**Representational State Transfer (REST and RESTful APIs)**

Table of Contents

[History of distributed computing 1](#_Toc31635864)

[Fundamentals of HTTP and REST 2](#_Toc31635865)

[HTTP 2](#_Toc31635866)

[REST 3](#_Toc31635867)

[Design a RESTful API 3](#_Toc31635868)

[Designing Results 5](#_Toc31635869)

[Designing Collections 5](#_Toc31635870)

[Hypermedia 6](#_Toc31635871)

[Handling more complex scenarios 6](#_Toc31635872)

[Designing Associations 6](#_Toc31635873)

[Paging 7](#_Toc31635874)

[Error handling 7](#_Toc31635875)

[Caching 7](#_Toc31635876)

[Functional APIs 7](#_Toc31635877)

[Async APIS 8](#_Toc31635878)

[Versioning 8](#_Toc31635879)

[Security 9](#_Toc31635880)

# History of distributed computing

* 1970s to 80s: Dawn of Distributed Computing
  + Remote Procedure Call (RPC)
  + Messaging
  + Queuing
* 1980s to 90s: Object-Oriented APIs
  + COM/DCOM
  + COBRA
  + JAVA/RMI
* 2000s: Web-Based APIs
  + XMLHTTP: Early component inside browsers. Browser to server communication.
  + REST: Really didn’t get large adoption until later. Represents a great portion of the working being done.
  + SOAP: Fallen out of favor lately
* 2020s: More recently we are seeing these technologies, but they are segmented for specific use cases.
  + GraphQL
  + gRPC

# Fundamentals of HTTP and REST

## HTTP

**How does HTTP work?**

At the barebones of how HTTP works, it is simply a single request out to a server, waiting for a response, getting the response and processing the response. Client makes a request to a remote server. The request is simply a text document comprised of three (3) pieces of information; verb, headers, and content that is optionally asked for. The server gets the request and can reject the request or accept and send back the response. Request and response are two individual calls. Each network call is short lived. There can be many calls in parallel with each other. The response is simply a text document comprised of three (3) pieces of information; status code, headers, and content a.k.a. the payload. The server is stateless.

**The request deconstructed**

**Verb: Action to perform on the server**

* GET: Request resource
* POST: Create resource
* PUT: Update resource that already exists on the server
* PATCH: Update partial resource (e.g. please update the address of the customer)
* DELETE: Delete resource that already exists on the server
* More methods a.k.a. verbs…

**Headers: Metadata about the request**

* Content Type: The format of the content
* Content Length: Size of the content
* Authorization: Who’s making the call
* Accept: What type(s) can accept
* Cookies: Passenger data in the request
* More headers…

**Content**

* HTML, CSS, JavaScript, XML, JSON
* Content is not valid with some verbs
* Information to help fulfill request
* Binary and blobs common (e.g. .jpg)

**The response deconstructed**

**Status code: A number (in ranges) that represents the level of operational success.**

* 100-199: Informational
* 200-299: Success
* 300-399: Redirection
* 400-499: Client errors
* 500-599: Server errors

**Headers: Metadata about the response**

* Content Type: The format of the content
* Content Length: Size of the content
* Expires: When to consider stale
* Cookies: Passenger data in the request

**Content**

* HTML, CSS, JavaScript, XML, JSON
* Binary and blobs common (e.g. .jpg)
* RESTful APIs often have their own types

## REST

**What is REST (Representational State Transfer)?**

The idea is to have transfer of data or state to be representational of the kinds of messages that you want to use.

**Concepts**

* Separation of client and server
* Server requests are stateless
* Cacheable requests
* Uniform interface or URI
* Idempotent: operation that can be applied multiple times without changing the result
  + POST is not idempotent

**Problems**

* Too difficult to be qualified as “REST”
* Dogma of REST vs. Pragmatism
  + Structured architectural style
  + The need to be productive

# Design a RESTful API

**Concerns of poor design**

* Can’t fix a RESTful API after publishing; clients are reliant on the design
* Too easy to add ad-hoc endpoints for solving individual use cases
* Helps to understand the requirements first then build it; you want to meet the requirements, not all RESTful APIs are simply data access

**Benefits of good design**

* REST APIs can mature and evolve over time; they should not change in large ways
* Better understanding of what to include or exclude when designing results
* Limit exposure of sensitive data; understanding safety and risk

