Chapter 11

RESTful Web Design

What is in This Chapter ?

This chapter explains how to follow **REST**ful web design guidelines. **REST**ful web design is like fine dining compared to fast food. Both will “feed” you data, but in very different ways. A fast-food **API** might give you what you need quickly, but it’s usually messy, inconsistent, and not designed for long-term use. **REST**, on the other hand, is like *fine dining*: every course (resource) is well-presented, the menu (**URL**s) is organized and predictable, and the service (**HTTP** methods) follows clear rules. The experience is structured, scalable, and meant to be appreciated … not just consumed in a hurry.

11.1 RESTful APIs

As we have seen in this course so far, web applications are not limited to delivering static **HTML** pages. Instead, they often need to send and receive data between the browser (or another client) and the server in real time. This is where **web APIs** come in.

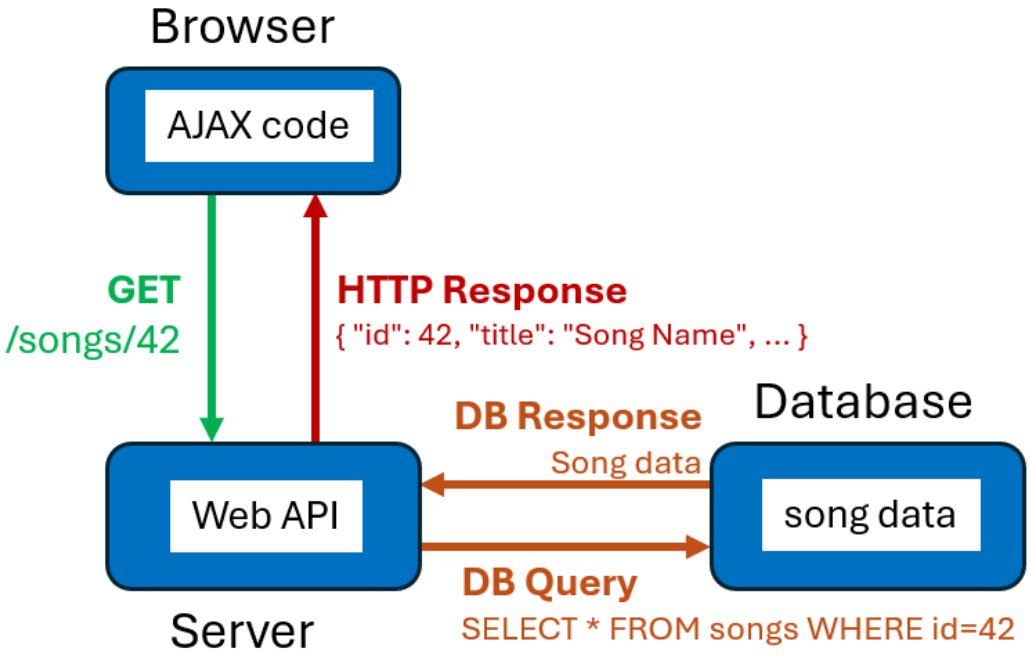
> ***A Web API is an interface that allows different software applications to communicate and exchange data over the web, typically using HTTP.***

It provides a clearly-defined way for clients (e.g., web browsers, mobile apps, smartwatch apps, other servers) to request, modify, and work with application data. To design these **API**s effectively, developers often use ***REST*** (**RE**presentational **S**tate **T**ransfer).

REST is not a protocol, format, or strict rulebook … it is really just some guiding principles to follow when designing our app. A ***RESTful* API** is one that follows three main principles of how web components should interact over the internet:

- **Stateless Communication** - every request is independent and contains all the information the server needs.
- **Resource-based Structure** - data and functionality are organized into distinct resources, each with a unique **URL**.
- **Use of HTTP Methods** - standard **HTTP** verbs (**GET**, **POST**, **PUT**, **DELETE**, etc.) indicate the desired action on a resource.

Consider, for example, what happens when a client requests some song data from a server. Look at the diagram below and see if you can verify the three principles mentioned above:



In the above diagram, the resource is the song data. The **URL** supplied by the browser specifies the song **ID** indicating the unique resource. Notice that all the information it needs is in the **GET** and the supplied **URL**, since it indicates: "Get me this exact song". Of course, for this to work, the client must know what it is able to ask for.

A **RESTful** web **API** presents all the server's available resources in a clear and structured way, making it obvious what clients can access and how to interact with them. Each resource is identified by a unique **URL** and can be manipulated using standard **HTTP** methods such as **GET**, **POST**, **PUT**, and **DELETE**. The resources themselves can take various forms, such as:

- **JSON** – for structured data exchange
- **HTML** – for web pages
- Plain text or **XML** – for simpler or legacy data formats
- Images, videos, or other files – for media content

While some resources may require authentication or permissions to access certain data, the **API**'s structure and available endpoints are fully visible, so clients can see what is possible, even if they cannot access everything. This clarity helps developers understand the system, know which actions are allowed, and integrate with it reliably.

