Working Smarter:
RAML, CRUD, Libraries and Resource Types

March 2, 2022, by Matthew Thurmaier – Principal, The Computer Classroom, Inc.

Why Read This Blog Post? (What problem are we solving?)

As people use RAML (the Restful API Modeling Language) to create API specifications, there inevitably is the question of how to define object models (data types) for the various CRUD (create, read, update, delete) endpoints for a given resource. This blog post provides what at The Computer Classroom, Inc. (TCCI) believes Is the best of all worlds.

What (How will we solve the problem)

In this blog, we will show you how to efficiently use Libraries, Resource Types, and most importantly, a consistent taxonomy, to easily define and extend API Specifications using RAML

Review of REST resources

well-define RESTful APIs have *resources* (the URI part of an API) defined based on **nouns** found in user stories. Consider the following two user stories from a local car dealer:

> *US1: As a salesperson, I need to be able to retrieve a list of all cars in our inventory by color, price, make, and model so that I can find a car our customer may want to buy.*

> *US2: As a salesperson, I need to be able to change the status of a specific car in inventory to "SOLD" so that no other sales-person tries to sell it once I have sold it to a customer.*

In these two user stories, we can see the nouns **car**, **salesperson**, and **customer**. These nouns become the *objects* that we wish to manipulate with our RESTful APIs. To do enable this manipulation, RESTful APIs typically have two URIs for each noun / object:

- A collection resource, typically named as a plural of the noun, e.g. /cars, /salespeople, /customers
- An individual member resource, typically the name of the noun followed by "ID", or "_id" or some such, as a URI Parameter (variable). For example, a URI for the member would be /customers/{customerID} (the curly-brackets surrounding *customerID* indicate that it is a **URI-Parameter** not fixed text.

Review of REST endpoints (methods for resources)

First, a quick definition: An *endpoint* is defined as a combination of a method and a resource, e.g. GET /customers, POST /customers., etc.

Typically, RESTful APIs have the following endpoints to implement the CRUD (create / read, update, delete) pattern on resources:

- POST /collection – Create – Adds a new item to the collection. For example, an API consumer would call POST on /cars to add a new car to a collection of cars.

- GET /collection – Read – Retrieve all (or some portion) of the items in a collection. For example, an API consumer would call GET on /cars to retrieve all cars in the collection.

  NOTE: To retrieve a partial list of cars, APIs allow *query parameters* to filter the results. For example, GET on /cars with a query parameter of "color=red" should return only cars whose color is red.

- GET /collection/{collectionID} – Read – Retrieve a single item in the collection, whose unique ID is its **collectionID** property. This should only retrieve ONE item from the collection. For example, use GET /cars/3 to get information about the car whose unique ID is **3**.

- PUT /collection/{collectionID} – Update the **entire** set of data about an object, but not any data related to that object. For example, if a **car** has an owner, PUT on /car/3 can update the owner for car 3, but, it should not update the information about the owner. For that, have a separate collection & member resource for owners / customers.
- PATCH /collection/{colelctionID} – Update (also) – Update *part* of a specific car's data. For example, maybe you only want to change the color, or the owner, but don't want to have to change anything else.

- DELETE /collection/{collectionID} – Delete – Removes a specific object from a collection. For example, DELETE /cars/3 would remove the car whose unique ID is 3.

  NOTE: for brevity, this blog post does not talk about the HEAD or OPTIONS HTTP / REST methods.

A Note About the ID

While this blog post is about designing a RESTful API with RAML, it is important to think about implementation for (at least) one thing: How does the implementation and the caller of the API specific a specific item, such as a specific car, customer, or sales-person?

Most RESTful APIs have a relational or similar database that holds records for each type of object represented by the API, and probably so meta-data too. One way to uniquely identify each record in a table of cars, customers, salespeople, etc. is to give each one a unique ID. This

is best done by the database management software, not the software submitting a new object through a POST request. For this reason, we need to have a separate object for submitting a new item with POST and retrieving an item via GET. It is the same object, but without the ID property, as that property gets set by the database (or other backend) system.

Neither PATCH nor PUT calls should provide an ID property.

A Note About PATCH

Because PATCH is designed to allow API consumers to only change certain parts of an object (and maybe not even all the fields we are allowed to change), it too needs a different data object. The object for PATCH should only list the properties of the object that API consumers are allowed to change. Further, each such property must be marked optional, since the API allows the properties to be changed, but does not require that all of said properties gt changed with each call to PATCH.

