    public List<string> Contacts { get; set; }
    // ... etc ...
}
```

```
[ApiController]
[Route("register")]
public class RegistrationController : ControllerBase
{
    private readonly IClientStore _clientStore;
    private readonly ISecretGenerator _generator;
    private readonly ILogger _logger;

    public RegistrationController(IClientStore clientStore, ISecretGenerator
generator, ILogger<RegistrationController> logger)
    {
        _clientStore = clientStore;
        _generator = generator;
        _logger = logger;
    }

    [HttpPost]
    public async Task<IActionResult> Register([FromBody]
ClientRegistrationRequest req)
    {
        // Validate JSON schema and mandatory fields
        if (req.RedirectUris == null || !req.RedirectUris.Any())
        {
            return BadRequest(new { error = "invalid_client_metadata",
                                     error_description = "Missing
redirect_uris." });
        }
        // (Additional validation of URIs, client name, etc., omitted for
brevity)

        // Optional: check for initial access token if required
        // if (!IsValidInitialAccessToken(Request)) return Unauthorized();
        // Generate new client record
        var clientId = Guid.NewGuid().ToString("N");
        string clientSecret = null;
        if (req.TokenEndpointAuthMethod != "none")
        {
            clientSecret = _generator.GenerateSecret(); // high-entropy string
        }

        // Hash secret for storage (Microsoft.AspNetCore.Identity)
        string hashedSecret = null;
        if (clientSecret != null)
        {
            var hasher = new PasswordHasher<ClientRecord>();
            hashedSecret = hasher.HashPassword(null, clientSecret);
        }

        // Generate registration access token
        var regToken = _generator.GenerateToken(); // e.g. random GUID
        // Optionally hash regToken for storage

        var now = DateTimeOffset.UtcNow;
        var record = new ClientRecord {
            ClientId = clientId,
            HashedSecret = hashedSecret,
            ClientIdIssuedAt = now,
            ClientSecretExpiresAt = clientSecret == null ?
(DateTimeOffset?)null : now.AddYears(1),
            RegistrationAccessToken = regToken,
            CreatedAt = now,
            RedirectUris = req.RedirectUris,
            ClientName = req.ClientName,
            TokenEndpointAuthMethod = req.TokenEndpointAuthMethod,
            GrantTypes = req.GrantTypes ?? new List<string> {
"authorization_code" },
            ResponseTypes = req.ResponseTypes ?? new List<string> { "code" },
            LogoUri = req.LogoUri,
            JwksUri = req.JwksUri,
            Scope = req.Scope,
            Contacts = req.Contacts
            // ... assign other metadata ...
        };
        _clientStore.Save(record);

        // Build response
        var response = new Dictionary<string, object>
        {
            ["client_id"] = clientId
        };
        if (clientSecret != null)
        {
            response["client_secret"] = clientSecret;
            response["client_secret_expires_at"] =
record.ClientSecretExpiresAt.HasValue
                ?
((DateTimeOffset)record.ClientSecretExpiresAt).ToUnixTimeSeconds() : 0;
        }
        response["client_id_issued_at"] =
record.ClientIdIssuedAt.ToUnixTimeSeconds();
        response["registration_access_token"] = regToken;
        response["registration_client_uri"] =
Url.ActionLink(nameof(ClientConfigurationController.GetClient),
                                            values: new { clientId =
clientId });

        // Include metadata fields (echo back)
        response["redirect_uris"] = record.RedirectUris;
        response["grant_types"] = record.GrantTypes;
        response["response_types"] = record.ResponseTypes;
        response["token_endpoint_auth_method"] = record.TokenEndpointAuthMethod;
        response["client_name"] = record.ClientName;
        response["logo_uri"] = record.LogoUri;
        response["jwks_uri"] = record.JwksUri;
        response["scope"] = record.Scope;
        response["contacts"] = record.Contacts;
        // ... additional fields ...

