RESTful API Design Cheatsheet

A quick reference guide covering the essential principles, methods, and best practices for designing RESTful APIs.

Core Principles

Key Concepts

REST (Representational State Transfer): An architectural style for building networked applications, relying on a stateless, client-server communication protocol, typically HTTP.
Resource: A key abstraction of information. It can be a document, image, service, or a collection of other resources.
Representation: The format in which a resource is transferred (e.g., JSON, XML).
Stateless: Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server.
Uniform Interface: REST relies on a uniform and predefined interface for interacting with resources.

Architectural Constraints

Client-Server: Separation of concerns; clients and servers can evolve independently.
Stateless: No client context is stored on the server between requests.
Cacheable: Responses should be cacheable to improve performance.
Uniform Interface: Standardized interaction via HTTP methods.
Layered System: Client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way.
Code on Demand (optional): Servers can extend client functionality by transferring executable code.

HTTP Methods

Common Methods

GET	Retrieve a resource. Should be a safe and idempotent operation.
POST	Create a new resource. May result in a new resource URI.
PUT	Update an existing resource. Replaces the entire resource.
PATCH	Partially modify a resource. Applies partial updates.
DELETE	Delete a resource.
OPTIONS	Describe the communication options for the target resource.
HEAD	Same as GET, but only transfers the status line and header section.

Idempotency

An operation is idempotent if performing it once has the same effect as performing it multiple times. `GET`, `PUT`, `DELETE`, and `HEAD` should be idempotent.
Example: Deleting a resource using `DELETE` multiple times should still result in the resource being gone after the first deletion.

Resource Design

URI Design

Use nouns to represent resources, not verbs. E.g., `/users` instead of `/getUsers`.
Use hierarchical URIs to represent relationships. E.g., `/users/{userId}/posts`.
Use plural nouns for collections. E.g., `/users`.
Avoid using file extensions in URIs. Content negotiation should be used instead (Accept header).
Use hyphens (-) to improve readability in URIs. E.g., `/blog-posts`.

URI Examples

/users	Collection of users.
/users/{userId}	A specific user.
/users/{userId}/posts	Posts by a specific user.
/posts/{postId}	A specific post.

Status Codes & Response Handling

Common Status Codes

200 OK	Successful request.
201 Created	Resource created successfully.
204 No Content	Request processed successfully, but no content to return.
400 Bad Request	Invalid request format or parameters.
401 Unauthorized	Authentication required.
403 Forbidden	The server understands the request, but refuses to authorize it.
404 Not Found	Resource not found.
500 Internal Server Error	Generic server error.

Content Negotiation

Use the `Accept` header in the request to specify the desired response format (e.g., `application/json`, `application/xml`).
Use the `Content-Type` header in the request to specify the format of the request body.
The server should respond with the appropriate `Content-Type` header indicating the format of the response.