In a **RESTful API**, data is presented as resources that can be created, retrieved, updated, or deleted. These operations are collectively known as ***CRUD***:

- **C**reate a resource (i.e., **POST** request)
- **R**ead a resource (i.e., **GET** request)
- **U**pdate a resource (i.e., **PUT** or **PATCH** request)
- **D**elete a resource (i.e., **DELETE** request)

For example, in a **RESTful API** for a site with songs, we might see requests/responses like this:

Method	URL	Resource Interaction	Response
GET	/songs	get a list of all songs	**200** OK
GET	/songs?genre=pop	get all songs in the pop genre	**200** OK
GET	/songs?artist=Taylor+Swift	get all songs by artist "Taylor Swift"	**200** OK
GET	/songs?genre=rock&year=2020	get all rock songs from 2020	**200** OK
GET	/songs/234	get song with ID 234	**200** OK
GET	/songs/9999999	get non-existing song with ID 9999999	**404** Not Found
GET	/songs/234/lyrics	get lyrics for song 234	**200** OK
GET	/songs/234/comments	get comments for song 234	**200** OK
POST	/songs	create a new song	**201** Created
PUT	/songs/234	update song with ID 234	**204** No Content
DELETE	/songs/234	delete song with ID 234	**204** No Content
POST	/songs/234/comments	add a comment to song 234	**201** Created
DELETE	/songs/234/comments/10	delete comment 10 on song 234	**204** No Content
GET	/songs/top?limit=10	get top 10 songs	**200** OK
GET	/artists/45/songs	get all songs by artist with ID 45	**200** OK

By looking at the request method and **URL** in the above table, the resulting resource interaction and response is somewhat intuitive. It shows that by following **REST** principles, **API**s become predictable, easier to understand, and simpler to integrate across platforms. This is why **REST** is commonly used in web service design.

A truly **REST**ful **API** is designed according to six architectural constraints defined by Roy Fielding (https://ics.uci.edu/~fielding/pubs/dissertation/top.htm). These are:

1. **Client–Server:**

   In **REST**, the client and server have separate responsibilities, so changes on one side don't directly affect the other.

   Server: Stores the data, rules, and logic, and provides information (i.e., representations of resources) in response to client requests.

   Client: Makes requests to the server and does something useful with the response.

   Ideally, the client and server see each other as "black boxes" where each knows the **API** (the interface) but not the internal implementation of the other. In some cases, the client may even learn about the **API** dynamically from the server.

   This separation provides flexibility:

   - Different types of clients (e.g., browser, mobile app, bots) can interact with the same server.
   - The server can change how it stores or manages data without affecting clients.
   - New components or services can be added without disrupting the system.

2. **Stateless:**

   In **REST**, each request from the client must include all the information the server needs to process it and respond. The server does not keep any memory of previous requests or client session state … it treats every request as independent. This makes it easy to add more servers to handle additional requests, which improves scalability.

   For example, imagine a product search: **GET /products?type=book**

   The server might return only the first **10** results. If the client wants more, it must include all necessary information to get the next set, such as one of these:

   ```
   GET /products?type=book&start=25
   GET /products?type=book&page=2
   GET /products?type=book&num=25&offset=25
   ```

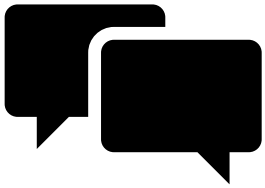
   We can see that each request is independent in that the server does not remember what the client requested previously.

   In contrast, non-**REST**ful systems often rely on server-side session state. The server remembers what the client was doing, allowing requests like:

```
GET /products?type=book
GET /nextPage
GET /nextPage
```

Here, the server keeps track of the client's progress, which violates statelessness. If that server crashes, the client may lose its session and have to start over. A stateless system, on the other hand, is more robust and scalable. Since each request contains all the information the server needs, any server can handle any request. If one server fails, another can take over without interrupting the client's interactions.

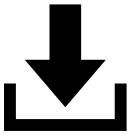
Even for something like a back-and-forth chat session, **REST** can remain stateless, despite the existence of chat history. Each request from the client must include all the information the server needs to process it, such as an authentication token, the message content, or the chat room ID.



But the chat history itself is data, not session state. The server can store messages in a database and return past messages on request. This does not violate statelessness because the server is not keeping any temporary memory of the client's ongoing session. Each request is handled independently, so if the client disconnects and reconnects, the server can still process requests correctly.

3. **Cacheable:**

In **REST**, the server explicitly indicates whether a response can be cached. This is done using **HTTP** headers (like `Cache-Control` or `Expires`). If a response is cacheable, the server can also specify how long it remains valid.



When a client has a valid cached copy, it can reuse it instead of making a new request, which reduces network traffic and server load. This makes the system more scalable and efficient.

As an example, consider an image-hosting website. The server stores both images and metadata (e.g., filename, upload date, tags, or description). While metadata may change, the image files themselves usually do not. The server can mark the images as cacheable for longer periods, while metadata may have shorter cache times or no caching at all. This way, clients and intermediaries can reuse large, unchanging resources efficiently.