        _logger.LogInformation("Registered new client {ClientId}", clientId);
        return Created(response["registration_client_uri"].ToString(),
response);
    }
}
```

For the **Client Configuration** (read/update/delete), one might use a separate controller or extend the same. Example:

```
[ApiController]
[Route("register/{clientId}")]
public class ClientConfigurationController : ControllerBase
{
    private readonly IClientStore _clientStore;
    private readonly ILogger _logger;

    public ClientConfigurationController(IClientStore clientStore,
ILogger<ClientConfigurationController> logger)
```

```
    {
        _clientStore = clientStore;
        _logger = logger;
    }

    // GET /register/{clientId}
    [HttpGet]
    public IActionResult GetClient(string clientId)
    {
        var record = _clientStore.Find(clientId);
        if (record == null) return Unauthorized(); // revoke token, client gone
        // Validate registration_access_token
        if (!ValidateRegistrationToken(record, Request))
            return Unauthorized();
        // Build response (same as register response)
        var resp = new {
            client_id = record.ClientId,
            client_secret = /* Note: never return hashed; if rotating, you need
to generate new plain-secret */,
            client_id_issued_at = record.ClientIdIssuedAt.ToUnixTimeSeconds(),
            client_secret_expires_at =
record.ClientSecretExpiresAt?.ToUnixTimeSeconds() ?? 0,
            registration_access_token = record.RegistrationAccessToken,
            registration_client_uri = /* this URL */,
            redirect_uris = record.RedirectUris,
            grant_types = record.GrantTypes,
            response_types = record.ResponseTypes,
            token_endpoint_auth_method = record.TokenEndpointAuthMethod,
            client_name = record.ClientName,
            logo_uri = record.LogoUri,
            jwks_uri = record.JwksUri,
            scope = record.Scope,
            contacts = record.Contacts
        };
        return Ok(resp);
    }

    // PUT /register/{clientId}
    [HttpPut]
    public IActionResult UpdateClient(string clientId, [FromBody]
ClientRegistrationRequest req)
    {
        var record = _clientStore.Find(clientId);
        if (record == null) return Unauthorized();
        if (!ValidateRegistrationToken(record, Request)) return Unauthorized();
        // Check client_id/client_secret in body matches
        if (req.ClientId != clientId) return BadRequest(new { error =
"invalid_client_metadata" });
```

```
        // (Assume req had client_secret field too, validate it if provided)
        // Update all metadata (replace with req values or nullify)
        record.RedirectUris = req.RedirectUris;
        record.ClientName = req.ClientName;
        record.TokenEndpointAuthMethod = req.TokenEndpointAuthMethod;
        record.GrantTypes = req.GrantTypes;
        record.ResponseTypes = req.ResponseTypes;
        record.LogoUri = req.LogoUri;
        record.JwksUri = req.JwksUri;
        record.Scope = req.Scope;
        record.Contacts = req.Contacts;
        // If client_secret is present in req and valid, we cannot overwrite it.
        // (Clients are not allowed to set their own secret here.)
        _clientStore.Save(record);
        _logger.LogInformation("Updated client {ClientId}", clientId);
        // Return updated info (reuse Get logic)
        return GetClient(clientId);
    }

    // DELETE /register/{clientId}
    [HttpDelete]
    public IActionResult DeleteClient(string clientId)
    {
        var record = _clientStore.Find(clientId);
        if (record == null) return Unauthorized();
        if (!ValidateRegistrationToken(record, Request)) return Unauthorized();
        _clientStore.Delete(clientId);
        _logger.LogInformation("Deleted client {ClientId}", clientId);
        return NoContent();
    }

