APIs and REST

API

- API – Application programming interface
  - A stable interface that is presented by an application or module or library
  - A way to access the functionality of the component
  - Same pattern we have seen, the API is the interface, the component is the implementation
- Standardized APIs
  - In order to standardize how APIs work, various API protocols have been defined
  - Some examples
    - RPC – remote procedure calls are language specific, like calling a function from a Python module
    - CORBA – a protocol and architecture for cross language procedure calls
    - SOAP – Designed for inter application messaging using XML files

Types of APIs: Imperative vs Declarative

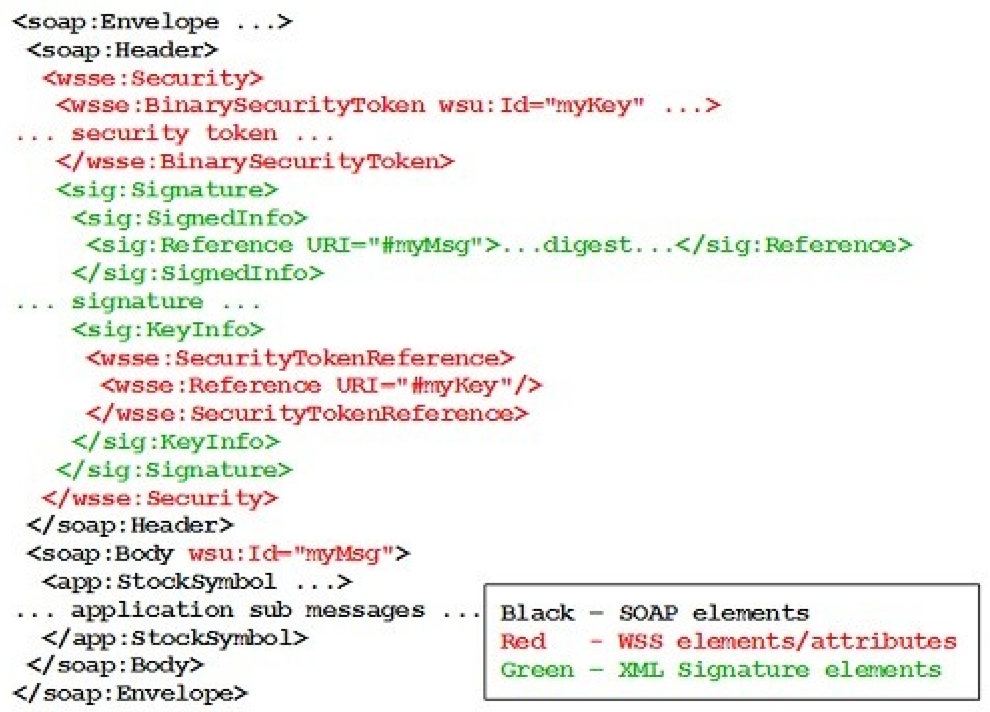
- Imperative (command/RPC style)
  - “Do this action now.”
  - Endpoints or methods are verbs that perform commands.
    - Eg rpc.CreateInvoice(request)
  - Pros:
    - Straightforward to call specific actions, often maps to internal functions, good for workflows
  - Cons:
    - Harder to compose, cache, or reason about state; can lead to many one-off endpoints; weaker use of HTTP semantics
    - Also makes for brittle interfaces
  - Characterized by having the message request a specific kind of processing to be performed

Types of APIs: Imperative vs Declarative

- Declarative (resource/state-oriented)
  - “Here’s something to process”
  - Primarily manipulate resources, also called entities or objects
  - Functionality to be executed depends on the entity provided
  - Examples:
    - Sending an invoice to the accounting department, the action implied is paying the invoice if it should be paid
    - The API receives an object and then the application figures out how to process it
  - The API usually provides CRUD functionality for these business objects
  - The objects are often transaction objects: sales, reports, shipments, hires, etc.

