ChatGPT

Building an MCP Server with the Kroger API

Building a multi-channel processing (MCP) server for Kroger involves integrating all of Kroger's public APIs – from product catalog search and store locations to cart management and loyalty profile retrieval. This guide provides a comprehensive walkthrough of **all Kroger API products and endpoints**, including how to authenticate via OAuth2, format requests and handle responses, manage errors and rate limits, leverage available tools (like OpenAPI specs and SDKs), and follow best practices for a secure, robust server integration.

Introduction to Kroger's API Ecosystem

Kroger's APIs enable developers to tap into the grocery retailer's data and services to build applications. The public API catalog covers:

- **Product Catalog** – search for products, retrieve product details, prices, and availability
- **Store Locations** – find store locations, chains, and departments
- **Cart** – create and modify a customer's shopping cart
- **Identity & Loyalty** – access basic user profile info (e.g. loyalty card number) for authenticated customers
- **Authorization** – OAuth2 endpoints for obtaining access tokens

All Kroger APIs are RESTful and accessed via HTTPS under a common base URL. The production base URL is `https://api.kroger.com/v1/` (with a parallel **certification** sandbox at `https://api-ce.kroger.com/v1/` for testing) [1]. Every request must include a valid OAuth2 access token in the header (e.g. `Authorization: Bearer <token>`) [2]. In the sections below, we cover each API category in detail, including endpoints, request/response schemas, and usage examples.

Authentication and OAuth2

Kroger's APIs use OAuth2 for authentication and authorization. **All API calls require an access token in the** `Authorization` **header** [2], and Kroger employs scope-based access control for different API products. To build your MCP server, you will need to register a Kroger Developer application to obtain a **Client ID** and **Client Secret**, then use the appropriate OAuth2 flow for your use case:

- **Client Credentials Grant** – Used for server-to-server calls that do not involve a specific user. This is suitable for "public" data like product search and store info. Your server exchanges its Client ID/Secret for an access token. For Kroger, you must include the desired API scope in this token request (e.g. the scope `product.compact` for product and location data) [3].
- **Authorization Code Grant** – Used for operations on behalf of a Kroger customer (e.g. modifying a user's cart or accessing loyalty info). This flow requires redirecting the user to Kroger's login/consent page and obtaining a code, which your server exchanges for an access token (and refresh token). The token will carry user-specific scopes (e.g. `cart.basic:write`, `profile.compact`) [4] that grant permission to modify their cart or read profile data.

Scopes: Kroger defines specific OAuth2 scopes for each area of the API. The main scopes available are [4]:

Scope Name	Access Granted
`product.compact`	Read-only access to product catalog and store/location info (no user context needed).
`cart.basic:write`	Read/Write access to a customer's cart (allows adding, updating, removing items).
`profile.compact`	Read access to basic user profile information (e.g. loyalty card number).

When requesting a token, your application specifies the needed scope(s). For client credential tokens, include the scope in the POST body to `/connect/oauth2/token`. For user authorization, include scopes when constructing the consent URL and token exchange.

Token Endpoints: Kroger's OAuth endpoints are under the base URL's `connect/oauth2` path. Key endpoints include:

- `POST /connect/oauth2/token` – Exchange credentials or auth code for an access token (and refresh token if using auth code). For a *client credentials* request, send `grant_type=client_credentials` and the desired `scope`. For an *authorization code* exchange, send `grant_type=authorization_code`, the `code` you received, the same `scope`(s) requested, and your `redirect_uri`. **HTTP Basic auth** is used for this endpoint: include an `Authorization: Basic <base64(client_id:client_secret)>` header [5] to authenticate your app. The request body must be form-encoded (`Content-Type: application/x-www-form-urlencoded`) [6] [7]. On success, the response contains a JSON with the `access_token`, its `expires_in` (lifetime), token `type`, and (for auth code flow) a `refresh_token`.

- `GET /connect/oauth2/authorize` – Used in the authorization code flow to initiate user consent. Your app directs the user's browser to this URL with query parameters including `client_id`, requested `scope`, `response_type=code`, `redirect_uri`, and a `state`. The user logs in and approves access, and Kroger then redirects back to your specified `redirect_uri` with a `code`. *(Note: You must configure the redirect URI in your Kroger app settings ahead of time. The OAuth flow will only redirect to whitelisted URIs.)*

Using the Access Token: Once obtained, the **access token must be included in the** `Authorization` **header for all API requests** (as a Bearer token) [2]. For example:

```
Authorization: Bearer eyJhbGciOiJI...   (token string)
```

Tokens have a limited lifespan (after which you will receive HTTP 401 Unauthorized errors). If using the auth code flow, utilize the `refresh_token` to get a new access token when needed by calling the `/token` endpoint with `grant_type=refresh_token`. In a long-running MCP server, implement automatic token refresh: before a token expires or upon a 401 response, use the refresh token to obtain a new `access_token` (and update the stored token). The Kroger API follows standard OAuth2 behavior where a

refresh token may be long-lived, but it can be invalidated if the user revokes access or if it's already been used (depending on Kroger's policy).

> **Development Tip:** When building your server, keep client credentials secure. Store the Client ID and Secret in environment variables or a secure vault – **never expose them on the client side**. For example, you might keep a `.env` file with: `KROGER_CLIENT_ID`, `KROGER_CLIENT_SECRET`, and your `KROGER_REDIRECT_URI` (for user flows) [8]. Use your backend server to handle all OAuth calls so that secrets remain protected.

