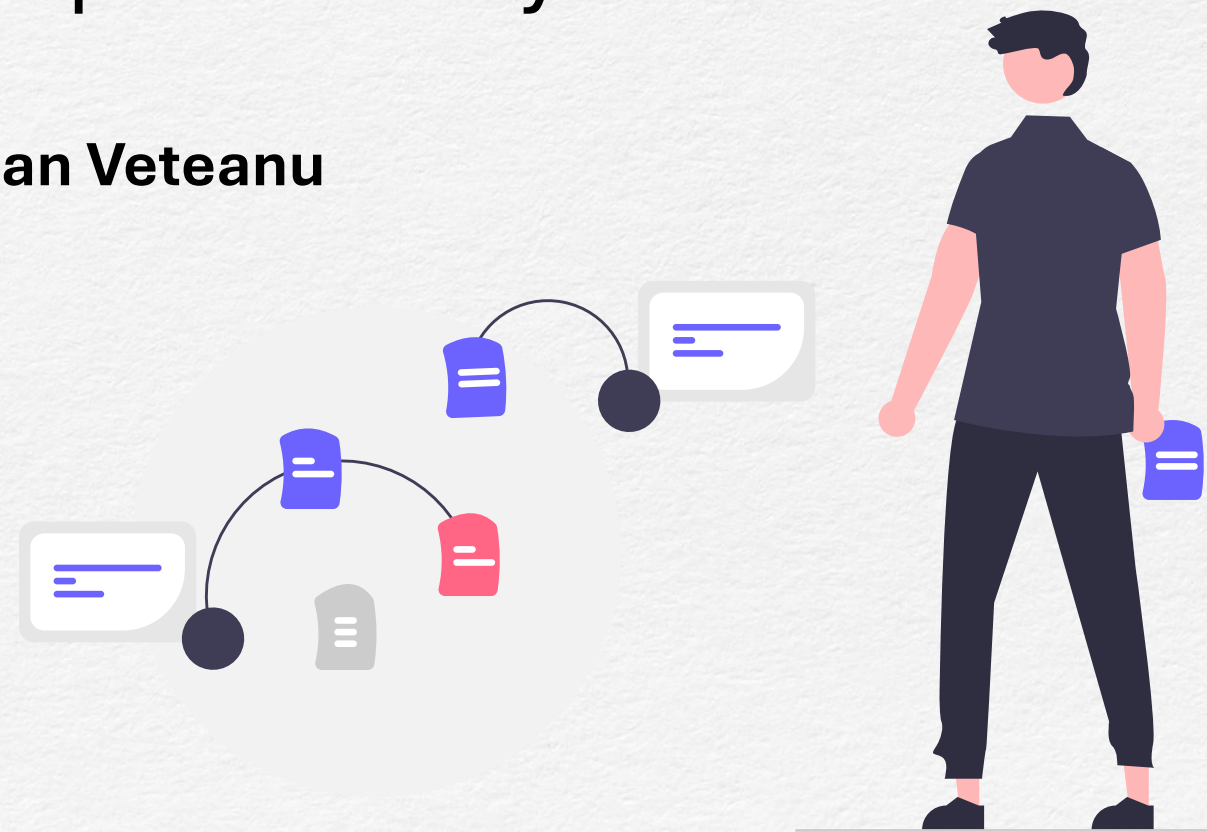
21 Tips for Designing Web APIs

Best Practices and Principles for Scalable, Secure, and Developer-Friendly APIs

by Marian Veteanu



From APIs to Web APIs

What is an API?

An **API (Application Programming Interface)** enables applications to communicate and share data.

Internal APIs

Internal APIs are used within an organization or application to connect services, modules, or systems, supporting modularity and enhancing functionality internally.

External APIs

External APIs are made available to third-party developers or customers to extend the functionality of a product.

Web APIs

Web APIs are specifically designed to enable communication over the Internet. Protocols enabling web APIs include REST, GraphQL, and SOAP.

Web APIs facilitate modular and scalable development, making it easy to integrate with third-party services and support microservices architectures.

Tip #1: Choose the Right API Style: REST, SOAP, or GraphQL

SOAP
Heavyweight, uses XML, and is more enterprise oriented. SOAP is generally found in older applications.