4. **Uniform Interface:**

A **REST**ful system uses a uniform interface to ensure consistent communication between clients and servers. This makes **API**s predictable, easy to understand, and interoperable across different systems. The uniform interface has several key aspects:

(a) Identification of Resources

REST is centered around resources, which are any entities that can be named (e.g., images, user profiles, orders, or real-time data streams). While each resource may be dynamically generated by the server, each resource must be uniquely identifiable by a **URL** that remains constant for its lifetime.

For example …

http://myapi.com/products ← collection of products
http://myapi.com/products/28812 ← a specific product
http://myapi.com/products/28812/reviews ← reviews for that product

URLs usually represent nouns, not verbs. **API**s that use verbs in **URL**s (e.g., /removeProduct or /updateAccount) are generally not **REST**ful).

(b) Manipulation of resources through representations:

In **REST**, the client interacts with *information about* the resource, not the resource itself or its storage mechanism. This information, often formatted as **JSON**, contains all the data needed to understand and work with it. The server handles the actual storage and management of the resource (e.g., by using a database, files, or even other servers) but the client doesn't need to know how this is done. This separation allows the server to change its internal implementation without affecting the client.

For example, suppose the server stores product information in a database. Internally, a product record might look like this →

```
id: 28812
name: "Wireless Mouse"
price: 29.99
stock: 12
supplierId: 451
barCode: "WM-122-XY"
```

The client (e.g., a web app or mobile app) doesn't need to know how the server stores this data. Instead, the client requests the product and receives a **representation** of it, usually in **JSON** →

```
{
    "id": 28812,
    "name": "Wireless Mouse",
    "price": 29.99,
    "stock": 12
}
```

The client now has all the information needed to display the product or let a user interact with it. The server may later change its internal storage (e.g., move from a database to a file system), but the client still receives the same **JSON** format, so its code doesn't break.

(c) Self-descriptive messages:

In a **REST**ful system, every request and response should contain enough information to be understood on its own, without relying on previous messages. Also, each message should contain a data type, which tells the receiver what kind of data it is receiving (e.g., `"Content-Type"` in **HTTP**).

RESTful systems should also follow the formal meaning of the **HTTP** verbs: **GET** to retrieve, **POST** to create, **PUT** to update and **DELETE** to remove. Because these verbs have well-known meanings, clients and servers can interact consistently, which contributes to a uniform interface.

(d) Hypermedia as the Engine of Application State (HATEOAS):

In **REST**, hypermedia links in the server's response act like signposts. The client doesn't need to "know" all **URL**s in advance; it can simply follow the links provided by the server. Starting from an initial **URL**, the client can discover and navigate other resources dynamically using these hyperlinks.

This approach allows clients to explore relationships (e.g., how a product is linked to its reviews or manufacturer), discover related resources (e.g., similar products), and interact with the system without hardcoding any **URL**s.

For example, a client requests the first page of products like this: **GET /products**
The server can respond with **JSON** including products and links:

```
{
  "products": [
    { "id": 101,
      "name": "Wireless Mouse",
      "price": 29.99,
      "links": {
           "reviews": "/products/101/reviews",
           "similar": "/products?category=accessories" }},
    { "id": 102,
      "name": "Mechanical Keyboard",
      "price": 79.99,
      "links": {
           "reviews": "/products/102/reviews",
           "similar": "/products?category=keyboards" }}],
  "links": {
     "nextPage": "/products?start=10" }
}
```

To see reviews for the mouse, the client follows **/products/101/reviews**, to see similar products it follows **/products?category=accessories**, to see the next page of products it follows **/products?start=10** etc..

For fully **REST**ful systems that implement **HATEOAS**, the client never hardcodes **URL**s … it discovers all actions dynamically through the links provided in the responses. However, in practice many so-called "**REST API**s" do not fully use hypermedia. The reality is that many **API**s still expect the client to know **URL**s in advance and don't provide hypermedia links. These are technically not fully **REST**ful, even if they use **HTTP** verbs and **URL**s.

5. Layered System:

In **REST**, the system can be composed of multiple layers of servers or services, but the client interacts with the system as a whole and never needs to know how many layers exist or how they are structured, it just sends requests and receives responses. Each layer only communicates with the layers directly above or below it. This separation of concerns makes the system more modular, maintainable, and scalable.

As an analogy of not needing to know the layers … just think of sending a FedEx package. We (i.e., client) drop it off at the FedEx counter (i.e., web server). The staff handles routing our package through storage, sorting, and shipping (i.e., other layers). We don't need to know what happens behind the scenes as long as our package gets delivered.

In a web app, we may have layers like: **Clients ←→ Servers ←→ Databases**

Additional layers can be added without affecting the client or other layers:

- **Caching layer:** speeds up responses by storing frequently requested data.
- **Authentication/Authorization layer:** checks security credentials.
- **Load balancer layer:** distributes requests across multiple servers.

This enforces separation of concerns within the system and increases modularity.

6. **Code on Demand (optional):**

In most **REST** interactions, the server sends static representations of resources (e.g., **HTML**, **JSON**, **XML**) that the client already knows how to handle. However, the server may optionally send executable code (such as **JavaScript**) that extends or modifies the client's behavior on the fly. This is useful when the server wants to give the client new capabilities without requiring the client application to be updated manually. For example:

- A photo-sharing site adds a brand new type of image filter, but the current client app doesn't know how to create that effect. The server can send **JavaScript** code with the image data that teaches the client how to render the new filter.

