# RESTful APIs

Nesli Erdoğmuş

# REST

- REST is an acronym for REpresentational State Transfer.
  - Roy Fielding, 2000
  - It is actually proposes an “architectural style for distributed hypermedia systems, describing the software engineering principles guiding REST and the interaction constraints chosen to retain those principles, while contrasting them to the constraints of other architectural styles.”
  - It has its guiding principles and constraints.
  - These principles must be satisfied if a service interface needs to be referred to as **RESTful**.
- A Web API (or Web Service) conforming to the REST architectural style is a REST API.

https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

# Guiding principles of REST

1. Uniform Interface:
    - Simply applying the **principle of generality** to the components interface, with the following constraints:
        i. **Identification of resources** – The interface must uniquely identify each resource involved in the interaction between the client and the server.
        ii. **Manipulation of resources through representations** – The resources should have uniform representations in the server response. API consumers should use these representations to modify the resources state in the server.
        iii. **Self-descriptive messages** – Each resource representation should carry enough information to describe how to process the message. It should also provide information of the additional actions that the client can perform on the resource.
        iv. **Hypermedia as the engine of application state** – The client should have only the initial URI of the application. The client application should dynamically drive all other resources and interactions with the use of hyperlinks.

# Guiding principles of REST

2. Client-Server
    - The client-server design pattern enforces the **separation of concerns**.
    - It separates the user interface concerns (client) from the data storage concerns (server).
        i. the portability of the user interface across multiple platforms
        ii. scalability
    - While the client and the server evolve, we have to make sure that the interface/contract between the client and the server does not break.

# Guiding principles of REST

3. Stateless
    - Statelessness is a restriction that the server does not store any state about the client session on the server-side.
    - Each request from the client to the server must contain all of the necessary information to understand the request.
    - The client is responsible for storing and handling the session related information on its own side.

> Statelessness means that every HTTP request happens in complete isolation.

# Guiding principles of REST

4. Cacheable
    - This constraint requires that a response should implicitly or explicitly label itself as cacheable or non-cacheable.
    - If the response is cacheable, the client application gets the right to reuse the response data later for equivalent requests and a specified period.

> Caching is the ability to store copies of frequently accessed data in several places along the request-response path.

# Guiding principles of REST

5. Layered System
    - The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior.
    - Each component cannot see beyond the immediate layer they are interacting with.
    - Servers can provide part of features delivered to the client in the form of code, and the client only needs to execute the code.

# Guiding principles of REST

6. Code on Demand (Optional)
    - Client functionality is extended by downloading and executing code in the form of applets or scripts.
    - It is optional because:
        i. Using COD reduces visibility.
        ii. Not every API needs this kind of flexibility.

For example, a web browser acts like a REST client and the server passes HTML content that the browser renders. At the server side, there is some sort of server-side language which is performing some logical work at the server side. But if we want to add some logic which will work in the browser then we (as server-side developers) will have to send some JavaScript code to the client side and the browser and then execute that JavaScript ...

# In summary

1. The uniform interface constraint defines the interface between clients and servers.
2. The necessary state to handle the request is contained within the request itself, whether as part of the URI, query-string parameters, body, or headers.
3. Clients can cache responses (if defined as cacheable).
4. Clients are not concerned with data storage and servers are not concerned with the user interface or user state
5. A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way.
6. Servers are able to temporarily extend or customize the functionality of a client by transferring logic to it that it can execute.

# What is a Resource?

- The key abstraction of information in REST is a resource. It is the primary data representation.
- The state of the resource, at any particular time, is known as the **resource representation**.
- The resource representations are consist of:
  - the data
  - the metadata describing the data
  - and the hypermedia links that can help the clients in transition to the next desired state
- REST uses resource identifiers to identify each resource involved in the interactions between the client and the server components.