    private bool ValidateRegistrationToken(ClientRecord record, HttpRequest req)
    {
        string auth = req.Headers["Authorization"];
        if (string.IsNullOrEmpty(auth) || !auth.StartsWith("Bearer ")) return
false;
        var token = auth.Substring("Bearer ".Length).Trim();
        // Compare token (hash compare if stored hashed)
        return (token == record.RegistrationAccessToken);
    }
}
```

Middleware and Configuration:
- Use ASP.NET Core's built-in **authentication** for any initial access tokens or JWTs. For example, if initial tokens are JWTs, use `AddJwtBearer`. For `registration_access_token` you could either use the above custom validation or issue it as a JWT and let the framework validate it (setting `IssuerSigningKey`, etc.).

- Always call `app.UseHttpsRedirection();` and enforce `[RequireHttps]` for controllers to guarantee TLS.
- Use `ILogger` for logging. Sensitive data (like secrets) should never be logged.

Security & Secrets

- **Transport Security:** Always require HTTPS (TLS 1.2+) for both registration and configuration endpoints [65]. Use HSTS. Ensure certificate validation per RFC 6125 [65].
- **Protect Secrets:** Store `client_secret` and `registration_access_token` securely (hash or encrypt at rest). Only expose the *plain* `client_secret` once on creation (not on reads). For `registration_access_token`, you may keep it raw or sign it as a JWT. Use at least 128-bit random values [66].
- **CSRF:** Since registration is typically a machine-to-machine API (no browser session), CSRF is minimal. If you provide any web UI around registration, ensure CSRF tokens. For pure API, require Bearer tokens or other auth for non-idempotent calls.
- **Replay Protection:** Use OAuth Bearer token standards (RFC 6750) – tokens expire or are rotated on read/update. Consider using `cache-control: no-store` to prevent replay via caches.
- **Input Validation:** Strictly validate JSON fields (no HTML/JS in URLs, length limits, valid charsets, etc.) to prevent injection. Use data annotations or manual checks (e.g. `Uri.TryCreate`). Reject any malicious input with errors, never throw raw exceptions to clients.

Recommended Libraries & Packages

- **IdentityServer4/Duende (Optional):** Frameworks like Duende IdentityServer have built-in DCR support [67]. If using them, enable the Configuration API.
- **OpenIddict:** Offers dynamic client registration support in ASP.NET.
- **Microsoft.AspNetCore.Authentication.JwtBearer:** For validating JWTs (if reg token is JWT).
- **Microsoft.AspNetCore.Identity:** Use `PasswordHasher<T>` for hashing client secrets.
- **System.Text.Json** or **Newtonsoft.Json:** JSON serialization; ensure proper casing (RFC uses snake_case).
- **Entity Framework Core:** For persisting client records (define a `DbSet<ClientRecord>`).
- **FluentValidation** or **DataAnnotations:** For request validation of client metadata fields.
- **NSwag/Swagger:** Document your DCR API endpoints for clarity.

Testing Strategies

- **Unit Tests:** Test validation logic (missing fields, invalid URIs, grant/response mismatches). Mock the client store to test controller logic.
- **Integration Tests:** Spin up the API with in-memory DB. Use `TestServer` or `WebApplicationFactory` to simulate HTTP requests: POST /register, GET/PUT/DELETE with tokens. Verify status codes and response bodies.
- **Negative Tests:** Attempt invalid flows: missing Bearer token, invalid token, invalid JSON, conflicting grant/response types, blacklisted redirect URI (should get `invalid_redirect_uri`), etc. Ensure proper RFC error codes and messages.
- **Concurrency Tests:** Simulate two parallel POST /register with same input; ensure either two distinct clients or a clean error. Test simultaneous update and delete.
- **API Tests:** Use Postman/Newman or curl scripts covering the full lifecycle (e.g. create a client, read it, update name, delete it). Check that after delete, the client ID and token no longer work (401).

Migration & Backward-Compatibility

- If upgrading from **static registration** (pre-configured clients), you can bootstrap those clients into the dynamic store with existing IDs/secrets. Provide a migration script or allow an admin API to import clients. The dynamic API should then return the old values.