SOAP Legacy Protocol

- Developed during the interest in service oriented architecture (SOA)
  - A client would send an XML message requesting a list methods it could call
  - The server would return an XML file describing the services
  - The client would then encode an RPC as an XML file to be sent to the server
  - The server would figure out what to do from the content of the file
  - Execute the requested service
  - Then encode the result in an XML file and send it back to the client



```
<soap:Envelope ...>
 <soap:Header>
  <wsse:Security>
   <wsse:BinarySecurityToken wsu:Id="myKey" ...>
... security token ...
   </wsse:BinarySecurityToken>
   <sig:Signature>
    <sig:SignedInfo>
     <sig:Reference URI="#myMsg">...digest...</sig:Reference>
    </sig:SignedInfo>
... signature ...
    <sig:KeyInfo>
     <wsse:SecurityTokenReference>
      <wsse:Reference URI="#myKey"/>
     </wsse:SecurityTokenReference>
    </sig:KeyInfo>
   </sig:Signature>
  </wsse:Security>
 </soap:Header>
 <soap:Body wsu:Id="myMsg">
  <app:StockSymbol ...>
... application sub messages ...
  </app:StockSymbol>
 </soap:Body>
</soap:Envelope>
```

```
Black  - SOAP elements
Red    - WSS elements/attributes
Green  - XML Signature elements
```

SOAP Legacy Protocol

- Problems
  - Very difficult to implement
  - Protocol was complex and very heavyweight
  - Eventually abandoned but there are still legacy apps using SOAP
- Part of the problem was that SOAP tried to do everything
  - The whole messaging stack had to be defined

REST Web Protocol

- A lightweight protocol
- Piggybacks on top of Web protocols like HTTP
  - This eliminates the need to develop a whole protocol stack
  - And the transmission is handled by existing web infrastructure
- The REST (Representational State Transfer) architectural style was invented by Roy Fielding in the year 2000.
  - Introduced it in his doctoral dissertation titled "Architectural Styles and the Design of Network-based Software Architectures"
  - The goal was to simplify communication between systems

Restful Web Service

Uses HTTP to work with "resources" that are represented as URL endpoints

- REST uses HTTP methods to define operations
  - GET – Read a resource
  - POST – Create a resource
  - PUT – Update a resource
  - DELETE – Delete a resource
- Data is described in either JSON or XML
- RESTful operations are atomic and transactional

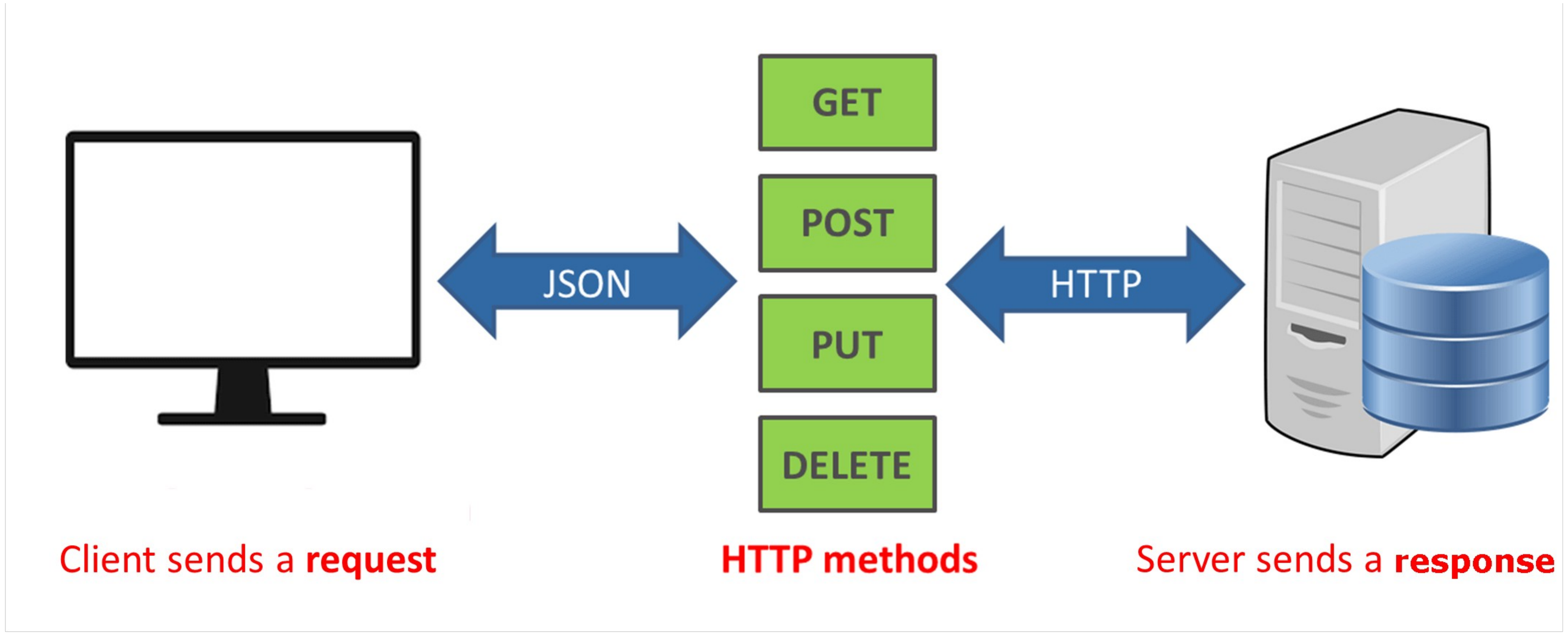
Representational state transfer (REST) is a software architectural style that defines a set of constraints to be used for creating Web services.