- An online form might receive **JavaScript** that validates new input formats without the client needing an update.

This approach allows the server to deliver new features instantly, keeping the system flexible and reducing deployment friction … though it's used sparingly due to security and caching concerns.

Not all **API**s that call themselves "**REST**ful" follow every one of these constraints. In general, the more constraints an **API** adheres to, the closer it is to the full **REST** architecture, which can improve scalability, maintainability, and clarity.

11.2 Designing RESTful APIs

A **REST**ful **API** is built around the idea of resources (i.e., entities in our system that we want to expose to clients). Designing one well means deciding what those resources are, how they are identified, and how clients can interact with them.

Here are some tips towards designing a **REST**ful **API**:

Identify Resources:

A resource is anything our **API** manages (e.g., a user, a product, an order, etc.) So, our first step is to come up with a list of the main "things" our system works with. These can be:

- **real-world objects** (e.g., `users`, `products`, `vehicles`)
- **digital items** (e.g., `images`, `reviews`, `songs`)
- **abstract concepts** (e.g., `orders`, `sessions` or `transactions`)
- **system resources** (e.g., `settings`, `logs`, `metrics`)

Next, we should identify the attributes for each resource:

users id name email phone address purchases reviews interests	**images** id url filename format size uploadedBy	**orders** id userId items status totalPrice createdAt	**settings** id name value updatedAt
products id name description price category images reviews stockQuantity	**reviews** id productId userId rating comment date	**sessions** id userId createdAt expiresAt ipAddress	**logs** id timestamp level message source
vehicles id make model year color VIN Mileage	**songs** id title artist album duration releaseDate genre	**transactions** id userId amount currency status timestamp	**metrics** id name value recordedAt

Then we need to come up with our ***endpoints*** … which is a specific **URL** + **HTTP** method that represents one resource or a set of related resources…

Use Consistent, Predictable Naming:

When choosing our endpoints, we should follow these guidelines:

- **Use plural nouns** in route paths to show that the endpoint works with a collection and returns multiple items, not just a single one:

  /products = the entire collection of products
  /users = the entire collection of users
  /reviews = the entire collection of users

- Append a **unique identifier** to the collection path for a single resource to make it obvious that the endpoint returns a single item:

  /products/35452 = the product with ID **35452**
  /users/18 = the user with ID **18**
  /reviews/164 = the review with ID **164**

- Represent **parent–child relationships** directly in the **URL** to improve discoverability and keep related data grouped together:

  /products/35452/reviews = all reviews for product **35352**
  /users/18/orders = all orders for user **18**

- Use **lowercase letters and hyphens** for multi-word resource names because hyphens are more **URL**-friendly, easier to read, and work better with search engines:

  /user-profiles … not **/UserProfiles** nor **/user_profiles**

- **Avoid verbs** in resource names. Let the **HTTP** method describe the action, not the path.

  POST /orders (create new order) … not … **/createOrder**
  DELETE /orders/37 (delete order **37**) … not … **/deleteOrder/37**

- Map **HTTP** verbs **to CRUD** operations:

  GET = retrieve one or more resources
  POST = add a new resource to a collection
  PUT = replace an existing resource
  PATCH = partially update an existing resource
  DELETE = remove a resource

- Use **query parameters** for filtering, sorting, and searching. The path should identify the resource, while the query parameters control how the results are returned:

  /products = get all products
  /products?name=paint = get products with **paint** in the name
  /products?type=book = get products that have a type of **book**
  /products?name=paint&type=book = get books with **paint** in the name

Avoid putting filter logic into the path:

/products?category=electronics&sort=price

instead of … **/products/electronics/sort/price**

- **Keep URIs stable** and version when necessary. Once published, treat a resource path as permanent. If breaking changes are needed, version at the root:

  /v1/products becomes **/v2/products** if a new version is needed

Use Pagination:

When a collection contains hundreds, thousands, or even millions of resources, returning the entire set in a single response is inefficient and unnecessary. Instead, the server should return a small subset (i.e., a *page* of results) for each request. This approach, called ***pagination***, reduces bandwidth usage and improves performance.

Pagination is often implemented using query parameters in the **GET** request that can specify:

- The number of resources to return (e.g., **limit** or **pageSize**)
- The page number to retrieve (e.g., **page**)
- Or, in some **API**s, a start and end range of results (e.g., **start**, **end**).