REST
Lightweight, more modern, uses HTTP, and is stateless.

```
// REST endpoint example
GET /api/products
```

GraphQL
Enables clients to specify required data, preventing over-fetching or under-fetching. Often considered alternative to REST.

```
// GraphQL query example
query { product(id: 1) { name, price } }
```

gRPC (gRPC Remote Procedure Calls)
A high-performance, language-agnostic RPC framework designed for fast communication between microservices, typically using HTTP/2.

In this presentation, our focus will primarily be on RESTful APIs.

Tip #2: Understand Idempotency and Its Importance

Idempotency is a concept in API design where making multiple identical requests has the same effect as making a single request.

In other words, an operation is idempotent if repeating it doesn't produce different outcomes or side effects.

GET: Always idempotent; retrieves the same data with each request.
PUT: Designed to be idempotent; updates or replaces a resource, resulting in the same final state if repeated.
DELETE: Should be idempotent; removes a resource, so repeated calls should yield the same "deleted" state.
POST: Typically non-idempotent; used for creating resources, where each request can produce a new resource.

Best Practices for Ensuring Idempotency

Use Unique Identifiers: For operations like POST, generate unique IDs client-side or server-side to avoid duplicate creations.
Implement Request Deduplication: For non-idempotent requests, consider using unique request IDs or tokens to detect and prevent duplicates.
Document Idempotency for Each Endpoint: Clearly indicate in the API documentation whether each endpoint is idempotent, helping developers avoid unintended behaviors.

Tip #3: Implement Rate Limiting to Prevent Abuse

Rate limiting is a technique used to control the number of requests a client can make to an API within a specific time period.

Limits protect the API from being overwhelmed by excessive requests, whether intentional (e.g., spam or DDoS attacks) or unintentional, ensuring that one client or user doesn't monopolize API resources.

User-Based Rate Limiting: Limits are applied based on user identity (e.g., 100 requests per user per minute).

IP-Based Rate Limiting: Limits are set per IP address, commonly used for APIs without user authentication.

Global Rate Limiting: Sets a limit across all clients to protect the server from high overall load.

Endpoint-Based Rate Limiting: Different rate limits can be applied to different endpoints (e.g., lower limits for data-heavy operations).

```
const rateLimit = require('express-rate-limit');

// Define rate limiting rule: 100 requests per 15 minutes per IP
const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100, // Limit each IP to 100 requests per windowMs
  message: "Too many requests from this IP, please try again later."
});
```

Tip #4: Use Versioning to Maintain Compatibility

API versioning allows developers to introduce new features, fix issues, or make improvements without breaking existing functionality for current users.

When to Introduce a New API Version?

- **Breaking Changes:** Introduce a new version if there's a significant change in data structure, required parameters, or behavior that could disrupt current users.
- **Major New Features:** Create a new version if a major new feature set requires modifications that would impact the current API design.
- **Deprecation of Old Features:** When phasing out legacy features, provide them only in older versions, encouraging users to migrate.

Common Versioning Approaches:

URI Versioning (Most common):

Adds the version number in the URL path, making it clear and accessible.

Example: GET /api/v1/users vs. GET /api/v2/users

Header Versioning:

Specifies the version in a custom header, keeping URLs clean.

Example: Accept: application/vnd.yourapp.v1+json

Query Parameter Versioning:

Appends the version as a query parameter in the URL, which can be flexible but may clutter the URL.

Example: GET /api/users?version=1

Content Negotiation:

Uses the Accept header to specify the version, allowing the server to choose the best response format.

Example: Accept: application/vnd.yourapp+json; version=1

Tip #5: Consider using HATEOAS for Dynamic Discovery

HATEOAS (Hypermedia as the Engine of Application State) is a principle within RESTful APIs where each response contains links that guide clients on how to navigate and interact with the API dynamically.

By embedding hypermedia links in responses, clients can discover available actions without hardcoding endpoint paths, making the API more self-descriptive and adaptable to future changes.