Example (Client Credentials Flow): To illustrate, here's a simplified example in Python pseudocode obtaining a token and calling a product API:

```
import requests, base64
client_id = "<your_client_id>"
client_secret = "<your_client_secret>"
auth = base64.b64encode(f"{client_id}:{client_secret}".encode()).decode()
resp = requests.post(
    "https://api.kroger.com/v1/connect/oauth2/token",
    headers={"Authorization": f"Basic {auth}", "Content-Type": "application/x-
www-form-urlencoded"},
    data={"grant_type": "client_credentials", "scope": "product.compact"}
)
token = resp.json()['access_token']
# Use the token to call an API endpoint:
products_resp = requests.get(
    "https://api.kroger.com/v1/products?
filter.term=milk&filter.locationId=01400943",
    headers={"Authorization": f"Bearer {token}"}
)
print(products_resp.json())
```

In this example, we request the `product.compact` scope to search products. The returned token can then be used to query products or locations. In a real server, you would also handle token caching (reusing the token until expiry) and errors (e.g., invalid credentials or expired token scenarios). A similar approach applies for user-scoped tokens, except you must first obtain a `code` via user consent and then include `grant_type=authorization_code&code=<code>&redirect_uri=<uri>&scope=...` in the token request [9] [10].

Products (Catalog) API

The **Products API** (also referred to as the Catalog API in newer documentation) provides access to Kroger's product catalog, including item details, pricing, and availability. This API allows searching for products by various criteria and retrieving detailed information for specific products. It is a read-only API (GET requests) and does not require a user context (a client credentials token with the `product.compact` scope is sufficient for full access [3]).

Base path: `/v1/products`

Key Endpoints:

- **Search Products** – `GET /products?filter.term=<keyword>&filter.locationId=<storeId>&...`
  Retrieves a list of products matching the search criteria. You can search by a text term, and filter further by store location, brand, fulfillment option, etc. For example, `filter.term=milk` will search product names/descriptions for "milk". **If no query parameters are provided, the API returns all products** (which could be a very large result set) [11], so usually you will specify at least a term or product ID filter.
- *Filtering Options:* The query supports multiple `filter.*` parameters [12] [13]:
  - `filter.term` – Text search term (fuzzy match) [12].
  - `filter.productId` – A specific product identifier (Kroger's internal product ID or a UPC) [14].
  - `filter.locationId` – A store location ID. **Including this is important** if you want pricing, availability, and aisle location info in the results [11]. Without a `locationId`, the product data will omit price and in-stock status (since those vary by store).
  - `filter.brand` – Filter results by brand name [13].
  - `filter.fulfillment` – Filter by fulfillment type. (For example, you might specify a value to filter for items available for delivery, curbside pickup, in-store, etc., if applicable. Common values seen include `"csp"` for curbside pickup, `"del"` for delivery, or `"ais"` for in-store availability – these codes are used internally for fulfillment methods.)
  - `filter.start` (integer) – Pagination start index (0-indexed offset) [15].
  - `filter.limit` (integer) – Number of results per page (max page size). Defaults to 10 if not specified [16] [17].
- *Pagination:* The API supports paginated results. If your query has more matches than the `limit`, the response's `meta.pagination` object will indicate the total number of results and the current `start` offset [18]. You can retrieve subsequent pages by incrementing the `filter.start` value (e.g. `start=10` to get the next page if limit=10).
- **Example Request:**
  `GET /v1/products?filter.term=milk&filter.locationId=01400943&filter.limit=5`
  This searches for "milk" in the product catalog at store location `01400943` (store IDs can be obtained from the Locations API).
- **Response:** A successful response returns JSON with a top-level `data` array of products, and a `meta` object for pagination. Each product in `data` includes fields such as:
  - `productId` – Kroger's product ID (often a numeric or string code).
  - `description` – Name/description of the product.
  - `brand` – Brand name of the product.
  - `categories` – Category hierarchy array (e.g. `["Dairy", "Milk"]`).
  - `aisleLocations` – If `locationId` was provided, an array of aisle location info in that store (each with fields like `number` (aisle number), `description` (aisle name or section), etc.) [19].
  - `items` – A list of specific item variants for the product. Typically, this includes one or more objects each representing a specific UPC/package size that falls under that product. Each item object contains:

- `itemId` – The specific item's ID (often the 12-digit UPC).
- `size` – The package size/quantity (e.g. "1 gallon").
- `price` – An object with pricing information (keys like `regular` for normal price, and `promo` for a promotional price if any) [20].
- `inventory` – An object indicating stock status at the specified store (e.g. `stockLevel` which might be `"InStock"`, `"OutOfStock"`, etc.) [21].
- `fulfillment` – An object with booleans for fulfillment options (`curbside`, `delivery`, etc.) indicating if that item is available via those methods at that store [22].
- (Additional fields like `temperature` indicators, `images`, and `taxonomies` may also appear for each product for more detailed info like if an item is frozen/refrigerated, and product categorization codes [23] [24].)