- Clients expecting OpenID Connect's dynamic registration can generally work with RFC 7591's JSON (they are compatible), but verify field names. Some older specs used `redirect_uri` (singular) – ensure you only support the RFC's array form.
- If you previously did not support read/update/delete, adding them now means clients that didn't know about `registration_access_token` may ignore those fields – document the change.
- **Versioning:** If you foresee changes (e.g. OAuth 2.1), consider a version in your endpoint (e.g. `/register/v2`) or use content negotiation.

Implementation Checklist

- [ ] **Registration Endpoint (`POST /register`):** Accept JSON metadata, protected by TLS. Optional initial access token. On success, create new client and respond 201.
- [ ] **Client ID/Secret:** Generate secure random values (e.g. GUID/cryptographic RNG for ID, long random for secret). Store the secret hashed. Include `client_id_issued_at` and `client_secret_expires_at` in response.
- [ ] **Response JSON:** Return `client_id` (required), `client_secret` (if issued), `registration_access_token`, `registration_client_uri`, plus all registered metadata fields [2] [4]. JSON object must not be wrapped (top-level object) [3].
- [ ] **Field Defaults:** If client omits `grant_types`/`response_types`, default to `["authorization_code"]` and `["code"]` respectively [50] [51]. If omits `token_endpoint_auth_method`, default to `client_secret_basic` [49].
- [ ] **Metadata Validation:** Strictly validate all metadata (especially `redirect_uris` – see OAuth 2.0 Section 3.1 and 10.3, and disallow wildcards unless your policy allows). Enforce relationship between `grant_types` and `response_types` [24]. Verify `jwks_uri` if present (fetch and parse?) or trust it.
- [ ] **Software Statement:** If supporting, parse and validate JWT (e.g. using `System.IdentityModel.Tokens.Jwt`). Use the JWT's claims to override request fields. Ensure you have a trust framework (e.g. check the signing key or `iss`). On failure, return `invalid_software_statement` [27] [68].
- [ ] **Registration Access Token:** Create a random bearer token (e.g. GUID, or JWT with random claims). Store it (preferably hashed). Return it in `registration_access_token` in all responses. Require it for GET/PUT/DELETE and rotate it on use if desired [21] [69].
- [ ] **Client Authentication Configuration:** Respect the `token_endpoint_auth_method` the client requests. (For example, if `"none"`, do not issue a `client_secret`.) Later, when issuing OAuth tokens, enforce that clients authenticate accordingly (Basic, POST, or JWT) at the token endpoint.
- [ ] **Error Responses:** For invalid input, return HTTP 400 with JSON `{ error: "...", error_description: "..." }`. Use the error codes defined in RFC 7591 [31]. For unauthorized/forbidden on GET/PUT/DELETE, use 401/403 as per RFC 7592 [44].
- [ ] **TLS & Security:** Enforce HTTPS/TLS on all endpoints (require TLS 1.2+). Use strict certificate validation. Do not allow insecure HTTP.

- [ ] **Storage:** Design a database (or key-value store) for client records. Include fields for all metadata plus secret hashes and tokens. Make concurrency safe (e.g. unique index on `client_id`, `software_id` if used).
- [ ] **Logging:** Log each registration, update, or deletion attempt with outcome. Do not log secret values. Monitor for abuse (e.g. many registrations from one IP).
- [ ] **Unit/Integration Tests:** Write tests covering all endpoints, success and error cases, and ensure compliance with RFC.

Sources: The above is drawn from RFC 7591 and related IETF documents, including RFC 7592 (registration management) [2] [4]. Security practices reference RFC 5246 (TLS) [65], RFC 6819 (OAuth threats) [66], and RFC 6750 (Bearer Token Usage) [2]. For OAuth basics, see RFC 6749.

---

[1] [2] [3] [10] [14] [15] [16] [17] [18] [19] [20] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [37] [47] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59] [60] [61] [64] [68] RFC 7591 - OAuth 2.0 Dynamic Client Registration Protocol
https://datatracker.ietf.org/doc/html/rfc7591

[4] [5] [6] [7] [8] [9] [11] [12] [13] [21] [32] [33] [34] [35] [36] [38] [39] [40] [41] [42] [43] [44] [45] [46] [62] [63] [65] [66] [69] RFC 7592: OAuth 2.0 Dynamic Client Registration Management Protocol
https://www.rfc-editor.org/rfc/rfc7592.html

[67] Dynamic Client Registration (DCR) | Duende Software Docs
https://docs.duendesoftware.com/identityserver/configuration/dcr/