# What is a Resource?

- Singleton and Collection Resources
  - For example, “customers” is a collection resource and “customer” is a singleton resource (in a banking domain).
- Collection and Sub-collection Resources
  - For example, sub-collection resource “accounts” of a particular “customer”
- URI: REST APIs use Uniform Resource Identifiers (URIs) to address resources.
  - When resources are named well, an API is intuitive and easy to use.

```
/customers/{customerId}
/customers/{customerId}/accounts
/customers/{customerId}/accounts/{accountId}
```

# What is a Resource?

- Any information that we can name can be a resource.
- How to name them?
  - Use nouns to represent resources
  - Divide the resource archetypes into four categories: **document, collection, store, controller**; and use use its naming convention consistently
  - Document: a single resource inside resource collection
    - Use “singular” name
  - Collection: a server-managed directory of resources
    - Use the “plural” name
  - Store: a client-managed resource repository
    - Use the “plural” name
  - Controller: a procedural concept
    - Use “verb”

# What is a Resource?

```
http://api.example.com/device-management/managed-devices/{device-id}
http://api.example.com/user-management/users/{id}
http://api.example.com/user-management/users/admin

http://api.example.com/device-management/managed-devices
http://api.example.com/user-management/users
http://api.example.com/user-management/users/{id}/accounts

http://api.example.com/song-management/users/{id}/playlists

http://api.example.com/cart-management/users/{id}/cart/checkout
http://api.example.com/song-management/users/{id}/playlist/play
```

# What is a Resource?

- Any information that we can name can be a resource.
- How to name them?
  - Use forward slash (/) to indicate hierarchical relationships
  - Do not use trailing forward slash (/) in URIs
  - Use hyphens (-) to improve the readability of URIs
  - Do not use underscores ( _ )
  - Use lowercase letters in URIs (when convenient)
  - Do not use file extensions
  - Never use CRUD function names in URIs
  - Use query component to filter URI collection

# Resource Representations

- The data format of a representation is known as a media type.
- The media type identifies a specification that defines how a representation is to be processed.
- They should be self-descriptive: the client does not need to know if a resource is an employee or a device. It should act based on the media type associated with the resource.
    - In practice, we create lots of custom media types – usually one media type associated with one resource.
    - Every media type defines a default processing model.
        - For example, HTML defines a rendering process for hypertext and the browser behavior around each element.

# Resource Representations

- The resources have to be decoupled from their representation so that clients can access the content in various formats, such as HTML, XML, plain text, PDF, JPEG, JSON, and others.
- The clients and servers exchange representations of resources by using a standardized interface and protocol.
- Typically HTTP is the most used protocol, but REST does not mandate it.
- Metadata about the resource is made available and used to control caching, detect transmission errors, negotiate the appropriate representation format, and perform authentication or access control.

# Resource Methods

- The resources are acted upon by using a set of simple, well-defined operations.
- Resource methods are used to perform the desired transition between two states of any resource.
- Ideally, everything needed to transition the resource state shall be part of the resource representation – including all the supported methods and what form they will leave the representation.

# REST, HTTP, CRUD

- Even though REST is heavily influenced by the Web-Technology, in theory the REST architecture style is not bound to HTTP.
  - Resource methods are not HTTP methods
- Roy Fielding, in his dissertation, has nowhere mentioned any implementation direction – including any protocol preference or even HTTP.
- However, HTTP is the only relevant instance of the REST.
  - REST is implemented by using HTTP

# REST, HTTP, CRUD

Create, Read, Update, and Delete — CRUD — are the four major functions for interacting with database applications. CRUD functions often play a role in web-based REST APIs, where they map (albeit poorly) to the HTTP methods GET, POST, DELETE, PUT, and PATCH. Importantly, REST APIs can still expose functionality not corresponding to CRUD, so long as they use the appropriate HTTP method.

# An example

https://opentdb.com/

# How to Design a REST API

- Learning REST in pieces is one thing while applying all these concepts to real application development is completely another challenge.
  1. **Identify the Resources – Object Modeling:** Identify the objects that will be presented as resources.
  2. **Create Model URIs:** Focusing on the relationship between resources and their sub-resources, decide the resource URIs. These resource URIs are endpoints for APIs.
  3. **Determine Resource Representations:** Most representations are defined in either XML or JSON format.
  4. **Assign HTTP Methods:** Decide all the applications' possible operations and map those operations to the resource URIs.
  5. **Work on Other Aspects:** Design solutions for other aspects such as logging, security, discovery, etc.