Sample Response (excerpt): *The following JSON illustrates a simplified result for a product search.*

```
{
  "data": [
    {
      "productId": "0001111045678",
      "description": "Kroger 2% Reduced Fat Milk 1 Gallon",
      "brand": "Kroger",
      "categories": ["Dairy", "Milk & Cream"],
      "aisleLocations": [
        {
          "number": "5",
          "description": "Dairy"
        }
      ],
      "items": [
        {
          "itemId": "0001111045678",
          "size": "1 gal",
          "price": { "regular": 3.49, "promo": 2.99 },
          "inventory": { "stockLevel": "InStock" },
          "fulfillment": { "curbside": true, "delivery": true }
        }
      ]
    }
  ],
  "meta": {
    "pagination": { "total": 47, "start": 0, "limit": 5 }
  }
}
```

In this example, we searched for "milk" at a particular store. The response shows one matching product (a gallon of 2% milk) with its pricing and availability info at that store, and indicates there are 47 total results

(with 5 returned in this page). The `meta.pagination` would allow our server to fetch more pages if needed.

- **Get Product Details by ID** – `GET /products/{id}?filter.locationId=<storeId>`
  Fetches detailed information for a single product specified by its ID or UPC. The `{id}` path can be either a Kroger `productId` or a UPC barcode number – the API will accept either and return the corresponding product data [25]. As with the search, including a `filter.locationId` is required if you need price, inventory, and aisle info for that product [26]. The response format is similar to the search results, but `data` will contain a single product object rather than an array. Use this endpoint when you know the exact product or UPC and want full details. (For example, after getting a list of products from a search, you might call the details endpoint for a specific item to get more info or ensure you have the latest pricing.)

Both product endpoints enforce the daily rate limit of **10,000 calls per day** for the Product API [16]. Your MCP server should cache frequent results when possible (for example, caching popular product info for a short time) to avoid hitting this limit.

Locations API

The **Locations API** provides access to Kroger's store location data, including store addresses, departments, phone numbers, and hours of operation. This is useful for letting users of your application find nearby Kroger family stores (Kroger, Ralphs, Fred Meyer, etc.) and get details about each store. Like Products API, the Locations API is public data – no user login is required, but you still need an access token (client credentials with `product.compact` scope works for this as well).

Base path: `/v1/locations`

Key Endpoints:

- **Search Locations** – `GET /locations?filter.zipCode.near=<zip>&filter.radiusInMiles=<N>&...`
  Retrieves a list of store locations matching the given search criteria. You **must** provide one of the "starting point" filter parameters to specify where to search around [27]: either a zip code, a latitude/longitude, or a combined lat/long string. Only **one** starting point parameter should be used per request (if you provide more than one or none at all, the API returns an error) [27] [28]. The options are:
- `filter.zipCode.near=<5-digit ZIP>` – find stores near the center of this ZIP code [29].
- `filter.latLong.near="<lat>,<lng>"` – find stores near the given latitude/longitude coordinate (pass as a comma-separated string) [30].
- `filter.lat.near=<lat>&filter.lon.near=<lng>` – an alternative way to specify lat/long by separate params [31].
  Additional optional filters:
- `filter.radiusInMiles=<distance>` – Search radius in miles. Default is 10 miles if not specified [32].
- `filter.limit=<number>` – Max number of locations to return (default 10) [32].

- `filter.chain=<string>` – Limit results to a specific store chain. For example, `filter.chain=Kroger` or `Ralphs` or `Fred Meyer`. If not provided, results may include **all** Kroger Co. family stores in the area [33].
- `filter.department=<string>` – Filter to stores that have a specific department. (For example, `filter.department=Pharmacy` might limit to stores with a pharmacy.) If provided, only stores containing that department will be returned.
- **Example Request:**
  `GET /v1/locations?filter.zipCode.near=45202&filter.radiusInMiles=5&filter.limit=5&filter.chain=Kroger`
  This finds up to 5 Kroger stores within a 5-mile radius of the ZIP code 45202.
- **Response:** Returns JSON with a `data` array of location objects. Each location object includes:
  - `locationId` – The store's unique identifier (used in other APIs when referencing this store, e.g. as `filter.locationId` for product pricing).
  - `chain` – The chain name (e.g. `"Kroger"`, `"Ralphs"`).
  - `address` – An object with the store's address (fields: `addressLine1`, `addressLine2`, `city`, `state`, `zipCode`, etc.) [34].
  - `geolocation` – Latitude/longitude coordinates of the store [35].
  - `phone` – Primary phone number for the store [36].
  - `hours` – An object listing the general store hours by day of week (and a flag if it's open 24 hours) [37] [38]. Each day (Monday-Sunday) has `open` and `close` times (likely in local time, often given as strings like `"06:00"` for 6 AM) and an `open24` boolean if it's open all day.
  - `departments` – A list of department objects for that store [39]. Each department has an `departmentId`, `name` (e.g. "Pharmacy", "Deli"), a contact phone, and potentially its own hours (since pharmacy or other services might have different hours than the main store) [40] [41]. The department hours structure is similar (days of week with open/close times) nested under each department.