Web services that conform to the REST architectural style, called RESTful Web services, provide interoperability between computer systems on the Internet.

RESTful Web services allow the requesting systems to access and manipulate textual representations of Web resources by using a uniform and predefined set of stateless operations."

Wikipedia

Restful Web Service



REST Web Service Design Rules

- Client-Server
  - There should be a separation between the server that offers a service, and the client that consumes it
- Stateless
  - Each request from a client must contain all the information required by the server to carry out the request
  - The server cannot store information provided by the client in one request and use it in another request
- Cacheable
  - The server must indicate to the client if requests can be cached or not

REST Web Service Design Rules

- Layered System
  - Communication between a client and a server should be standardized in such a way that allows intermediaries to respond to requests instead of the end server, without the client having to do anything different
- Uniform Interface
  - The method of communication between a client and a server must be uniform
- Code on demand
  - Servers can provide executable code or scripts for clients to execute in their context
  - This constraint is the only one that is optional

Restful Web Service

The CRUD operations are indicated by the HTTP operation

- All endpoints should refer to some meaningful entity in the domain
- Operations can be represented by some form of transaction object
  - *For example, recording that something has been sold can be done by sending a "car" object with the appropriate data to represent registering a car.*
- The objects referred to in the API are always determined by the business domain

```
GET http://api.coolcars.io/cars/

GET http://api.coolcars.io/cars/{id}

DELETE http://api.coolcars.io/cars/{id}

POST http://api.coolcars.io/cars/
  {
   "make":"chevrolet",
   "model":"Silverado 3500",
   "year": 2004,
   "vin":"1GCJK33104F173427"
   }

Response: {"data":{"id": "8b7138db-0c7c-4e2e-8494-
bd5daf1788e0"}
```

BAD Restful Web Service

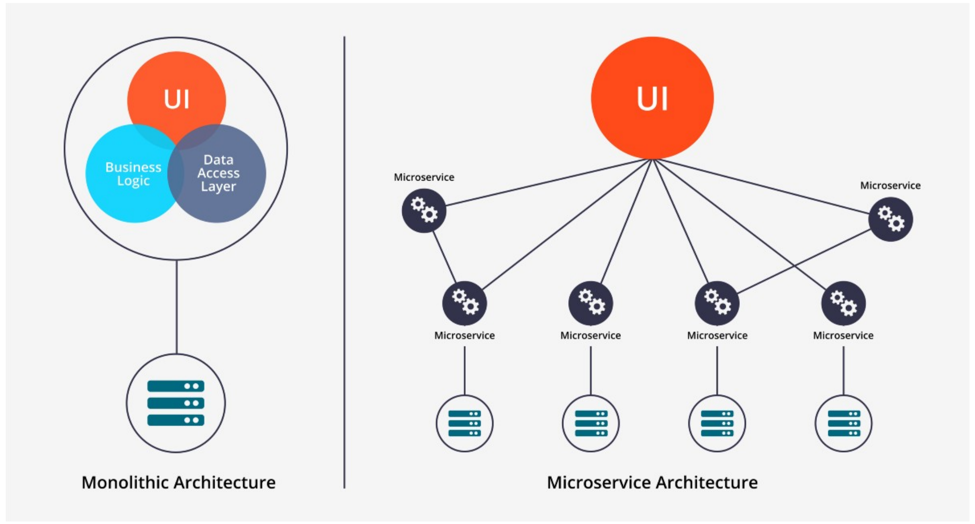
The REST endpoints are verbs (imperative)