Here are some typical examples:

/products	get all **products** (use default pagination (e.g., first **20** items))
/products?page=3	retrieve the **3rd** page of **products**
/products?limit=20	return only **20 products** per page
/products?offset=40&limit=20	skip the first **40 products**, then return the next **20**
/products?start=41&end=60	return **products** with positions **41** through **60**
/products?type=book	get all **products** of type **book** (default pagination)
/products?type=book&limit=10	get first **10 book** products
/products?type=book&page=2&limit=10	get **2nd** page of **books**, **10** per page (items **11**–**20**)
/products?type=book&offset=20&limit=10	get **books** starting from item **21**, next **10** items
/products?type=book&sort=price-asc&limit=5	get first **5 books** sorted by **ascending price**

In a query string, the plus sign (+) usually represents a space between words. However, the exact meaning can vary depending on how the server handles it. For example:

/products?type=book&search=fidget+toy&page=1&limit=5

This may either search for books with **"fidget toy"** in the title or description and give the first **5** results. Alternatively, it may search for books with both **"fidget"** and **"toy"** in the title or description. Most modern **API**s use the second approach.

Use Appropriate Response Codes:

Using the right response codes helps clients quickly understand what happened (i.e., whether the request worked, needs adjustment, or failed). The most common codes are grouped below by category, along with their typical meaning and when to use them:

Informational:

100	Continue	Client may continue with request
101	Switching Protocols	Server is switching protocols

Success:

200	OK	Request succeeded
201	Created	Resource successfully created
202	Accepted	Request accepted for processing
204	No Content	Request succeeded, no content returned

Redirection:

301	Moved Permanently	Resource permanently moved; use new **URL** for future requests
302	Found	Temporary redirect; client may retry at original **URL** later
303	See Other	Redirect after **POST**; client should use **GET** method
304	Not Modified	Resource unchanged; cached version is still valid
307	Temporary Redirect	Temporary redirect; repeat request using same method
308	Permanent Redirect	Permanent redirect; repeat request with same method at new **URL**

Client Errors:

400	Bad Request	Request malformed or invalid
401	Unauthorized	Authentication required or failed
403	Forbidden	Authenticated, but access is not allowed
404	Not Found	Resource not found
405	Method Not Allowed	Method is not allowed on this resource
408	Request Timeout	Client took too long to send request
429	Too Many Requests	Client sent too many requests (rate limit)

Server Errors:

500	Internal Server Error	Generic server error
501	Not Implemented	Method not supported by server
502	Bad Gateway	Server received an invalid response from another server it contacted
503	Service Unavailable	Server is overloaded or down
504	Gateway Timeout	Server didn't receive a response in time from another server

---

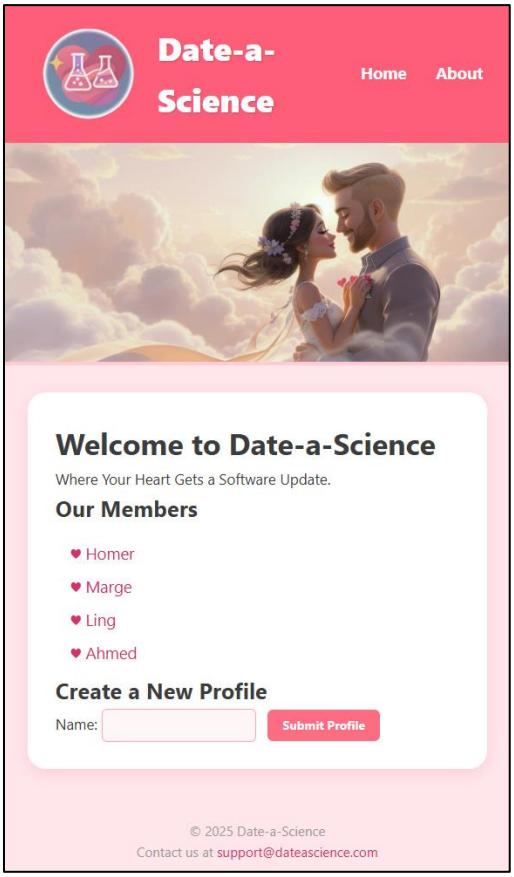
In conclusion … **REST**ful design is all about treating resources consistently and letting clients interact with them through a standard interface. Since different clients may need the same resource in different ways (i.e., humans using a browser, programs running on servers, or mobile apps) it's best that our server support multiple data formats. **HTML** works well for humans in a browser, while programs or other applications usually prefer machine-readable formats like **JSON** or plain text.

To achieve this, the server typically stores a single main representation of each resource, such as a **JSON** file or a database entry, and converts it into other formats on demand. This can be done by checking the client's request headers (e.g., **Accept: application/json** vs. **Accept: text/html**) or by using query parameters (e.g., **?format=json**).

By providing flexible representations, our **API** can be used by a wide range of clients without duplicating data or creating separate endpoints for each format.

11.3 Practice Exercise - Date-a-Science Website API

Let's do a practice exercise. Going back to our Data-a-Science site, lets design a RESTful **API** that will allow us to **search members, view member profiles, see suggested matches,** and **manage user accounts**. We will build this **API** step-by-step, with questions guiding our design. Of course, here in the notes, we are just explaining it all, but pause at each question (in red text) to see if you can come up with the results on your own.



