API specification related Design Decisions

This document will summarize the design decisions which have been considered in the Technical Assignment to achieve the 3 sample use cases mentioned below.Each use case is discussed in detail, with considered enhancements (some followed in the assignment, some may not), and at the end of the document it will discuss best practices of Rest API design in related to the assignment.

Use Case 1

Goal to achieve:

A consumer may periodically (every 5 minutes) consume the API to enable it (the consumer) to maintain a copy of the provider API's customers (the API represents the system of record)

Design considerations

As per the use case 1, the consumer is periodically calls the API, to make sure consumers list of Customers are in sync with the Customers in the API. When the number of customers are expanding, and the frequency of the consumer consume the api is high, it will be a big data overhead in the system bandwidth and the response time is also degraded unnecessary. To mitigate that, there are number of ways to design the api.

- Maintain version and lastUpdated fields in 'Customer' entity, which will eventually track the changes made on entities along with the timestamp which can be used to synchronize the local copy of resources with the api.
- In the assignment, this is achieved by the use of content-cacheable trait. Basically, with the use of the 'ETag' and 'If-None-Match' header field. A client that has one or more entities previously obtained from the resource can verify that none of those entities is updated/current by including a list of their associated entity tags in the If-None-Match header field. On a subsequent GET request, consumer requests the resource only if its version (previously returned in the ETag header) is different than the one provided in this header. If the resource has not been modified (it still has the same version), then HTTP status code '304 Not Modified' (with empty body) will be returned instead of the actual resource. This will allow the consumer to synchronize its local copy of 'Customers' with api, in a decent manner, rather than fetching all the resources unnecessary.
- Same as ETag, "Cache-Control' is used to determine whether a response is cacheable, and if so, by whom, and for how long etc. There are many properties to customize e.g. max-age or s-maxage directives.

Use Case 2

Goal to achieve:

A mobile application used by customer service representatives that uses the API to retrieve and update the customers details

Design considerations

A mobile application has a higher sensitivity to number of network trips, latency and data size than a web application. So the API should be designed to limit backend calls and minimize the size of data it will return. Below describe design considerations.

- As already mentioned in use case 1, content-caching trait helps to minimize amount of data being exchanged between mobile application and the api.
- The API should expose specifically fine-grained functionality so it can be invoked independently. We can expose fine-grained services that mobile application can access directly, and add coarse-grained services on top of them to support broader use cases. So the mobile application can choose to call the fine-grained APIs directly or if they need the combined functionality of multiple fine-grained calls they can use the coarse-grained APIs.By introducing nested resources(/customerId), query parameters (access_token) in the assignment, give the flexibility for fine grained function calls and filtering of the collection.
- Mobile clients usually display just a few attributes in a list. So they might not need all attributes of a resource. By giving the API consumer the ability to choose returned fields, as field selection it will also reduce the network traffic and speed up the usage of the API. The 'partial' trait is introduced in the exercise to achieve the same.
- When updating the resources, method 'PUT' handles it by replacing the entire entity, while 'PATCH' only updates the fields that were supplied, leaving the others alone. So this comes very handy, when updating the entities with many properties. Below example further illustrate how 'Patch' will be a wise decision to be used in mobile application to update the resources.

```
PUT /customers/1
{
  "firstName": "Jane",
  "lastName": "Starlin"      // new lastName
}
```

You can accomplish the same using PATCH. That might look like this:

```
PATCH /customers/1
{
  "lastName": "Richie"      // new lastName
}
```

Use Case 3

Goal to achieve:

Simple extension of the API to support future resources such as orders and products.

Design considerations

There are many advantages of reusing patterns across multiple resources and methods. It gives the ability to support future resources without much of a hassle. We can define resource types, extract it from an existing resource and reuse for future resources. For example, if we are trying to represent resources that could be inferred from a business model, it will likely be dealing with the CRUD model. Given a resource, you can **c**reate a new one, **r**etrieve one or all of them and **u**pdate or **d**elete an existing one. In that sense, we can easily identify an existing resource (to be fetched, deleted or updated), a new one (to be added to a collection) and the collection itself (to be retrieved).

- The resource 'Customer' is used to define and parameterize the resource types in this exercise. I have defined 'collection', a list of Customers and 'collection-item', single Customer entity as 2 main resource types which can be easily extend/reuse for future resources like 'Orders' and 'Products'. The characteristics of a collection-type resource can

be defined and then applied to multiple resources. This use of patterns encourages consistency and reduces complexity for servers and clients.

- Another important aspect to stress is that defining and applying a resourceType to a resource doesn't forbid you from overwriting any of the map's elements. As an example, still the GET method is present in both, resource and resource Type. Having said that, it offers the flexibility to redefining something that changes from one resource to other. So this is more or less similar to the very powerful practice in OOP, the 'inheritance'.
- Also reduce the quantity of code you write, while making it more reusable and maintainable.

Best practices and design considerations followed in the assignment

- Use of built in aspects of the HTTP protocol (status codes, headers etc), where appropriate.
- Only Get and Post method allowed in collection level, as 'Delete' operation is risky as it will delete all the customer objects. It is only limited to entity/object level.
- Introduce 'traits', to add more operational flexibility on collection, like filtering,sorting, pagination etc. This can plug on methods where appropriate. (As in get method in the assignment).
- Access_token query parameter in put method, as security authorization is important in altering a collection or record.
- Field 'customerId' is used to identify unique resource (Customer), as firstName,lastName can not be considered as unique.
- Version and lastUpdated fields on Customer to track alterations and for object data synchronization.
- As multiple addresses are required for Customer, Address is maintain as a json object with few address related fields.
- Defined schema (customer) , the schemas and reference them by name and apply that to the corresponding body parameters as well.
- Defining resource types and reuse them.
- Version the API. Versioning helps to iterate faster and prevents invalid requests from hitting updated endpoints. It also helps smooth over any major API version transitions as you can continue to offer old API versions for a period of time.
- Field selection, use of a fields query parameter that takes a comma separated list of fields to include.
- Sorting, pagination, ordering to enhance data related operations.

Some future enhancements

- Give more protection to resources, by adding security schema which RAML supported built-in security scheme types like OAuth 1.0. I haven't explicitly use this feature in the assignment, as 'Customer' does not hold such secured/sensitive data except basic name,address.
- Use of Hateoas, a design principle that hypertext links is used to create a better navigation through the API.