Response with HATEOAS Links:

```
app.get('/api/v1/users/:id', async (req, res) => {
    const user = await UserService.getUserById(req.params.id);

    // Constructing the HATEOAS response
    const response = {
      id: user.id,
      name: user.name,
      email: user.email,
      links: {
        self: { href: `/api/v1/users/${user.id}` },
        posts: { href: `/api/v1/users/${user.id}/posts` },
        comments: { href: `/api/v1/users/${user.id}/comments` }
      }
    };

    res.json(response);
  });
```

Using HATEOS may increase both server-side and client-side complexity. Some clients may need adjustments to handle dynamic navigation based on hypermedia, which may be challenging if they’re designed for static URLs.

Tip #6: Embrace Statelessness for Scalability

In a stateless API, every request must contain all the necessary information for the server to understand and process it, without relying on previous interactions (the server does not store any session or client data between requests).

Why is statelessness important?

Scalability: Because there's no need to store session data on the server, it's easier to distribute requests across multiple servers, enabling horizontal scaling.

Resilience: Statelessness allows any server instance to handle any request, improving reliability and enabling failover if one server goes down.

Simplifies Load Balancing: Load balancers can distribute requests evenly without concern for session continuity, making load distribution straightforward.

Enhanced Security: Statelessness reduces the risk of data leakage, as session data isn't stored server-side. Instead, any necessary state is transferred with each request, usually in the form of tokens or session identifiers.

Authentication

In a stateless API, authentication is typically managed using tokens (e.g., JWT - JSON Web Tokens) sent with each request. Clients authenticate once, receive a token, and include it in the Authorization header for each subsequent request.

```
GET /api/v1/orders
Host: example.com
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
```

Tip #7: Secure APIs with OAuth and JWT

APIs are often accessible over the internet, making them vulnerable to unauthorized access and various types of attacks.

OAuth Use Case: An application requiring user authentication via a third-party service, such as a mobile app allowing users to sign in with Google or Facebook.

Authorization Request: The client app redirects the user to an authorization server (e.g., Google) where the user grants permission.
Authorization Code: The authorization server redirects the user back to the app with an authorization code.
Token Exchange: The client app exchanges the authorization code for an access token by authenticating itself with the authorization server.
Access Token Usage: The app uses the access token to access the API on the user's behalf, providing it in the Authorization header.

JWT Use Case: A single-page application (SPA) that uses JWT for secure, stateless authentication, allowing the app to make authenticated requests without storing user sessions on the server.

User Logs In: The user sends their login credentials (e.g., username and password) to the API.
Token Issuance: If credentials are valid, the server generates a JWT containing user information and returns it to the client.
Client Stores Token: The client stores the JWT (e.g., in local storage or cookies).
Token Sent with Requests: For each API request, the client sends the JWT in the Authorization header.
Token Verification: The server verifies the JWT on each request and, if valid, processes the request.

JWT Code Example in Node.js

```
const jwt = require('jsonwebtoken');
const express = require('express');
const app = express();

// Secret for JWT signing
const JWT_SECRET = 'your_jwt_secret';

// Login route to authenticate users and issue a token
app.post('/login', (req, res) => {
  const { username, password } = req.body;
  // Authenticate user
  const user = authenticateUser(username, password);
  if (user) {
    const token = jwt.sign({ id: user.id, username: user.username },
JWT_SECRET, { expiresIn: '1h' });
    res.json({ token });
  } else {
    res.status(401).json({ message: 'Invalid credentials' });
  }
});

// Middleware to protect routes
function authenticateToken(req, res, next) {
  const authHeader = req.headers['authorization'];
  const token = authHeader && authHeader.split(' ')[1];
  if (!token) return res.sendStatus(401);

  jwt.verify(token, JWT_SECRET, (err, user) => {
    if (err) return res.sendStatus(403);
    req.user = user;
    next();
  });
}

// Protected route example
app.get('/api/protected', authenticateToken, (req, res) => {
  res.json({ message: "Welcome to the protected route, " +
req.user.username });
});

app.listen(3000, () => console.log("Server running on port 3000"));
```

Tip #8: Choose the Correct HTTP Methods

The most common HTTP methods used in RESTful APIs are **GET**, **POST**, **PUT**, **PATCH**, and **DELETE**. Each method has a specific purpose, following REST principles.

