REST API Fundamentals - REST Concepts

An API, or Application Programming Interface, is a set of rules and protocols that allows different software applications to communicate with each other.

REST, which stands for **Representational State Transfer,** is an architectural style for designing networked applications.

- approach for building web services and APIs (Application Programming Interfaces)
- Rest is an architectural style, not a standard
- Not an implementation technology
- **RESTFul** means that it complies with the REST architectural style. API follows the best practices outlined by REST to create simple, scalable, and interpersonal communication between clients and servers over the internet

Ecosia Example - Searching suggestions in search bar: Query is sent to ecosia website, query the REST API, which searches the query in the index, puts the data into JSON data structure, and returns the JSON data back to the webpage to populate the drop down search suggestions menu.

Why REST

- Call from JavaScript in web pages, for interactive and functional web pages
- Call from mobile apps to get data as mobile applications are often developed in JavaScript so highly compatible
- Use extensively for mobile apps to populate graphical interfaces
- Use REST for general integration of applications - integrating UI components with data access components
    - Wide support for HTTP in programming languages
    - JSON is a text-based transfer format
    - Wide support for JSON in programming languages - programming languages allow reading in of JSON files and convert to a structure that is native. Many programming languages provide built-in support for parsing and generating JSON data, simplifying data handling in RESTful applications
    - JSON is natively supported in JavaScript
    - Wide support for development of REST APIs in many languages. Developers can create RESTful services using a wide range of languages and frameworks, allowing them to choose the technology stack that best suits their project requirements

CRUD

- stands for Create, Read, Update, and Delete, which make up these four fundamental operations.

- CRUD operations apply to relational database management systems (such as PostgreSQL or SQL Server) and the more recent NoSQL databases (such as MongoDB)

https://www.crowdstrike.com/cybersecurity-101/observability/crud-vs-rest/

Constraints of REST

- **Client-server architecture** - Client and server on network communicate using HTTP integration. Client is the user interface such as web page or mobile app, server waits for HTTP requests and responds by returning JSON to the client. REST enforces a clear separation between the client and the server. This separation allows for independent development and scalability of client and server components
- **Stateless** - REST is designed to be stateless, meaning no client specific state information stored on the server side (HTTP is inherently stateless) Each request from a client to a server must contain all the information necessary to understand and process the request. The server should not rely on any information from previous requests. Stateless communication ensures that each request is self-contained and can be processed independently
- **Cacheability** - Clients and intermediaries can cache responses from the server, reducing the need for repeated requests to the server. Responses can explicitly specify whether they are cacheable or not. Caching can occur at various levels, including client-side browser caches, in-memory caches, or Content Delivery Networks (CDNs). Efficient caching improves the overall scalability and performance of the system
- **Layered system -** REST allows for a layered architecture, client does not need to know whether its communicating directly to the REST API, or communicating through an intermediary. Intermediaries like proxies, load balancers, or security layers can be introduced between the client and server. These intermediaries can enhance scalability by distributing load and providing shared caches. Importantly, the client and server do not need to be aware of these intermediaries and how the data being accessed is stored, for instance sql or NoSQL databases, ensuring that the architecture can evolve without affecting the core components. Layering also enables the addition of security mechanisms as a separate layer, enhancing security without modifying client or server code

- **Uniform interface** - The uniform interface constraint is fundamental to RESTful design
    - **Resource identification in request, data format transferred not the same as storage**. Individual resources are identified in requests using URIs (Uniform Resource Identifiers). Resources are conceptually separate from representations returned to the client. Server can send data from its database as HTML, JSON, or XML allowing for multiple representations of the same resource
    - **Resource manipulation through representation** - Representation is the data that is retrieved from service previously. Clients that hold representations of resources, including metadata, have enough information to modify or delete the resource's state. This constraint allows clients to interact with resources without needing to understand the server's internal state.
    - **Self-descriptive messages** - Each message exchanged between the client and server includes enough information to describe how to process the message (primarily done using key elements in JSON structure which are descriptive names of data that correspond to the value). This information often includes media types, which specify how the message should be interpreted.
    - **Hypermedia as the engine of application state** - when content is returned by REST API, content should include information to navigate to other information that may be needed. In a HATEOAS-based REST system, clients initially access a URI (analogous to a website's home page), and from there, they can discover available resources dynamically by following hyperlinks provided by the server. This allows clients to navigate the application without having prior knowledge of the server's structure.

REST Technologies

- Message exchange
  - HTTP
- Describing the location, action and target
  - URIs - uniform resources identifier
  - HTTP methods - GET PUT POST PATCH DELETE
- Representing the data
  - JSON - most common
  - XML or HTML (JSON default but XML and HTML less common)
- Describe the service
  - Swagger/OpenAPI - Document that explains how to use API service

Swagger, also known as the **OpenAPI** Specification, is an open-source framework for documenting and defining RESTful APIs. It provides a standardised way to describe the structure and behaviour of APIs, making it easier for developers to understand, consume, and interact with those APIs.

Swagger/OpenAPI has tools and libraries that support API development and consumption. This includes code generators, validators, documentation generators, and testing tools