November 2020

HTTP RESTFUL API DESIGN USING FREEFORM URL TO INTERACT WITH HIERARCHICAL JSON DATA STORES

Santosh Nair

Sachin Yerrawar

HTTP RESTFUL API DESIGN USING FREEFORM URL TO INTERACT WITH HIERARCHICAL JSON DATA STORES

AUTHORS:
Santosh Nair
Sachin Yerrawar

ABSTRACT

Techniques are presented herein that support a Hypertext Transfer Protocol (HTTP) Representational State Transfer (RESTful) Application Programming Interface (API) employing a freeform Uniform Resource Locator (URL) to interact with a hierarchical JavaScript Object Notation (JSON) data store. Additionally, aspects of the techniques presented herein incorporate a smart caching strategy to, for example, optimize cache size, support reads and writes to an underlying database, and reduce cache misses.

DETAILED DESCRIPTION

Websites comprising personalized user interfaces are plentiful. Examples include, among others, customer portals, shopping websites, social media sites, ticketing websites, etc. Additionally, an enterprise website or portal which offers a personalized user experience needs to save and retrieve various user data relevant to the website experience. In short, such websites are found anywhere users can login and experience a site according to their data and preferences.

Such websites are evolving from monolithic model-view-controller (MVC) models to become microservice-based and cloud-based distributed solutions. As a result, storing and accessing user preferences and personalized settings may be duplicated in distributed components. For example, different modules may be developed and maintained by different teams, resulting in, possibly among other things, such data being saved in non-uniform formats thus limiting the data's reuse or aggregation across modules.

To address these types of challenges techniques are presented herein that support a freeform Uniform Resource Identifier (URI) based, RESTful web service supporting hierarchical JSON data storage and retrieval.

Aspects of the techniques presented herein expose a dynamic and freeform "path" in a HTTP URL to store JSON data. The saved data in effect aggregates into a single virtual JSON artifact, per user or globally. Portions of the virtual JSON structure may then be retrieved using the corresponding freeform paths. Such a capability becomes useful for distributed cloud-based applications as they interact with a shared and dynamic data model for user and global preferences and shared settings. Additionally, aspects of the techniques presented herein incorporate a smart caching strategy to provide, for example, faster API responses by caching the most frequently used data entries for selected users who may be active at any time.

Aspects of the techniques presented herein include a number of unique features, various of which are briefly described below.

A first unique feature of the techniques presented herein comprises a freeform path in a HTTP web service having the URL pattern `https://<host>/<context-path>/user/<path>`. Data at the `/user/*` path is saved per user Id per path where a user is uniquely identified through an authentication token that is included in the HTTP request.

A second unique feature of the techniques presented herein comprises another URL pattern that saves data globally (i.e., shared across all users) -- `https://<host>/<context-path>/global/<path>`. Data at the `/global/*` path is saved commonly across all users.

Under a third unique feature of the techniques presented herein, a web service stores and retrieves the URLs and the associated JSON data from a relational database or from a non-relational (e.g., NoSQL) data store. Data is stored along with metadata information including, for example, a timestamp and an owner identifier (e.g., a user Id or "global").

Under a fourth unique feature of the techniques presented herein, a web service provides the ability to store and retrieve partial data using nested URL paths to traverse the JSON data. Illustrative examples of this feature are presented below.

A fifth unique feature of the techniques presented herein comprises a caching strategy in support of enhanced performance. Of interest and note in connection with the caching strategy are, for example:

1. The use of write-through and a lazy-loaded cache for faster API response.

2. Each cache entry saves metadata information including, possibly among other things, a last accessed timestamp, a created timestamp, an identifier of the user who owns the cached data, etc.
3. Cache logic that detects which paths are most requested (e.g., active paths), and by which users (e.g., active users), and then reloads that data into container memory every few seconds.
4. At service startup the cache is 'warmed up' to load the most recently used paths using the strategy that was described in Item #3, above.
5. Intelligence behind the 'warm up' process by which the cache is only 'warmed up' if there are active users employing the service. If there are no active users, then the 'warm up' process may sleep to save Central Processing Unit (CPU) cycles.