Sample Response (excerpt):

```
{
  "data": [
    {
      "locationId": "01400943",
      "chain": "Kroger",
      "address": {
        "addressLine1": "123 Example St",
        "city": "Cincinnati",
        "state": "OH",
        "zipCode": "45202"
      },
      "geolocation": { "latitude": 39.1031, "longitude": -84.5120 },
      "phone": "513-555-1234",
      "hours": {
        "monday": { "open": "06:00", "close": "23:00", "open24": false },
        "tuesday": { "open": "06:00", "close": "23:00", "open24": false },
        "...": "...",
```

```
        "sunday": { "open": "07:00", "close": "22:00", "open24": false }
      },
      "departments": [
        {
          "departmentId": "018", "name": "Pharmacy", "phone": "513-555-9876",
          "hours": {
            "monday": { "open": "09:00", "close": "20:00" },
            "sunday": { "open": "10:00", "close": "18:00" },
            "...": "..."
          }
        },
        { "departmentId": "002", "name": "Deli", ... }
      ]
    },
    { ... more stores ... }
  ],
  "meta": {
    "pagination": { "total": 12, "start": 0, "limit": 5 }
  }
}
```

In this example, one store (with ID `01400943`) is returned with its address, hours, and two departments (Pharmacy and Deli) detailed. The `meta.pagination.total` indicates there were 12 stores within 5 miles; we limited to 5 results on this page.

- **Location Details** – `GET /locations/{locationId}`
  Fetches details for a specific store location by its `locationId`. This returns the same information as a single entry from the list above (address, hours, departments, etc.), but for one store. Use this if you already know a store's ID (perhaps from a previous search or because it's stored as a user's preferred store) and want to retrieve or verify all its details. The response JSON will have a single `data` object for that location.

- **Location Existence (Head)** – `HEAD /locations/{locationId}`
  A special endpoint to simply check if a given location ID is valid. It returns HTTP 200 if the location exists, or 404 if not [42]. This is rarely needed in typical integrations (since you would normally get valid IDs from the search or known data), but it's available as a quick check.

The Locations API has a rate limit of **1,600 calls per day per endpoint** [16]. "Per endpoint" means the limit applies separately to each distinct endpoint (e.g., 1,600 for search, 1,600 for detail lookups, etc.). Distributing load across endpoints can help stay within limits [43], but in practice these limits are high enough for most use cases if caching is used. Your MCP server should avoid making redundant location searches by caching results for common queries or store IDs.

Cart API

The **Cart API** allows your application to create and manage a user's online shopping cart on Kroger's platform. This enables use cases like building a shopping list in your app and then pushing those items to the user's Kroger cart for checkout. **All Cart API operations require an authorized user context** – in other words, you must have an access token obtained via the Authorization Code flow (with the `cart.basic:write` scope) for the specific user [3]. The Cart API lets you add items, update item quantities, and remove items from a cart, as well as create or retrieve the cart itself.

Base path: `/v1/carts`

Key Endpoints & Operations: (All require `Authorization: Bearer <user_token>` with `cart.basic:write` scope)

- **Create a Cart** – `POST /carts`
  Creates a new cart for the authenticated user. In Kroger's model, a user can have multiple carts (though typically one active cart per fulfillment context). This endpoint initializes a cart and returns a cart identifier. The request likely requires specifying the store context for the cart. For example, the body may include a `locationId` indicating at which store the cart is intended for pickup/delivery. *(Note: The official docs imply that creating or retrieving a cart is done via the same* `/carts` *endpoint* [44]*. In practice, if an active cart already exists for the user in that context, the API might return it or you may need to explicitly retrieve it. The Cart by ID endpoint described below is used to fetch cart contents.)*

Response: On success, the API returns a cart object (with an `id`). Take note of the cart `id` (a string or GUID) – it will be used for subsequent operations on that cart.

- **Get Cart (by ID)** – `GET /carts/{cartId}`
  Retrieves the contents and details of a specific cart. The cartId would be the identifier obtained from the create operation (or possibly a known active cart ID). The response includes the list of items currently in the cart (with product details, quantities, prices, etc.), and any cart-level info. Each item in the cart typically has: `itemId` (identifying the product variant, same as the UPC or item ID from the Products API), `quantity`, and maybe a nested product summary (name, size, price). If the cart is tied to a specific store or fulfillment method, that info might also be included at the cart level.

- **Add Item to Cart** – `POST /carts/{cartId}/items`
  Adds one or more items to the specified cart. The request body should be a JSON payload specifying the item(s) to add. Typically, you would include the `itemId` (UPC or product variant ID) and `quantity` for each item to add. For example, a minimal payload might look like:

```
{
  "items": [
    {
      "itemId": "0001111045678",
      "quantity": 2
    }
```

```
  ]
}
```

This would add 2 units of the item with ID `0001111045678` (e.g. a gallon of milk) to the cart. You can add multiple different items in one call by including multiple objects in the `items` array. If an item is already in the cart, adding it again may either increase its quantity or result in an error (depending on the API design; typically, it will increment the quantity). The response will usually return the updated cart or the added item details. **Note:** The user must have an active session (access token) and the cart must have been created for the corresponding store context that the item is sold in (i.e., you should ensure the `locationId` of the cart matches where the item is available).

- **Update Item Quantity** – `PUT /carts/{cartId}/items/{itemId}` (or possibly a similar endpoint)
  Updates the quantity of a specific item already in the cart. The request might either use a sub-path with the `itemId` or accept a body with itemId and new quantity. According to the Postman collection, there is an endpoint labeled "Updates item quantity" [45], which suggests a `PUT` on the item resource. For example, `PUT /carts/{cartId}/items` with a body could be used to change quantities, or `PUT /carts/{cartId}/items/{itemId}` with a body `{"quantity": X}`. In any case, this operation lets you change the amount of a particular product in the cart (including setting it to 0, though there is also a remove operation for deleting an item entirely). After updating, the response should reflect the new quantity and updated cart totals.