GET
Purpose: Retrieves data from the server without modifying it.
Idempotent: Yes, meaning multiple GET requests should return the same result.
Use Case: Fetching data (e.g., retrieving a user's profile or list of products).
Example:

```
GET /api/v1/users/123
```

PUT
Purpose: Updates or replaces an existing resource. If the resource doesn't exist, some APIs may create it.
Idempotent: Yes, as repeated PUT requests result in the same final state.
Use Case: Updating a resource with new data (e.g., updating user information).

DELETE
Purpose: Removes a specified resource from the server.
Idempotent: Yes, as deleting a resource repeatedly will yield the same result (resource remains deleted).
Use Case: Deleting a resource (e.g., removing a user from the system).
Example:

```
DELETE /api/v1/users/123
```

POST
Purpose: Creates a new resource on the server.
Idempotent: No, as each request typically results in a new resource.
Use Case: Adding new resources (e.g., creating a new user, adding an item to an order).
Example:

```
POST /api/v1/users
Content-Type: application/json
{
  "name": "Alice",
  "email": "alice@example.com"
}
```

PATCH
Purpose: Partially updates an existing resource by only sending the fields to be modified.
Idempotent: Yes, as repeating the same request has the same effect.
Use Case: Making partial updates to an object (e.g., updating only the email).
Example:

```
PATCH /api/v1/users/123
Content-Type: application/json
{
  "email": "newemail@example.com"
}
```

Tip #9: Use Caching for Performance Optimization

Caching reduces the need to reprocess frequently requested data, which decreases the workload on the server.

Client-Side Caching:
The client (e.g., a browser or mobile app) stores a copy of the data locally, reducing the need to repeatedly fetch the same data from the server.
Useful for static assets and infrequent updates.
Controlled with HTTP headers like Cache-Control and ETag.

Server-Side Caching:
The server caches frequently requested data to reduce redundant processing.
Commonly used for caching database queries, heavy computations, or API responses.
Can be stored in memory (e.g., using Redis or Memcached) or in a dedicated caching layer.

Proxy Caching:
Proxy servers or Content Delivery Networks (CDNs) store copies of frequently requested responses closer to the client's location, reducing latency.
Ideal for APIs serving global clients or handling high traffic.

Tips for Effective Caching

Use Distributed Caching for High Availability:

- In distributed systems, use a distributed cache (e.g., Redis, Memcached) that multiple servers can access to ensure data consistency and availability.

Combine Caching Strategies:

- Use a mix of client-side, server-side, and proxy caching to create a layered caching strategy that maximizes efficiency.

Cache Preloading:

- For high-demand resources, consider preloading caches during off-peak hours to ensure data is ready for peak traffic.

Tip #10: Use Pagination for Large Datasets

By returning data in smaller chunks, pagination reduces memory usage and processing time on the server and allows APIs to handle large datasets by distributing data across multiple requests.

Clients can retrieve only the data they need instead of loading entire datasets, which enhances responsiveness and user experience.

Offset-Based Pagination

Uses an offset and limit to determine which set of results to return. Commonly used with SQL databases.

Example: GET /api/v1/products?offset=20&limit=10 (skips the first 20 items and returns the next 10).

Pros: Simple to implement and widely supported.

Cons: Performance can decrease for large offsets, especially with large datasets.

Page-Based Pagination

Divides results into numbered pages, where each page contains a fixed number of items.

Example: GET /api/v1/products?page=3&size=10 (returns the 3rd page with 10 items per page).

Pros: Intuitive and easy to navigate.

Cons: Similar to offset pagination.

Cursor-Based (Keyset) Pagination

Uses a unique identifier (e.g., timestamp or ID) from the last item in the current set to fetch the next set of results.

Example: GET /api/v1/products?cursor=12345&limit=10 (returns items after item with ID 12345).

Pros: Efficient for large datasets and avoids issues with large offsets.

Cons: More complex to implement and requires a unique, sequential key.

Time-Based Pagination

Uses timestamps to paginate through records that are sorted chronologically, useful for time-sensitive data like logs or event streams.

Example: GET /api/v1/logs?start=2023-01-01T00:00:00Z&end=2023-01-01T23:59:59Z&limit=100.