- Creates a link between application logic and the web service API
- Add or changing app logic requires changing the web service
- For example, adding a “lease” capability to the business
- All business logic, state information and interaction with the client is done at the application layer instead of the application layer just interpreting the request and delegating to the appropriate module

```
GET http://api.coolcars.io/cars/

GET http://api.coolcars.io/cars/sell/{id}

GET http://api.coolcars.io/cars/buy/{id}

POST http://api.coolcars.io/cars/trade
{
  "to ": {
    "make": "chevrolet",
    "model": "Silverado 3500",
    "year": 2004,
    "vin": "1GCJK33104F173427"
  },
  "from": {
    "make": "honda",
    "model": "civic",
    "year": 1990,
    "vin": "2HGED6349LH506746"
  }
}
```

Web Service vs Microservices

- Web services are applications that use web protocols
  - Usually REST but can also include other technologies like SOAP
  - Defined by how clients interact with them
- Microservices are an architectural pattern
  - Can use any technology, not just web protocols
  - Defined by its structure and behavior
- Microservices are often built initially as web services because the technology is easy to deploy as a microservice
  - The 12 factor app principles are guidelines for converting web services to microservices



Both architectures can be implemented as a web service

HTTP Methods

- HTTP Operations: GET, PUT, POST, DELETE
- GET
  - Retrieve data.
  - Should not change state (idempotent, safe).
  - Example: GET /products/123 gets product with ID 123
  - Example: GET/products gets all the products
- POST
  - Create new resources.
  - Example: POST /products with JSON body.
- PUT
  - Replace an existing resource completely.
  - Example: PUT /products/123 with full resource data.
- DELETE
  - Remove a resource.
  - Example: DELETE /products/123

Message Architecture

- Anatomy of an HTTP request
- Every HTTP request has three main parts:
- Start line (request line)
  - Contains the HTTP method, the URL, and the protocol version.
  - Example: GET /users/42 HTTP/1.1
- Headers
  - Provide metadata about the request
  - Common headers:
    - Content-Type: type of data in body Example: application/json).
    - Authorization: authentication info. Example: Authorization: Bearer <token>
    - Accept: response format client expects. Example: Accept: application/json

Message Architecture

- Body (payload)
  - Optional, used for methods like POST, PUT, PATCH
  - Contains the data being sent in the body of the request, usually JSON or XML
  - Example JSON payload:
    - { "name": "Alice", "email": "alice@example.com"}
  - Request headers examples:
    - Authorization: Bearer <token>: Authentication
    - Accept: application/json: Client wants JSON format
    - User-Agent: PostmanRuntime/7.36.0: Identifies client making the request
  - Response headers examples:
    - Content-Type: application/json: Response format
    - Location: /users/42:Where the newly created resource lives (after POST)
    - ETag: "abc123": Unique identifier for resource version (used for caching)