- **Remove Item** – `DELETE /carts/{cartId}/items/{itemId}`
  Removes an item from the cart. Provide the specific `itemId` (or some item line identifier) in the URL. A successful removal will return either an empty response with a 204 status or the updated cart contents without that item. If you attempt to remove an item not in the cart, you'd get an error (404 or similar).

- **Update Cart (Checkout preferences, etc.)** – `PUT /carts/{cartId}`
  There is mention of an "Update cart" operation [46] which might cover cart-level updates. This could involve actions like changing the fulfillment option on the cart (e.g. switch from pickup to delivery) or associating a loyalty card or coupons to the cart. It might also be used to indicate the cart is ready for checkout or to update other metadata. The exact details aren't fully documented in public, but keep an eye on the official docs if you need this. In many cases, you may not need to explicitly call a cart update unless changing something like fulfillment method. Adding/removing items is the main interaction.

Important: The Cart API is meant to **work on behalf of an authenticated Kroger customer**. That means the user must log in via OAuth and your server uses their token. The cart is actually the same cart that the user would see if they log into Kroger's website or app. For example, if your app adds items to a user's cart, those items will show up in their cart on Kroger.com ready for checkout. Conversely, if they already had items in their cart from before, retrieving the cart via API will show those items. This also implies that **the cart is tied to a specific store/fulfillment context** – typically when the user shops on Kroger's site they choose a store or delivery option first. When using the API, ensure you create the cart with the correct `locationId` (store) and appropriate fulfillment type (if needed) to match the user's intent.

Example Workflow: A typical usage in an MCP scenario might be: 1. User logs into your app and authorizes Kroger access (you obtain an access token with `cart.basic:write`). 2. Your server calls `POST /carts` (with `locationId`) to create a new cart for that user (if they don't have one active). 3. For each item the user wants to add (perhaps selected in your app), call `POST /carts/{cartId}/items` with the item's ID and quantity. 4. If needed, adjust quantities with `PUT` or remove items with `DELETE`. 5. Optionally, fetch the cart `GET /carts/{cartId}` to show a summary to the user (e.g. list of items, total cost if provided, etc.). 6. The user can then be directed to Kroger's own checkout to complete payment (the API itself doesn't process payments – the typical flow is to hand off to Kroger for the actual purchase).

Response Data: The Cart API responses will include cart and item structures. A cart object might include fields like `id`, `locationId`, `lastModifiedDate`, and an array of `items`. Each item in the cart typically has: - `itemId` (the product's identifier), - `quantity`, - possibly an `itemPrice` (price * quantity), - and a nested `product` detail (name, size, price each, etc.).

For example, an item might look like:

```
{
  "itemId": "0001111045678",
  "quantity": 2,
  "price": 2.99,
  "productName": "Kroger 2% Reduced Fat Milk 1 Gallon"
}
```

In the official API, the structure could be more nested (with a `product` object containing the name, etc.). Always refer to the actual API response for exact format. After adding items, you may also see a cart-level summary like total item count, and maybe a running total cost (though Kroger might also calculate final pricing during checkout).

The Cart API has a daily limit of **5,000 calls per day** [16]. Each operation (add, update, etc.) counts against this. Be mindful to batch operations when possible (e.g. adding multiple items in one POST) to reduce call volume, and avoid constantly polling the cart.

Note: Only **write operations** (adding/updating) are allowed by third-party apps – you **cannot** actually submit the cart for checkout via the public API (checkout involves payment and is handled on Kroger's site). Your MCP server's role is to assemble the cart; the user will complete the purchase through Kroger's normal checkout process (for example, you might provide them a deep link or prompt to open Kroger's app/site to finish the order).

Identity (Loyalty/Profile) API

The **Identity API** provides access to basic profile information of an authenticated Kroger customer. For third-party developers, the most relevant piece of data here is the customer's **loyalty card number** (Plus Card number), which can be retrieved if the user has authorized the `profile.compact` scope. This can be useful for linking the user's loyalty account or displaying their number for reward point tracking.

Base path: `/v1/identity/profile`

Key Endpoints:

- **Get Loyalty Card Number** – `GET /identity/profile/loyalty`
  Returns the loyalty information for the authenticated user. According to the documentation, this will return the customer's loyalty card number (the number on their Kroger Plus card) [47]. The user **must** be authenticated with OAuth2 using the Authorization Code flow (you cannot get this with just client credentials) [48]. The expected response would include the loyalty account number (and possibly type of account). For example:

  ```
  {
    "data": {
      "loyaltyCardNumber": "4123456789012"
    }
  }
  ```

  (The exact field name could be `loyaltyCardNumber` or similar, containing the 12-digit number. If the user has no loyalty card linked, the response might be empty or a 404.)

- **Get Profile ID** – `GET /identity/profile`
  Returns a minimal profile object, likely containing the user's internal Kroger profile ID (a UUID or numeric ID associated with their account). The public documentation notes that the Identity API (Public) "provides access to the profile ID of an authenticated customer" [49]. This might not be broadly useful unless you need to store a reference to the user's Kroger profile. In many cases, you may not need to call this at all, or the act of retrieving loyalty info might implicitly confirm the user's identity. But if needed, this endpoint could be used simply to verify that the user's token is valid and get their ID.