1. What resources will we need?

- **Real-world objects**:
  - `members`
  - `users`
- **Digital items**:
  - `photos`
  - `reviews`
  - `messages`
- **Abstract concepts:**
  - `matches`
  - `sessions`
  - `interactions` (e.g., likes, dislikes, match accepted/rejected)
  - `notifications` (e.g., match/like/message/security alerts)
- **System resources:**
  - `settings`

2. What data do we need for each resource?

- `members`
  - `id` → unique identifier
  - `name` → first/last or display name
  - `age` / `birthdate`
  - `gender` / `pronouns`
  - `location` → city, state, or coordinates
  - `bio` / `description` → short personal summary
  - `photos` → array of photo **URL**s
  - `interests` / `hobbies` → array of strings
  - `lastActive` → timestamp for last login or activity

- **users**
  - `id` → unique user ID
  - `username` / `login` → account identifier
  - `passwordHash` → stored securely
  - `email` → optional, for verification or notifications
  - `profileId` → links to the member profile
  - `createdAt` → account creation timestamp

- **photos**
  - `id` → unique photo ID
  - `url` / `path` → location of photo
  - `caption` → optional text
  - `uploadedAt` → timestamp
  - `ownerId` → member ID who owns the photo

- **reviews**
  - `id` → unique review ID
  - `authorId` → member writing the review
  - `targetId` → member receiving the review
  - `rating` → number (e.g., 1–5)
  - `comment` → optional text
  - `createdAt` → timestamp

- **messages**
  - `id` → unique message ID
  - `senderId` / `receiverId` → member IDs
  - `content` → text
  - `sentAt` → timestamp
  - `readStatus` → boolean or timestamp

- **matches**
  - `id` → unique match ID (optional)
  - `memberId` → the member receiving suggestions
  - `matchedMemberId` → suggested member ID
  - `score` / `compatibility` → optional number indicating match strength
  - `createdAt` / `suggestedAt` → timestamp

- **sessions**
  - `id` / `token` → session token
  - `userId` → linked account
  - `createdAt` / `expiresAt` → timestamps
  - `device` / `IP` → optional metadata

- **interactions**
  - `id` → unique interaction ID
  - `memberId` → person performing the action
  - `targetId` → person receiving the action
  - `type` → like, super-like, dislike, etc.
  - `timestamp` → when the action occurred

- **notifications**
  - **id** → unique notification ID
  - **memberId** → recipient
  - **type** → message, match, like, system alert
  - **content** → optional text
  - **readStatus** → boolean
  - **createdAt** → timestamp

- **settings**
  - **id** / **userId** → links to account
  - **notificationsEnabled** → boolean
  - **visibility** / **privacy settings** → e.g., show profile to everyone or matches only
  - **language** / **locale** → string
  - **theme** / **display settings** → optional

3. What are the API endpoints (i.e., method and URL)?

- **members:**

GET	/members	Search or list members with optional query params
GET	/members/{mID}	Retrieve a specific member profile
PUT	/members/{mID}	Update a member profile (bio, photos, interests, etc.)
DELETE	/members/{mID}	Delete a member profile (optional, if allowed)

- **users:**

POST	/users	Create a new user account
GET	/users/{uID}	Retrieve user account info
PUT	/users/{uID}	Update user account info (password, email, settings)
DELETE	/users/{uID}	Delete a user account

- **photos:**

GET	/members/{mID}/photos	List all photos for a member
POST	/members/{mID}/photos	Upload a new photo
GET	/members/{mID}/photos/{photoID}	Retrieve a specific photo
DELETE	/members/{mID}/photos/{photoID}	Delete a photo

- **reviews:**

GET	/members/{mID}/reviews	List reviews for a member
POST	/members/{mID}/reviews	Submit a review for a member
GET	/members/{mID}/reviews/{reviewID}	Retrieve a specific review
DELETE	/members/{mID}/reviews/{reviewID}	Delete a review

- **`messages:`**

GET	/members/{mID}/messages	List messages for a member
POST	/members/{mID}/messages	Send a new message
GET	/members/{mID}/messages/{messageID}	Retrieve a specific message
DELETE	/members/{mID}/messages/{messageID}	Delete a message

- **`matches:`**

GET	/members/{mID}/matches	Get a list of suggested matches for a member
POST	/members/{mID}/matches	Record or generate a new match (system-driven)

- **`sessions:`**

POST	/sessions	Create a new login session (authenticate)
DELETE	/sessions/{sessionID}	Logout / delete a session

- **`interactions:`**

GET	/members/{mID}/interactions	List interactions for a member
POST	/members/{mID}/interactions	Record a new interaction
GET	/members/{mID}/interactions/{interactionID}	Retrieve a specific interaction
DELETE	/members/{mID}/interactions/{interactionID}	Remove an interaction

- **`notifications:`**

GET	/members/{mID}/notifications	List notifications for a member
POST	/members/{mID}/notifications	Create a notification (system)
GET	/members/{mID}/notifications/{notificationID}	Retrieve a specific notification
PUT	/members/{mID}/notifications/{notificationID}	Mark notification as read
DELETE	/members/{mID}/notifications/{notificationID}	Delete a notification

- **`settings:`**

GET	/users/{uID}/settings	Retrieve user settings / preferences
PUT	/users/{uID}/settings	Update user settings / preferences