Return Codes

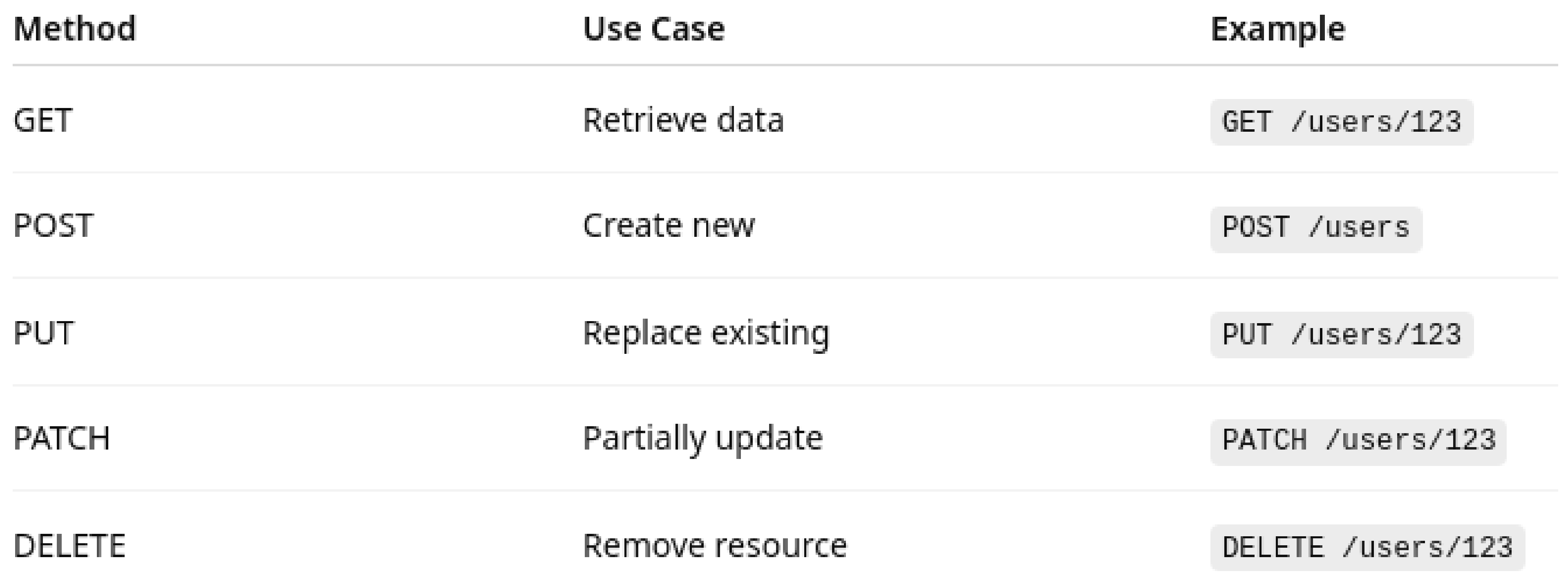
- Common status codes
- Returned by the actual application when processing the request
  - 2xx – Success
    - 200 OK: Request succeeded.
    - 201 Created: Resource created (use with POST).
    - 204 No Content: Operation succeeded, no response body.
  - 4xx – Client Errors
    - 400 Bad Request: Invalid input.
    - 401 Unauthorized: Authentication required.
    - 404 Not Found → Resource doesn't exist.
  - 5xx – Server Errors
    - 500 Internal Server Error: Generic server crash.
    - 503 Service Unavailable: Server is down/overloaded.

Designing a RESTful Interface

- Identify resources (nouns, not verbs)
  - REST revolves around resources, which are things in your system (like users, orders, products).
  - Resources should be represented as nouns in URIs
    - Use /user instead of /getUser
    - Use /order instead of /doOrderProcessing
    - Use /product instead of /createNewProduct
- Use URIs to show hierarchy
  - Resources may have relationships.
  - Use nested URIs to express parent-child relations.
    - /users/123/orders – get orders for user 123
    - /users/123/orders/567 – get order 567 for user 123
    - *Don’t do this: /getOrdersForUser?id=123