Pros: Effective for time-series data and real-time updates.

Cons: Limited to data sorted by time and may not be suitable for all datasets.

Tip #11: Implement Standard HTTP Status Codes

Standard HTTP status codes provide clients with a clear understanding of the outcome of their requests.

Using well-known codes reduces ambiguity, as developers are familiar with them, making debugging easier.

Standard status codes allow APIs to work seamlessly with various clients, libraries, and tools that rely on these codes for error handling and response handling.

```
if (!user) {
  // 404 Not Found
  return res.status(404).json({
    status: 404,
    error: "Not Found",
    message: `User with ID ${userId}
              not found`
  });
}
```

```
// 200 OK
res.status(200).json({
  status: 200,
  data: user
});
```

2xx Success Codes:

200 OK
201 Created
202 Accepted
204 No Content

3xx Redirection Codes:

304 Not Modified (Used for caching)

4xx Client Error Codes:

400 Bad Request
401 Unauthorized
403 Forbidden
404 Not Found
405 Method Not Allowed
409 Conflict
429 Too Many Requests

5xx Server Error Codes:

500 Internal Server Error
502 Bad Gateway
503 Service Unavailable
504 Gateway Timeout

Tip #12: Use Query Parameters and Path Parameters Correctly

Path Parameters and **Query Parameters** are two primary ways to pass information to RESTful APIs, but each serves a distinct purpose.

Path Parameters

Used to identify specific resources in an API, typically forming part of the URL structure.

Example: /api/v1/users/123 (where 123 is a path parameter identifying a specific user).

Query Parameters

Used to filter, sort, or modify data retrieval, allowing clients to customize their requests.

Optional, and do not change the resource itself; instead, they modify the response or narrow down the results.

Example: /api/v1/users?sort=name&limit=10 (where sort and limit are query parameters used to refine results).

Tip #13: Ensure Data Consistency for Complex Operations

In complex systems, multiple operations can affect the same data. Ensuring consistency helps prevent issues such as race conditions, data duplication, and loss of data integrity.

Database Transactions: Utilize database transactions to group multiple operations into a single atomic unit. If any operation fails, the entire transaction is rolled back, preserving data integrity.
In relational DBs, you can use BEGIN, COMMIT, and ROLLBACK commands to control transactions. Some NoSQL DBs, like MongoDB, also support multi-document transactions to ensure atomicity across multiple documents.

Distributed Transactions: For distributed systems, consider using distributed transaction protocols like two-phase commit (2PC): Phase 1: Prepare, Phase 2: Commit/Rollback.

Distributed Locking: Prevent concurrent operations from causing conflicts. Redis provides a popular library called Redlock for distributed locks.

Eventual Consistency: Allows for faster operations, higher availability, and scalability, as the system does not need to lock or synchronize data immediately. Updates propagate across the system, but clients might see different versions temporarily.

Optimistic Concurrency Control (OCC):

- Assume that conflicts are rare and check for them at commit time.
- If a conflict is detected, the transaction is aborted, and the client can retry the operation.
- Suitable for low-conflict scenarios.

Pessimistic Concurrency Control (PCC):

- Lock resources to prevent concurrent access and conflicts.
- More suitable for high-conflict scenarios, but can impact performance.

Tip #14: Consider Using Custom Data Types (DTOs)

DTO (Data Transfer Object) is a design pattern used to transfer data between layers or components in an application, typically from the server to the client in the context of APIs.

When the data structure in your application is complex, and direct exposure would be confusing, DTOs allow you to design simpler, client-friendly structures.

```
// User DTO definition
class UserDTO {
    constructor(user) {
      this.id = user._id;
      this.name = user.name;
      this.email = user.email;
      this.role = user.role;
    }
  }

  // Express route with DTO
  app.get('/api/v1/users/:id', async (req, res) => {
    // Fetch user from the database
    const user = await UserService.getUserById(req.params.id);
    const userDTO = new UserDTO(user); // Transform user to DTO
    res.json(userDTO);
  });
```

Keep DTOs free of business logic; they should strictly be used to format and transfer data. Business logic should remain in the service or business layer.