**RESTful APIs have several parts**

* URIs: paths to resources on a server (e.g. api.yoursever.com/users)
  + Query strings
    - For non-data elements: can be added (optional) to end of URIs to indicate things (e.g. format, sorting, searching, etc.)
    - None-resource properties
    - Examples
      * /sites?sort=name
      * /sites?page=1
      * /sites?format=json
  + Identifiers
    - Access individual items in a resource collection
    - Use unique identifiers; point to a specific resource
    - Does not have to be a ‘primary key’ – could be a proxy key (something that identified the individual site but didn’t necessarily expose the implantation of a data store)
    - What you design for the unique identifier is up to you but the further you can get away from needing to have primary keys the better design your REST API normally is - you want the benefits of uniqueness and developer readability.
    - Examples
      * /sites
      * /sites/1
      * /sites/stone-henge
      * /sites/uk-101
* Endpoints: Some container that holds resources the user may or may not want and the developer wants access to
  + Nouns are good (e.g. /users)
  + Verbs are bad (e.g. /getUsers)
* Resources: A collection of canonical nouns (e.g. people, invoices, payments, products, etc.) that you want to expose through the RESTful API but they can be more complicated than that; think of resources as being something inside of a context… simple or complex object
* Entities: Easy to think of resources as just the entities that some data store holds
* Verbs and URIs; not a requirement that every URI supports every verb
  + Get
    - /customers, GET list
    - /customers/123, GET item
  + POST
    - /customers, Create item
    - /customers/123, Error
  + PUT
    - /customers, Update batch
    - /customers/123, Update item
  + PATCH
  + DELETE
    - /customers, Error
    - /customers/123, Delete item

## Designing Results

* Think about data structures; the use cases for people who want to consume your REST APIs
* Decide on formats
  + Do you need to be able to return in multiple formats?
  + Prefer not to use query strings for formats
    - /api/customer?format=json
  + Abide by certain data types
    - Accept: application/json
  + Return sane default (usually JSON)
    - Content Type: application/json
  + Common formats
    - JSON: application/JSON
    - XML: text/xml
    - JSONP: application/javascript (requires callback parameter)
    - RSS: application/xml+rss
    - ATOM: application/xml+atom
* Be considerate of data mapping
  + Often the format your storing in is not the best version
  + Often a good idea to change the format that’s going to be more useful
* Be considerate of Member names
  + Shouldn’t expose server details (e.g. Ruby, Python, JAVA or .NET)
  + Be platform agnostic
  + Be consistent (e.g. camelCase)

## Designing Collections

* You can return the collection as just an array of results
* Often helpful to return collections in a different way for example when limiting the results a user can get at
  + How many results returned
  + Next and previous pages

## Hypermedia

* An idea that REST wants you to do
* Useful in some cases and not useful in other cases
* Allows/wants results to be self-describing
* Allows programmatic navigation
* Adds complexity; you don’t always need to include hypermedia but instead use it as a tool how you want to better serve your clients
* Hypermedia can be helpful, but pragmatism means that most projects don’t need it; Error on pragmatism vs. dogma

The idea here is to say we want there to be links or other things that describe our objects so that we can use software to navigate to these different things more automatically or simply without having to figure out what they are.

* Example
  + “\_links”: { “self”: “/api/site/1”, “region”: “/api/region/us”, “relatedSites”: “/api/region/us/sites” }

# Handling more complex scenarios

## Designing Associations

Sub objects for existing APIs. The idea behind associations in these simple cases is to allow the navigation of the URL to imply ownership. Building up relations (hierarchy) between your resources.

* /api/customers/123/invoices
* /api/games/halo-3/rates
* /api/invoices/2003-01-24/payments

In the same way, both of these URIs, one that returns all or individual invoices, should also return the same type of shape that the invoices within a specific customer would return. The difference here is the scope, invoices inside an individual customer should return a collection, but only invoices that belong to that customer where the other URI or top level URI would return invoices across different customers.

* /api/customers/123/invoices
* /api/invoices

In the same way, URI endpoints can have multiple associations. So even though we have invoices for a specific customer, we could also have payments or shipments for a specific customer. There’s not a limitation that an endpoint for a specific customer or resource has to have have only one (1) association to it. It’s pretty common for it to have multiple associations.

* /api/customers/123/invoices
* /api/customers/123/payments
* /api/customers/123/shipments

These associations shouldn’t be confused with search queries.

* /api/customers?st=GA
* /api/customers?st=GA&salesid=144
* /api/customers?hasOpenOrders=true

## Paging

* Lists should support paging
* Query strings are commonly used
  + /api/sites?page=1&page\_size=25
* Use wrappers to imply paging
  + { totalResults: 255, nextPage: “/api/sites?page=5”, prevPage: “/api/sites?page=3”, results: […] }

## Error handling

* Return the object with the error info
  + Not just status codes
  + Communicate errors
  + Help the user recover
  + Use this wherever you think it’s helpful to the user but doesn’t compromise security
    - 400 Bad Request { error: “Failed to supply id” }
  + Not necessary for obvious errors (e.g. 404 Not Found)

## Caching

* Required to be truly RESTful
* Service-side caching is good but not really what REST is talking about
* Use HTTP for caching mechanism
  + On request GET version=last\_xyz then response should be “304 Not Modified”
    - Versioning of individual instances of data or resources that your returning form servers.
  + Another way it can do this is on request, look at header called “If-Match” then response could be “412 Precondition Failed” if no match
  + A great way to handle caching is Entity Tags (ETags)
    - Well-known and well-supported; best ways to handle caching scenarios because they are efficient ways to deal with updates or frequent GETS without having to handle the larger payloads.
    - Strong caching support means I can go ahead and store it in my rich client or on my phone app and 4 weeks later I should be able to send it back with changes and this ETag should be able to be constructed by the server to see if there’s a change
    - Weak caching support is for things that are just very short lived. These ETags are returned in the response and we can see ETag as a header type with some identifier and start with “W/”