Designing a RESTful Interface

- Follow HTTP method semantics



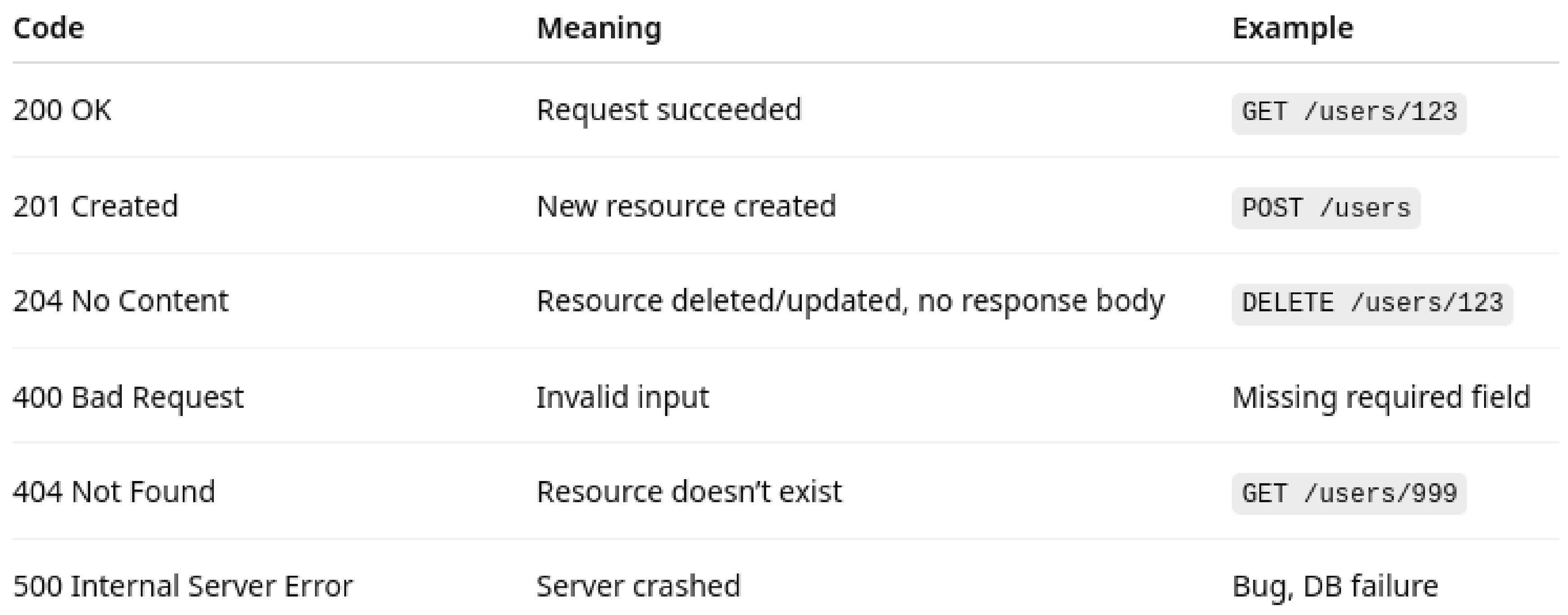
Method	Use Case	Example
GET	Retrieve data	`GET /users/123`
POST	Create new	`POST /users`
PUT	Replace existing	`PUT /users/123`
PATCH	Partially update	`PATCH /users/123`
DELETE	Remove resource	`DELETE /users/123`

Designing a RESTful Interface

- Use query parameters for filtering, sorting, and pagination
  - Use query parameters when returning collections of resources.
    - *Filtering: GET /products?category=books&author=asimov*
    - *Sorting: GET /products?sort=price&order=asc*
    - *Pagination: GET /users?page=2&limit=20*
- Other considerations
  - Consistency: Use consistent naming (/users vs /customers).
  - Plural vs. Singular: Use plural (/users) for collections.
  - Statelessness: Don’t rely on server sessions; each request should be independent.
  - Error Handling: Always include clear error messages.
  - Hypermedia (HATEOAS): Advanced design where responses include links to related actions.

Designing a RESTful Interface

- Provide Meaningful Status Codes
  - Status codes help clients understand results without parsing the whole response
  - Always accompany errors with a JSON error body: { "error": "User not found" }