4. What are the expected queries for these endpoints?

Resource / Type	Attributes	HTTP Method	Endpoint / URL Pattern	Typical Query / Request Parameters	Notes / Response
Members / Profiles	id, name, age, gender, bio, location, photos[], interests[], lastActive	GET	/members	name, min-age, max-age, interest, gender, location, limit, offset, sort	List/search members
		GET	/members/{mID}	none	Retrieve single member profile
		PUT	/members/{mID}	JSON body with attributes to update	Update member profile
		DELETE	/members/{mID}	none	Delete member profile
Users / Accounts	id, username, passwordHash, email, profileId, createdAt	POST	/users	JSON: {username, password}	Create new account, 201 Created
		GET	/users/{uID}	none	Retrieve user account
		PUT	/users/{uID}	JSON body with updates	Update account
		DELETE	/users/{uID}	none	Delete account
Photos	id, url, caption, uploadedAt, ownerId	GET	/members/{mID}/photos	limit, offset, sort	List member photos
		POST	/members/{mID}/photos	file + optional JSON metadata	Upload photo
		GET	/members/{mID}/photos/{photoID}	none	Retrieve photo
		DELETE	/members/{mID}/photos/{photoID}	none	Delete photo
Messages	id, senderId, receiverId, content, sentAt, readStatus	GET	/members/{mID}/messages	sender-id, receiver-id, read-status, limit, offset, sort	List messages
		POST	/members/{mID}/messages	JSON: {receiver-id, content}	Send message
		GET	/members/{mID}/messages/{messageID}	none	Retrieve message
		DELETE	/members/{mID}/messages/{messageID}	none	Delete message
Reviews / Testimonials	id, authorId, targetId, rating, comment, createdAt	GET	/members/{mID}/reviews	author-id, rating-min, rating-max, limit, offset, sort	List reviews
		POST	/members/{mID}/reviews	JSON: {author-id, rating, comment}	Submit review
		GET	/members/{mID}/reviews/{reviewID}	none	Retrieve review
		DELETE	/members/{mID}/reviews/{reviewID}	none	Delete review
Matches	id, memberId, matchedMemberId, score, createdAt	GET	/members/{mID}/matches	limit, sort, exclude-viewed	Get suggested matches
		POST	/members/{mID}/matches	optional system-generated	Record new match
Sessions / Login	id/token, userId, createdAt, expiresAt, device/IP	POST	/sessions	JSON: {username, password}	Login / create session
		DELETE	/sessions/{sessionID}	none	Logout / delete session
Interactions / Likes	id, memberId, targetId, type, timestamp	GET	/members/{mID}/interactions	type, target-id, limit, offset, sort	List interactions
		POST	/members/{mID}/interactions	JSON: {target-id, type}	Create interaction
		GET	/members/{mID}/interactions/{interactionID}	none	Retrieve interaction
		DELETE	/members/{mID}/interactions/{interactionID}	none	Remove interaction
Notifications	id, memberId, type, content, readStatus, createdAt	GET	/members/{mID}/notifications	read-status, type, limit, offset, sort	List notifications
		POST	/members/{mID}/notifications	optional system-generated	Create notification

		GET	/members/{mID}/notifications/{notificationID}	none	Retrieve notification
		PUT	/members/{mID}/notifications/{notificationID}	JSON: {read-status: true/false}	Mark read/unread
		DELETE	/members/{mID}/notifications/{notificationID}	none	Delete notification
Settings / Preferences	id/userId, notificationsEnabled, visibility, privacy, language, theme	GET	/users/{uID}/settings	optional section	Retrieve settings
		PUT	/users/{uID}/settings	JSON body with updates	Update settings

What are examples of queries and their expected **200 OK** response bodies?

GET /members?name=Alex&min-age=25&max-age=35

```
[
  {
    "id": 101,
    "name": "Alex Johnson",
    "age": 28,
    "gender": "male",
    "bio": "Loves hiking and coffee",
    "location": "Toronto",
    "interests": ["hiking", "coffee"],
    "lastActive": "2025-10-12T12:34:56Z"
  },
  {
    "id": 102,
    "name": "Alexandra Smith",
    "age": 27,
    "gender": "female",
    "bio": "Enjoys painting and yoga",
    "location": "Toronto",
    "interests": ["painting", "yoga"],
    "lastActive": "2025-10-12T11:20:10Z"
  }
]
```

GET /members?interest=hiking&location=Toronto&limit=5

```
[
  { "id": 101, "name": "Alex Johnson", "age": 28 },
  { "id": 103, "name": "Brian Lee", "age": 30 },
  ...
]
```

GET /members?limit=10&offset=20&sort=last-active-desc

```
[ ... next 10 member objects ... ]
```

GET /members/101/photos?limit=5&sort=uploaded-at-desc

```
[
  { "id": 501, "url": "/images/501.jpg", "caption": "Hiking trip", "uploadedAt": "2025-10-10T10:00:00Z" },
  { "id": 502, "url": "/images/502.jpg", "caption": "Coffee shop", "uploadedAt": "2025-10-08T15:30:00Z" }
]
```