Document the structure of each DTO in your API documentation, including fields, data types, and descriptions to help clients understand the expected responses.

Tip #15: Document Your API Effectively

Clear documentation helps developers quickly understand how to integrate and interact with your API, reducing frustration and confusion.

Consider providing a quick start guide with basic authentication, example requests, and setup instructions to help new users get started faster.

OpenAPI Specification (OAS) / Swagger, is a widely adopted standard for API documentation that provides a structured format for defining endpoints, parameters, responses, and more.

```yaml
openapi: 3.0.0
info:
  title: Example API
  version: "1.0"
paths:
  /users/{id}:
    get:
      summary: Retrieve user by ID
      parameters:
        - name: id
          in: path
          required: true
          schema:
            type: integer
      responses:
        '200':
          description: User found
```

Tools like Redoc, Slate, and RapiDoc generate professional documentation based on OpenAPI specifications.

These tools offer clean, user-friendly interfaces for browsing API details

Tip #16: Use CORS for Secure Cross-Origin Requests

CORS (Cross-Origin Resource Sharing) is a security mechanism that controls how web pages can request resources from a different origin (domain, protocol, or port) than the one that served the web page.

CORS restricts which external websites can access your API, reducing the risk of unauthorized data access and enhancing security.

```
// Configure CORS
app.use(cors({
  origin: 'https://example.com', // Restrict to specific origin
  methods: ['GET', 'POST', 'PUT', 'DELETE'], // Allowed methods
  allowedHeaders: ['Content-Type', 'Authorization'], // Headers
  credentials: true // Allow cookies and credentials
}));

app.get('/api/v1/data', (req, res) => {
  res.json({ message: "This is a CORS-enabled response!" });
});
```

Best Practices for Configuring CORS

Allow Specific Origins Only: Avoid setting Access-Control-Allow-Origin to *

Limit Allowed Methods: Avoid allowing methods like DELETE unless required.

Restrict Headers Carefully: For instance, only allow Authorization and Content-Type if these are required.

Enable Credentials Only When Necessary: Setting Access-Control-Allow-Credentials: true may expose sensitive data to other origins

Use Preflight Responses to Control Access: Ensure your server responds to preflight requests correctly by including Access-Control-Allow-Methods and Access-Control-Allow-Headers.

Tip #17: Handle File Uploads Securely

File uploads, if not properly managed, can expose APIs to malicious files, such as malware, that could harm the server or end-users.

Best Practices for Secure File Uploads

Limit Allowed File Types: Only permit specific file types (e.g., images, PDFs) and reject unsupported formats to reduce the risk of harmful files. Use MIME type checking or libraries like **file-type** in Node.js to detect file types based on content signatures:

```
const fileType = require('file-type');
const fileTypeResult = await fileType.fromBuffer(fileBuffer);
if (fileTypeResult.ext !== 'jpg' && fileTypeResult.ext !== 'png’)
{
  throw new Error('Invalid file type');
}
```

Set Maximum File Size and **Implement Rate Limiting for Uploads:** Limit the file size to prevent large uploads that could consume excessive resources or result in Denial of Service (DoS) attacks.

Use Secure Storage Locations: Store files in a separate directory outside the application’s root directory to avoid accidental access through the web server. Consider using cloud storage services like AWS S3, Google Cloud Storage, or Azure Blob Storage for storing files securely.

Rename Uploaded Files: Rename uploaded files to prevent issues with path traversal and overwrite attacks. A common approach is to generate a unique file name based on a UUID or timestamp.

Implement Virus Scanning: Scan uploaded files for malware before processing them.

Tip #18: Monitor and Log API Activity

Monitoring API activity helps detect and respond to suspicious behavior, such as unauthorized access attempts or abuse.

Use Structured Logging: Store logs in structured formats like JSON to make logs easier to parse and analyze programmatically, enabling more effective troubleshooting.

Set Log Levels: Define log levels (e.g., info, warn, error, debug) and log data accordingly to reduce noise. Errors and warnings should be logged separately from informational messages to improve readability.

Implement Log Rotation and Retention Policies:
Avoid excessive storage use by setting log rotation policies (e.g., daily log rotation) and retention limits, especially for high-volume APIs.