## Functional APIs

* One-off operational APIs that you want to support that aren’t truly RESTful.
  + Restart a machine, recalculate totals
* Be pragmatic
* Make sure these are well-documented
* Should be the exception rather than the rule
  + /api/calculateTax?state=GA&total=149.99
  + /api/restartServer?isColdBoot=true

## Async APIS

* Some APIs aren’t RESTful in nature
* Need long-life polling
* Non-REST solutions are useful

Aysnc API solutions to consider

* Comet
* gRPC (Google)
* SignalR – (.NET)
* Socket.IO
* More…

# Versioning

Should you version your API? Versioning is critical to a mature, long lasting API. Deciding on which versioning scheme means fitting requirements. Content versioning is complex, but could benefit long lived APIs.

* Once you publish, it’s set in stone
* Users (internal and external clients) rely on the API not changing
* Requirements will change
  + Evolve the API without breaking clinets
  + API version isn’t product versioning (you can decouple)
  + Need to support both new and old versions and have a story about deprecation over time
  + Side-by-side deployment usually isn’t feasible

URI versioning strategies

* Serving your clients, not your developers are yourself
* URI path
  + <https://foo.org/api/v2/customer>
  + Very clear to clients where the version is handled
  + Every version needs to change URIs, can be brittle
* Query sting
  + <https://foo.org/api/customers?v=2.0>
  + Versioning is optionally included (can use default version)
  + Too easy for clients to miss needing the version
* Header
  + X-Version: 2.0
  + Separates versioning from the rest of the API
  + Requires more sophisticated developer to manipulate headers
* Accept Header
  + Accept: application/json; version=2.0
  + No need to create your own custom header
  + Even less discoverable than query strings
* Content Type (custom content type)
  + Content-Type: application/vnd.yourapp.camp.v1+json
  + Accept: application/vnd.yourapp.camp.v1+json
  + Can version the payload as well as the API call itself
  + Requires a lot more development and maturity to create and maintain

# Security

**Locking down your API**

* Why
  + Using private or personalized data
  + Sending sensitive data across the ‘wire’
  + Using credentials of any kind
  + Trying to protect against overuse of your servers
  + Designing an API without considering security is a big mistake
  + Security requirements will affect what data you’re willing to expose
  + Be pragmatic with security; don’t assume every app needs a vault
* Threat model
  + Transport security
    - In-transit
    - Eavesdroppers (packet sniffers, etc.)
    - Man-in-the-middle attacks
  + Access to actual boxes
    - Server infrastructure
    - Hackers/personnel (intrusion and physical security)
  + Access to data
    - Users/hackers
* APIs and security
* Cross domain security
  + By default, not allowed
  + Prevent cross-site scripting from occurring
  + Most platforms don’t allow a separate domain to call an API by default; this is a browser limitation to prevent malicious code from running in the browser that might call the separate API
  + Public
    - Should allow
    - Cross Origin Resource Sharing (CORS)
      * Allows finely grained control
      * Domain, resource, and verb control
      * Only limits browser, not app
      * Most platforms support CORS
    - How does CORS work?
      * Cross-Origin request
      * Browser requests access
      * Server replies with rules
      * Browser issues with CORS header
  + Private
    - Consider for partners
* Authentication
  + Who you are
  + Information to determine identity
  + Credentials/claims
  + Types
    - App Authentication
      * Identifying an app for your API
      * Authenticating as the developer
      * AppID + Key is typical
    - User Authentication
      * Identifying as a User
  + Methods (primary ways APIs are secured)
    - Cookies
      * Easiest and common
      * Subject to request forgery
      * Security ramifications are pretty high
    - Basic Auth
      * Easy to implement
      * Not secure, unless enforcing SSL
      * Not secure, passing credentials on each and every request; increases surface area of attached the more you pass the higher the chance of someone intercepting it
    - Token Auth
      * Most commonly used
      * Mix of secure and simple
      * Industry standard tokens are easy; middle ware support creating and validating tokens on most platforms
      * Expire much faster (typically 5-20 minutes)
      * Diagram
        + Client sends credentials to server
        + Server returns token (just a series of characters)
        + Client never needs to decode
        + Client includes token on subsequent calls
        + Server validates token and returns response
        + Repeats until token timeouts
      * JSON Web Tokens (JWTs)
        + Industry standard
        + Small packets for small amount of data that includes information about the User
        + Self-contained, small and complete

User information

Claims

Validation signature

Other information

* + - OAuth
      * Use trusted third-party to identify Users
      * The app never gets the credentials so you’re not leaking your username and password or credentials to another party
      * User authenticates with the third party
      * User uses another token to confirm identity
      * Lowers the surface area in many ways
* Authorization
  + What you can do
  + Rules about rights (e.g. Roles, rights)
* Security considerations during design
* Types of API security
* Bation server
* Rate limit