- **Check if Profile Exists** – `HEAD /identity/profile?email=<email>` or `GET /identity/profile/invoke/exists?email=<email>`
  These endpoints allow you to determine if a given email is associated with an existing Kroger.com account [50] [51]. It's a way to check, for example, if a user needs to create a new Kroger account or not. The `HEAD` request returns 200 or 404, and the `invoke/exists` GET returns a boolean or 404/401 depending on the scenario. This is more of an edge-case utility (likely used during onboarding flows). In an MCP server, you might not need this unless you are helping users link accounts and want to check their email. Remember that for privacy, this will only work for authorized contexts (and Kroger might restrict its use).

Using the Identity API requires the `profile.compact` scope on the user's token [3]. The daily rate limit for Identity API calls is **5,000 per day** [16] (shared across its endpoints). This is usually plenty, as you typically only call these endpoints once per user session (e.g., to fetch loyalty number after login).

Error Handling, Throttling, and Retry Logic

When integrating any API into your server, robust error handling is crucial. The Kroger API communicates errors via HTTP status codes and JSON error responses. Your MCP server should anticipate common error scenarios and implement appropriate retry or user feedback logic:

- **Invalid Requests (400)** – If your request is malformed or missing required parameters, the API will return HTTP 400 with an error payload. For example, a 400 Bad Request might occur if you call the product search without any of the required filters. The error response typically looks like:

```
{
  "timestamp": 1609459200000,
  "code": "InvalidRequest",
  "reason": "At least one of filter.term or filter.productId must be
provided"
}
```

  The `code` and `reason` fields explain what went wrong [52]. In this case, the client should adjust the request (e.g., add a search term).

- **Unauthorized (401)** – If your access token is missing, invalid, or expired, you'll get a 401 Unauthorized. The error body for OAuth-related issues often follows OAuth2 standard fields, for example:

```
{
  "errors": {
    "error": "invalid_token",
    "error_description": "The access token expired"
  }
}
```

  [53]. In this scenario, your server should initiate the token refresh flow (if a refresh token is available) or ask the user to re-authenticate if needed. For client credential tokens, just request a new token using the Client ID/Secret (and consider caching token lifetimes to refresh proactively).

- **Forbidden (403)** – If your token is valid but does not have the required scope for that endpoint, or if the application has not been granted permission, a 403 Forbidden may be returned. For example, trying to use a client token (no user) to access the Cart API will result in 403 since that operation requires a user authorization.

- **Not Found (404)** – You may get 404 for endpoints that include an ID when the resource doesn't exist. E.g., `GET /products/123` with an unknown ID, or `GET /locations/99999` for a non-existent store. Also, removing an item not in cart might return 404. For HEAD or exists checks, 404 indicates a negative (e.g., profile or location not found).

- **Rate Limit Exceeded (429)** – If you exceed the allowed number of calls, the API will likely return HTTP 429 Too Many Requests. The response may include a `Retry-After` header indicating when you can resume, though if not provided, assume until the next day for daily limits. The error payload may have a code like `"RateLimitExceeded"` or a similar reason. **Retry logic:** In case of 429, your server should back off and not immediately retry all requests. Implement logic to queue or delay calls until the rate limit window resets. It's wise to build in monitoring/logging for how many calls you're making and ensure you stay under the limits (10k/day for products, 5k/day for cart, etc.). If you approach limits, consider caching or slowing down non-critical requests.

- **Server Errors (500)** – Unexpected server-side issues at Kroger (rare, but possible) will return 5xx codes. The response may contain an `errors` object with a generic `reason` and a `code` [54]. For example, a 500 might look like:

```
{
  "errors": {
    "reason": "Internal error processing request",
    "code": "InternalError",
    "timestamp": 1690000000000
  }
}
```

  For 500-series errors (500, 502, etc.), implement a **retry with backoff**. That is, wait a short period (a few seconds) and retry the request, perhaps up to a couple of times. Often these are transient. But do not retry endlessly, and log these occurrences. If a 500 persists, it might indicate an issue with Kroger's API or something unexpected with your request.

Error Response Structure: Note that Kroger's error format sometimes uses a top-level `"errors"` key (especially for OAuth or certain calls) and other times a top-level object with `code` / `reason`. Be prepared to handle both. The general pattern is that **business logic errors** (bad requests, not found, etc.) yield a response with `code` and `reason` [52], whereas **authorization errors** might appear under an `errors` object with OAuth-style fields [53]. In all cases, returning an informative message to the user (or calling service) is helpful. For instance, if adding to cart fails because the item is not sold at that store, the API might respond with a specific reason which you can relay or handle.

Logging: Your MCP server should log error responses with context (but avoid logging sensitive data) to assist in debugging issues. This is especially important during initial development and testing with the certification environment.

Using the OpenAPI (Swagger) Specification

Kroger provides official **OpenAPI (Swagger) specifications** for their APIs, which can greatly aid integration. Each API (Products, Locations, Cart, Identity, Auth, etc.) has a downloadable OpenAPI spec (in JSON or YAML) available on the developer portal [55]. For example, on the **Products API** documentation page, you'll find a link "Download OpenAPI specification" [55]. These specs define all endpoints, parameters, and response schemas.