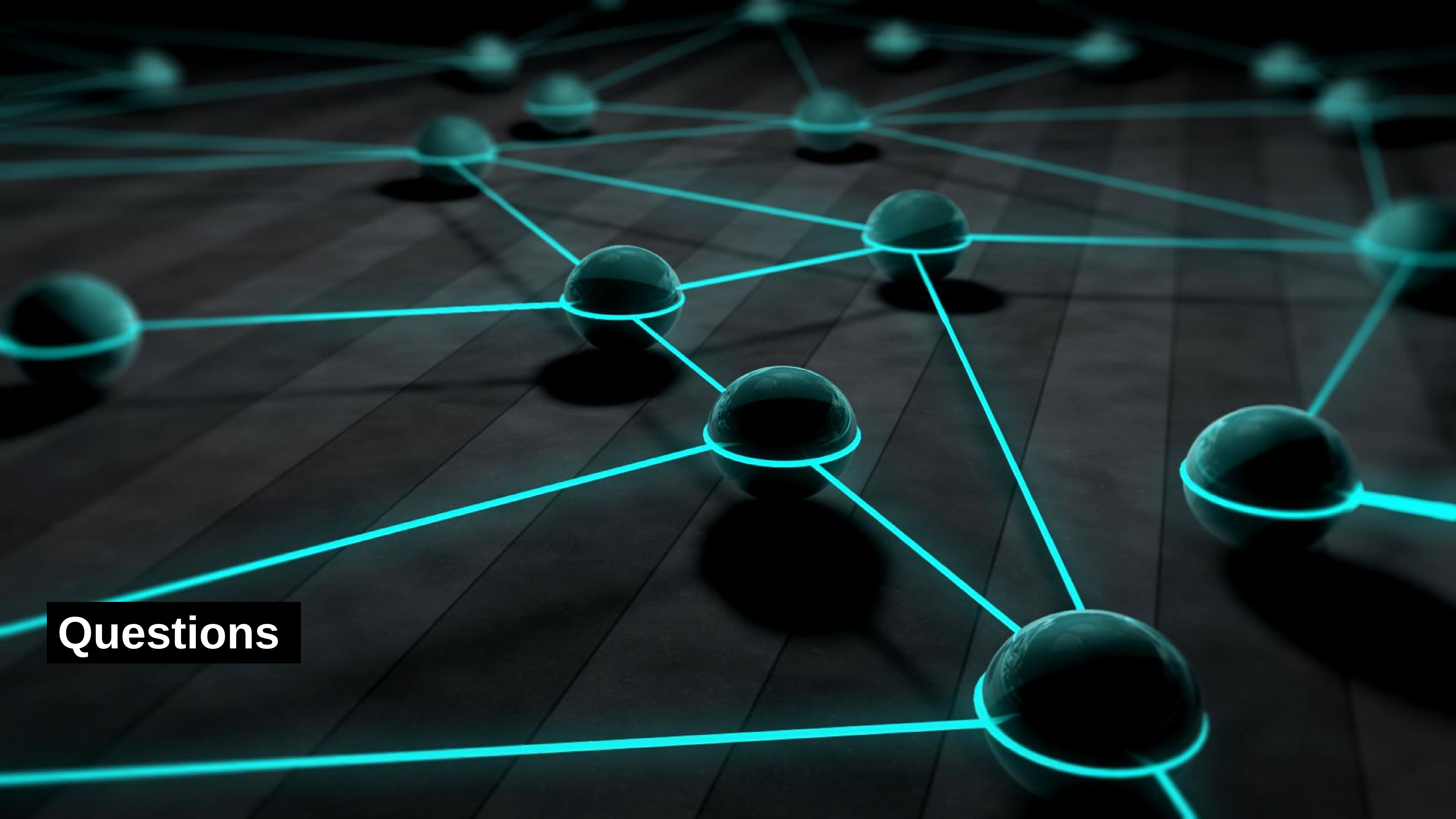
Code	Meaning	Example
200 OK	Request succeeded	`GET /users/123`
201 Created	New resource created	`POST /users`
204 No Content	Resource deleted/updated, no response body	`DELETE /users/123`
400 Bad Request	Invalid input	Missing required field
404 Not Found	Resource doesn’t exist	`GET /users/999`
500 Internal Server Error	Server crashed	Bug, DB failure

REST vs. gRPC

- gRPC (Google Remote Procedure Call)
  - A modern, high-performance framework developed by Google
  - Data format: protocol buffers (binary, compact)
  - Communication style: remote procedure calls (functions over the network)
  - Strengths:
    - Very fast and efficient (binary data)
    - Strongly typed with automatic code generation
    - Supports client-server bidirectional streaming
  - Weaknesses:
    - Harder to debug manually (binary payloads are not human-readable)
    - Requires more tooling and setup
  - Common use cases:
    - Microservices in cloud environments
    - Real-time apps (video, chat, IoT)

REST versus gRPC

- Rest and gRPC are the two most popular API protocols
- REST (Representational State Transfer)
  - An architectural style that uses HTTP and resource-based URIs
  - Data format: typically JSON (but can use XML, YAML)
  - Strengths:
    - Human-readable, simple, widely adopted
    - Stateless and scalable
  - Easy for web and mobile clients
  - Weaknesses:
    - Can be less efficient (JSON is verbose)
    - Limited in real-time streaming use cases
  - Common Use Cases:
    - Public APIs (GitHub, Twitter, Spotify)
    - Web/mobile applications



Questions