Anonymize or Obfuscate Sensitive Data:
Avoid logging sensitive information like user passwords, credit card details, or personally identifiable information (PII) to protect user privacy and comply with data protection regulations.

Use Distributed Tracing for Microservices:
In a microservices environment, use distributed tracing tools (e.g., OpenTelemetry, Jaeger) to trace requests across services and identify bottlenecks or failures in the call chain.

Alert on Anomalies:
Set up automated alerts for abnormal patterns, such as spikes in error rates, latency, or traffic, to enable proactive issue detection and faster resolution.

Tip #19: Use Webhooks for Real-Time Data

Webhooks are HTTP callbacks that enable one system to send real-time data or event notifications to another system automatically. Instead of polling for updates, a webhook allows an API to push data whenever a specific event occurs, such as a new user registration or payment transaction.

Step 1: Subscription/Registration:
The client registers a webhook URL with the service provider, specifying the events they want to receive (e.g., “new user signup” or “order completed”).

Step 2: Event Trigger:
When an event occurs in the service provider’s system (e.g., a new order is placed), the service triggers the webhook.

Step 3: Data Transmission:
The service provider makes an HTTP request (usually a POST) to the webhook URL with a payload containing details about the event.

Step 4: Processing the Webhook:
The client system receives and processes the webhook data, triggering the appropriate actions (e.g., updating a database, sending notifications, etc.).

Examples of Platforms Offering Webhooks

Stripe: Payment success/failure, subscription updates, chargebacks.
GitHub: Push events, pull requests, issue comments.
Shopify: New orders, product updates, inventory changes.
Twilio: SMS received, call status, messaging errors.

Tip #20: Make Use of API Management Services from Azure, AWS, and GCP

API Management services offered by major cloud providers like **Azure API Management**, **AWS API Gateway**, and **Google Cloud API Gateway** provide end-to-end solutions for deploying, securing, monitoring, and managing APIs at scale.

These services streamline API development and enable teams to enforce policies, set rate limits, secure endpoints, and gather analytics without needing to build these features from scratch.

Feature	Azure API Management	AWS API Gateway	Google Cloud API Gateway
API Security	OAuth2, JWT, IP filtering	IAM roles, API keys, Cognito	Google IAM, API keys, JWT, Google ID
Rate Limiting	Yes	Yes	Yes
Monitoring and Logging	Azure Monitor, Application Insights	CloudWatch, X-Ray	Cloud Monitoring, Cloud Logging
API Versioning	Built-in versioning and revisions	Custom implementation	Custom implementation
Multi-Protocol Support	REST, SOAP	REST, WebSocket	REST, gRPC
Serverless Integration	Azure Functions, Logic Apps	AWS Lambda	Cloud Functions, Cloud Run
Developer Portal	Built-in and customizable	Limited; requires custom setup	No native portal

Tip #21: Listen to Feedback and Continuously Improve

A responsive API team that continuously improves the service based on real user input increases trust, making the API more attractive to both current and prospective users.

Strategies for Gathering and Analyzing Feedback

User Surveys and Interviews: Conduct regular surveys or feedback sessions with users to understand their experience, what features they find valuable, and any challenges they encounter.

API Analytics and Monitoring: Use analytics platforms to capture metrics such as average response time, error rates, and most frequently used endpoints.

Feature Requests and Voting: Implement a feature request board where users can suggest new features and vote on existing suggestions.

Changelog and Release Notes: Track changes in a public changelog, so users can see what's new, fixed, or improved.

Conduct Usability Testing and Hackathons: Organize usability testing sessions with new users to observe how they interact with the API, highlighting potential onboarding issues or confusing endpoints.

Engage with the Developer Community: Participate in discussions on developer forums, GitHub issues, and social media, responding to feedback and keeping users informed of planned improvements.

Measure the Impact of Improvements: Track the impact of each change. Monitoring the effect of changes helps validate if updates align with user needs.

Marian Veteanu

Technology Architect and Product Leader

Excited to join an organization where I can make an impact!

Let’s connect and explore opportunities—message me!

https://www.linkedin.com/in/mveteanu/
https://x.com/mveteanu