How to use them: You can download the OpenAPI specs and use them to generate client code or documentation:

- **Client SDK Generation:** Many tools (Swagger Codegen, OpenAPI Generator) can take a spec and produce client libraries in various languages. You could generate a Java, Python, or JavaScript client for Kroger's API which might save you from writing a lot of boilerplate HTTP calls. (Keep in mind you'll still need to handle authentication, but some specs include the auth endpoints too.)
- **Interactive Docs:** You can load the spec into Swagger UI or Postman. In fact, Kroger's public Postman collection is likely derived from the same specs. Postman allows importing an OpenAPI file to create a collection of requests. This is very useful for exploration and initial testing.
- **API Exploration:** Using Swagger UI (either locally or via an online viewer) with Kroger's spec lets you see all operations, required parameters, models, and even try out calls (you'll need to plug in a token). This complements the written documentation.

For example, the **Authorization Endpoints** spec (v1.0.13) and others are documented via Redocly on Kroger's site [56]. The OpenAPI files will list endpoints like `/products`, `/locations` etc., with parameter details and response object definitions. Incorporating these into your development workflow can ensure you don't miss any fields or mis-handle request formatting.

In summary, the OpenAPI specs are a valuable resource for **keeping your implementation aligned with Kroger's API contract**. Whenever Kroger updates an API (they version their APIs as seen in the spec versions like 1.2.3, 1.2.4 etc.), you should check for updated specs and adjust your integration if needed.

SDKs, Libraries, and Tools

Currently, Kroger does not provide an official SDK, but the developer community has created some libraries to simplify using the Kroger API:

- **Unofficial Kroger API Python Client:** There is a community-maintained Python library that wraps all the endpoints and handles OAuth2 token management for you [57]. This library demonstrates how to obtain tokens, refresh them, and call the product, location, cart, and identity APIs with simple methods. For example, using that library, adding an item to a cart is as easy as calling a function instead of manually crafting HTTP calls. While unofficial, it can serve as a reference or quickstart for your own integration (especially for understanding token flow). Always ensure you understand what third-party libraries are doing, especially with handling your credentials.

- **Postman Collection:** Kroger has an official Postman workspace with all the Partner API endpoints (the same ones we've discussed) [58]. You can fork this collection and use Postman to test requests. Postman's environment templates provided by Kroger allow you to just plug in your client ID, secret, and generate tokens via the Authorization helper. This is extremely useful for exploring the API manually and verifying behavior before coding it into your server.

- **API Explorer:** On Kroger's developer portal (when logged in), you might find interactive API docs or an explorer (some portals allow making test calls directly). Given the portal uses Redocly for docs, they might have a "Try it" feature if you authenticate.

- **Community Projects:** Aside from Python, you might find community examples in other languages (e.g., a Node.js wrapper or some how-to articles). For instance, a community member demonstrated usage with PHP Guzzle [59]. Additionally, check Kroger's developer blog or forums (if any) for shared snippets.

When choosing to use an SDK or library, consider how actively maintained it is. The safest route for a production MCP server is often to write a thin integration layer yourself using the OpenAPI spec as a guide, so you have full control. But during development, these tools can accelerate learning and prototyping.

Best Practices for Server Setup and Secure Integration

Finally, when building your MCP server around the Kroger API, keep these best practices in mind to ensure a secure, reliable, and maintainable integration:

- **Use the Certification Environment for Testing:** Kroger offers a **certification (sandbox) environment** at `api-ce.kroger.com` for testing your integration without affecting production data [60] [1]. Use this during development and QA. Note that the certification environment might have test data (e.g., fake products or limited store info) and it might require separate client credentials or app registration. You cannot mix environments for a given set of credentials (an app is locked to either prod or cert) [1], so decide early on and get a cert environment app if needed.

- **Secure Storage of Keys:** Treat your Client ID and Secret like passwords. Store them in secure configuration on your server (environment variables, secure config files, or a secrets manager). **Do not embed them in mobile apps or front-end code.** All token exchanges should happen on your server side over HTTPS to Kroger. This ensures the secret is never exposed. Also, do not log the secret or access tokens. If you need to log API calls, mask out the `Authorization` header for security.

- **Token Management:** Implement a robust token management strategy. This includes:

- Caching tokens in memory or a database so you don't request a new token for every single API call. Requesting a token is a separate call that also counts against rate limits (though Kroger's token endpoint might not be limited in the same way, it's just good practice to reuse tokens until they expire).
- Tracking expiration times (the `expires_in` value). For example, if a token lasts 30 minutes, your server can refresh it at the 25-minute mark proactively.
- Using refresh tokens for user auth flows: store refresh tokens securely (they are essentially long-lived credentials) – possibly encrypted at rest – and use them to get new access tokens without user intervention. Build automatic retry logic so if an API call returns 401, your code attempts one refresh cycle and then retries the API call with the new token [61] [62].

- Clearing or rotating tokens on logout: if a user disconnects their Kroger account from your app, discard their tokens.

- **Error and Retry Strategy:** As discussed, implement intelligent retries with backoff for transient failures. For user-facing actions (like adding to cart), ensure you handle API errors gracefully – e.g., if adding an item fails because it's out of stock, inform the user rather than simply showing a generic

error. Logging is your friend – log failures with enough detail to debug (HTTP status, error code from response), but avoid logging sensitive PII.