Under a sixth unique feature of the techniques presented herein, using the stored data a web service exposes API endpoints to provide aggregated information for analytics such as, for example, most active users, user last login, most and least paths used, etc.

A seventh unique feature of the techniques presented herein, noted briefly above, comprises data storage compatibility with relational and non-relational (e.g., NoSQL) databases. Of interest and note in connection with such data storage compatibility are, for example:

1. Each datum that is saved is saved as a record with a unique identifier that is formed from the hash value of the path. That is, there is a key (i.e., a path) and a value (i.e., the data itself).
2. Multiple save requests to different paths (including a path and, possibly, one or more subpaths) will create separate records. In a relational database this is stored as a "row" in a "table". In a non-relational (e.g., NoSQL) database, this is stored as a "document" in a "collection". If a save request overwrites a path for which records of subpaths exist in the database, those subpaths records are deleted as part of the request.
3. When a read request is received (a) all the records for that path and subpath are read (from a database or from the cache), (b) sorted based on the timestamp of each record, and (c) processed linearly and consolidated into a single JSON

structure by the service. The sorting ensures that older saves are overwritten by the newer data in the final output.

Aspects of the techniques presented herein employ, among other things, standard web API features. For example:

1. A web service is authenticated using JSON Web Tokens (JWT) for Open Authorization Protocol (OAuth) 2.0. See, for example, https://oauth.net/2/jwt.
2. Following RESTful design practices, HTTP methods are used to determine an action that is to be performed. For example, a GET is used to read a JSON artifact at a path, a POST is used to write a JSON artifact to a path, and a DELETE is used to delete data at a path.

Elements of the techniques presented herein may be further explicated through the exemplary design illustration that is presented in Figure 1, below.

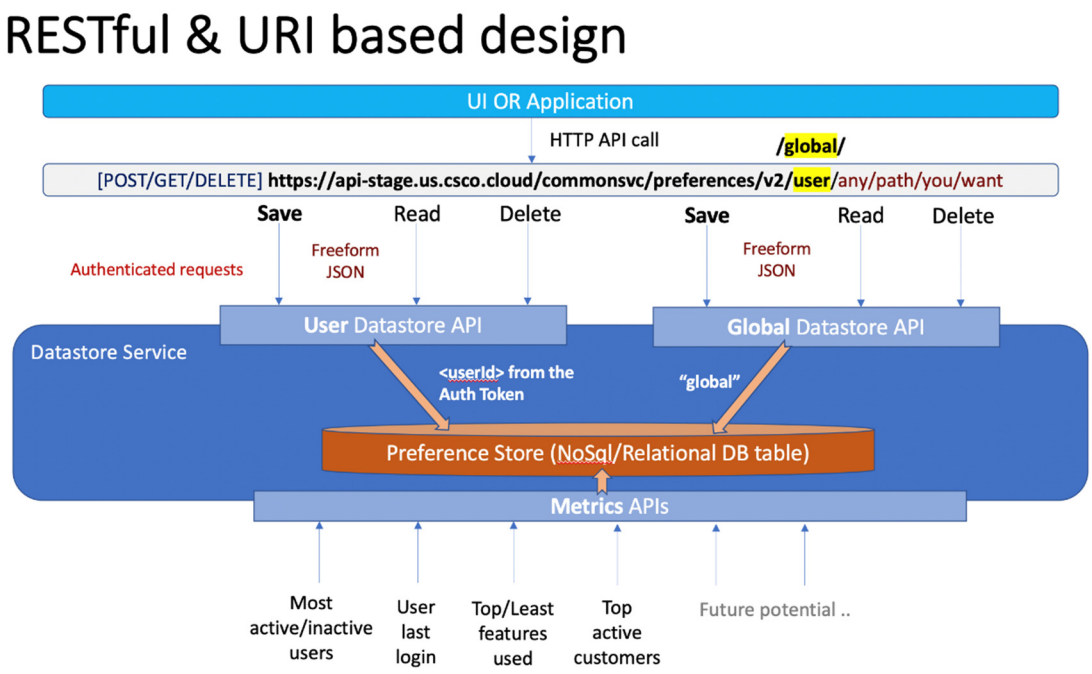

Figure 1: Design Illustration

Aspects of the techniques presented herein may be better understood through the illustrative example that is presented below.