GET /members/101/messages?sender-id=202&read-status=false

```
[
  { "id": 1001, "senderId": 202, "receiverId": 101, "content": "Hi there!", "sentAt":
"2025-10-12T10:12:00Z", "readStatus": false }
]
```

GET /members/101/messages?limit=20&offset=0&sort=sent-at-desc

```
[ ... last 20 messages ... ]
```

GET /members/101/reviews?rating-min=4&rating-max=5&limit=10

```
[
  { "id": 201, "authorId": 102, "targetId": 101, "rating": 5, "comment": "Great person!",
"createdAt": "2025-10-11T14:00:00Z" }
]
```

GET /members/101/reviews?sort=created-at-desc

```
[ ... reviews sorted newest first ... ]
```

GET /members/101/matches?limit=5&sort=score-desc

```
[
  { "memberId": 201, "matchedMemberId": 101, "score": 98 },
  { "memberId": 202, "matchedMemberId": 101, "score": 95 }
]
```

GET /members/101/matches?exclude-viewed=true

```
[ ... only unseen match suggestions ... ]
```

GET /members/101/interactions?type=like

```
[
  { "id": 301, "memberId": 101, "targetId": 202, "type": "like", "timestamp": "2025-10-
10T09:00:00Z" }
]
```

GET /members/101/interactions?limit=10&offset=20&sort=timestamp-desc

```
[ ... next 10 interactions ... ]
```

GET /members/101/notifications?read-status=false

```
[
  { "id": 401, "memberId": 101, "type": "match", "content": "You matched with Alex!",
"readStatus": false, "createdAt": "2025-10-12T08:00:00Z" }
]
```

GET /members/101/notifications?type=match&sort=created-at-desc

```
[ ... match notifications, newest first ... ]
```

5. What are the expected response codes?

Resource / Type	HTTP Method	Endpoint / URL Pattern	Response Codes	Notes / Response
Members / Profiles	GET	/members	200, 400	List/search members
	GET	/members/{mID}	200, 404	Retrieve single member profile
	PUT	/members/{mID}	200, 400, 404	Update member profile
	DELETE	/members/{mID}	204, 404	Delete member profile
Users / Accounts	POST	/users	201, 400	Create new account
	GET	/users/{uID}	200, 404	Retrieve user account
	PUT	/users/{uID}	200, 400, 404	Update account
	DELETE	/users/{uID}	204, 404	Delete account
Photos	GET	/members/{mID}/photos	200, 404	List member photos
	POST	/members/{mID}/photos	201, 400, 404	Upload photo
	GET	/members/{mID}/photos/{photoID}	200, 404	Retrieve photo
	DELETE	/members/{mID}/photos/{photoID}	204, 404	Delete photo
Messages	GET	/members/{mID}/messages	200, 404	List messages
	POST	/members/{mID}/messages	201, 400, 404	Send message
	GET	/members/{mID}/messages/{messageID}	200, 404	Retrieve message
	DELETE	/members/{mID}/messages/{messageID}	204, 404	Delete message
Reviews / Testimonials	GET	/members/{mID}/reviews	200, 404	List reviews
	POST	/members/{mID}/reviews	201, 400, 404	Submit review
	GET	/members/{mID}/reviews/{reviewID}	200, 404	Retrieve review
	DELETE	/members/{mID}/reviews/{reviewID}	204, 404	Delete review
Matches	GET	/members/{mID}/matches	200, 404	Get suggested matches
	POST	/members/{mID}/matches	201, 400, 404	Record new match
Sessions / Login	POST	/sessions	201, 400, 401	Login / create session
	DELETE	/sessions/{sessionID}	204, 404	Logout / delete session
Interactions / Likes	GET	/members/{mID}/interactions	200, 404	List interactions
	POST	/members/{mID}/interactions	201, 400, 404	Create interaction
	GET	/members/{mID}/interactions/{interactionID}	200, 404	Retrieve interaction
	DELETE	/members/{mID}/interactions/{interactionID}	204, 404	Remove interaction
Notifications	GET	/members/{mID}/notifications	200, 404	List notifications
	POST	/members/{mID}/notifications	201, 400, 404	Create notification
	GET	/members/{mID}/notifications/{notificationID}	200, 404	Retrieve notification
	PUT	/members/{mID}/notifications/{notificationID}	200, 400, 404	Mark read/unread
	DELETE	/members/{mID}/notifications/{notificationID}	204, 404	Delete notification
Settings / Preferences	GET	/users/{uID}/settings	200, 404	Retrieve settings
	PUT	/users/{uID}/settings	200, 400, 404	Update settings

Success codes: **200** OK, **201** Created, **204** No Content
Client errors: **400** Bad Request, **401** Unauthorized, **403** Forbidden, **404** Not Found, **429** Rate Limit
Server errors: **500** Internal Error … implied for all endpoints

As we can see, creating an **API** for a website involves many "moving parts", from defining resources and endpoints to specifying query parameters, request bodies, and response codes. However, by taking the time to design the **API** carefully and consistently, we make the implementation much smoother, reduce the likelihood of errors, and ensure that our code will be easier to maintain and extend in the future.