- **Rate Limit Management:** Even though daily quotas are high, if your MCP server serves many users or high traffic, you should keep an eye on how close you get to limits. Use in-memory counters or monitoring to track calls. If nearing a limit, you might temporarily disable some non-critical calls (for instance, if you have a background job syncing some data, you could pause it). For spikes in usage, consider contacting Kroger for higher limits or optimizations. Also, design your system such that if Kroger's API quota is reached, your application degrades gracefully (maybe certain features become read-only or queue tasks for later).

- **API Key Rotation:** Kroger's developer portal may allow you to rotate your client secret. It's good practice to have a process for updating the secret in your server configuration periodically (especially if you suspect it may have been exposed). If rotating, update your server and test the new credentials in the cert environment first. Also, be mindful that you cannot change an app's environment (cert vs prod) or its basic details after creation [1] – if you need a fresh start or new scopes, you might create a new application in the portal. Plan and document the process for regenerating secrets and updating them to avoid downtime.

- **Data Handling and Caching:** Respect Kroger's acceptable use policies [63]. Do not store sensitive customer data unnecessarily. If you cache product or location data to improve performance, set reasonable TTLs (Kroger updates prices and stock regularly, so don't serve stale data for too long – e.g., cache for a few hours at most or refresh daily). For personal data (loyalty numbers, etc.), store only what's needed and comply with privacy requirements (e.g., if a user disconnects, delete their data).

- **Multi-Channel Considerations:** Since you are building a **multi-channel** server, likely serving web, mobile, etc., ensure that you centralize the Kroger integration in your backend. All channels should go through your server for Kroger API calls – this way you can consistently enforce security (no client has direct access), and you can aggregate calls (reducing duplicate searches, etc.). For example, if both your web and mobile app call for the "weekly specials" and you have to hit the products API for that, have your server call Kroger once and distribute the data to clients, possibly caching it briefly. This reduces load on Kroger's API and on your system.

- **Testing and Monitoring:** Use the certification environment to simulate user flows. Write unit/integration tests for scenarios like *search products -> add to cart -> verify cart content*. Monitor your production integration for any changes in API behavior (Kroger might update an API version or deprecate something; staying active in their developer community will help you catch changes). Monitor response times as well; if Kroger's API latency increases, you might need to adjust timeouts or user feedback (e.g., a loading spinner while waiting for products).

By adhering to these practices, you'll create a robust MCP server that leverages the **full Kroger API** while maintaining security and performance. This concludes the deep dive guide – you should now have a clear roadmap for integrating product search, store info, cart building, and loyalty data from Kroger into your multi-channel application. Happy coding, and may your users enjoy a seamless grocery shopping experience!

Sources:

- Kroger Developer Documentation – *API Reference and Guides* [17] [26] [64] [28] [47]
- Kroger API Rate Limits (Products, Locations, Cart, Identity) [43]
- Kroger OAuth2 Usage – *Understanding OAuth2 and Token Requirements* [2] [5]
- Unofficial Kroger API Python Client – *Examples and Scopes* [4] [8] [57]
- Kroger Postman Collection – *Endpoint examples and error formats* [52] [53]

---

[1] Applications - Kroger Developers
https://developer-ce.kroger.com/documentation/support/website-help/applications

[2] Understanding OAuth2 - Kroger Developers
https://developer-ce.kroger.com/documentation/public/security/guides-oauth

[3] [4] [8] [16] [43] [57] [61] [62] [63] GitHub - CupOfOwls/kroger-api: A comprehensive Python client library for the Kroger Public API, featuring robust token management, comprehensive examples, and easy-to-use interfaces for all available endpoints.
https://github.com/CupOfOwls/kroger-api

[5] Service to Service - Kroger Developers
https://developer-ce.kroger.com/documentation/partner/service-to-service?beta=true

[6] [7] [9] [10] KrogerController.java
https://github.com/mattbutcher-swe/shop/blob/e045fd58f8711a332e847cea96600bfea8db5e8e/backend/src/main/java/github/com/mattbutcher_swe/shop_backend/controllers/KrogerController.java

[11] [12] [13] [14] [15] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [47] [48] [50] [51] [52] [53] [54] [64] Kroger Partner APIs | Documentation | Postman API Network
https://www.postman.com/kroger/the-kroger-co-s-public-workspace/documentation/nryx3kn/kroger-partner-apis?entity=request-4826685-08910009-1c49-48c3-9af2-c6d41adaaa0d

[44] Cart Tutorial - Kroger Developers
https://developer-ce.kroger.com/documentation/api-products/partner/carts/tutorial?beta=true

[45] [46] [58] Add to cart | Kroger Partner APIs | Postman API Network
https://www.postman.com/kroger/the-kroger-co-s-public-workspace/request/7uyizn9/add-to-cart

[49] Identity API - Kroger Developers
https://developer.kroger.com/reference/api/identity-api-public

[55] Products API - Kroger Developers
https://developer-ce.kroger.com/api-products/api/product-api-partner

[56] Authorization Endpoints - APIs | Kroger Developers
https://developer.kroger.com/api-products/api/authorization-endpoints-partner

[59] Kroger API Examples How To [PHP Guzzle Async]
https://webprogrammer.io/kroger-api-examples/

[60] API Basics - Kroger Developers
https://developer-ce.kroger.com/documentation/public/getting-started/apis