Resulting Data Types for a Resource

Given the past three sections, what we need for noun / resource is three different variations on the object:

- objectIN – this is a definition of the object with all of the properties, required and mandatory, except the ID column. For example, carIN may defined as follows:

```
types:
    carIN:
            properties:
                    color: string
                    engineSize: number
                    class:
                            enum [ COUP, CONVERTIBLE, SEDAN, SUV ]
                    owner: string # customerID
                    address: AddressIN
```

  NOTE: Address is itself its own data type. Unless the API uses the same properties for Address for all CRUD methods (GET / POST / PUT / PATCH / DELETE), it too should have separate data types for that object as well, e.g. AddressIN, AddressOut, AddressPatch.

- objectOUT – this is the definition of the object *with* the ID property. It should also have any sub-types used by the object with versions that end in OUT.

For example, building on carOUT above, declare carIN similar to the following:

```
types:
    carOUT:
            type: carIN # this brings the properties from carIN
            properties: # these are the additional / overrriden properties
                    carID: string
                    address: AddressOUT
```

- objectPATCH – This is the definition of the object just listing the properties the API consumers are allowed to change. For example, suppose only the color, owner, and address could be changed for a car. The type definition should look similar to the following:

```
types:
    carPATCH
            properties:
                    color?: string
                    owner?:  number
                    address?: AddressPatch
```

Using Libraries

You have three choices for creating the data types defined in the previous section:

1. Define all three datatypes in the *root* RAML file. If they aren't going to be reused at all, this is acceptable, but makes the root file large if there are many such datatypes. It has the advantage of putting all of the types next to each other so that when you add / remove / modify a property in one type, you have the other types right in front of you to change them as well as needed.
2. You can define all three types in separate dataType RAML fragments. This has the advantage of shortening the root file, but the disadvantage that when you add / remove / modify properties, you have to potentially edit all three files. This increases the opportunity for errors.
3. Define all three datatypes in a library RAML fragment. This is how TCCI recommends you do it. It addresses the shortcoming of the two previous options: it has all the fields in one place, it shortens the root file.

An example library for the car datatype would be:

```
#%RAML 1.0 Library

Types:
    carIN:
        properties:
          color: string
          engineSize: number
          class:
                  enum [ COUP, CONVERTIBLE, SEDAN, SUV ]
          owner: string # customerID
          address: AddressIN
    carOUT:
        type: carIN
        properties:
          carID: string
          address: AddressOUT
    carPATCH:
        properties:
          color?: string
          owner?:  number
          addres?s: AddressPatch
```

Consider using a name like *object*LIB.raml to sore the library, e.g. *carLib.raml*.

Use a Consistent Taxonomy

It is vital that you use a pattern when you define your library names, as well as the objects in your library. This consistent pattern is sometimes referred to as a *taxonomy*. When you follow the pattern, you have a *consistent taxonomy*. Having this consistency allows you to use RAML *resourceTypes* to describe your collections and member endpoints.

Finish it off with resourceTypes

RAML allows you to parameterize your resources. If (and only if) you have defined thee three object types for each resource, ideally in a Library, you can easily use resourceTypes to quickly and succinctly describe the methods you allow on each resource, and the dataTypes used for input and output.

A note about macros used in resourceTypes.

The RAML specification provides lists the macros that can be used in a resourceType (and a trait) here : Of those, the most TCCI recommends using are: methodName, resourcePathName, !singularize, !uppercamelcase.

- resourcePathName – the right-most part of a resource, excepting URI parameters. For example:
  - /customers -> customers
  - /customers/{customerID} -> customers
  - /customers/{customerID}/cars -> cars
- methodName – the name of the method for the endpoint
- !singularize – turns a plural word into its singular version. If it is already singular, leaves it alone. For example:
  - /people -> person
  - /customers -> customer
- !uppercamelcase – upper-cases the first word in a string, and every subsequent word. For example:
  - /person -> Person
  - /customerCars -> CustomerCars

With these parameters, create a resourceType for you collections similar to the following:

```
#%RAML 1.0 ResourceType

post?:
  displayName: Create a << resourcePathName | !singularize | !uppercamelcase>>
  description: Add a << resourcePathName | !singularize | !uppercamelcase>> record to the /<<
resourcePathName>> collection
  body:
    type: <<resourcePathName | !singularize>>.<< resourcePathName | !singularize |
!uppercamelcase>>In
  responses:
    201:
      description: Created a new << resourcePathName | !singularize | !uppercamelcase>>
      headers:
        Location:
          example: _<<resourcePathName>>/1234_
      body:
        type: nil
    503:
      description: |
        Failed to create a new Customer due to DB connection failure
      body:
        type: jer
        example:
          message: Failed to connect to DB
get?:
  is: [ traits.hasAcceptHeader, traits.cacheable ]
  displayName: Retrieve << resourcePathName | !uppercamelcase>>
  description: Retrieve 0+ << resourcePathName | !singularize | !uppercamelcase>> records from
the /<< resourcePathName>> collection
  responses:
    200:
      description: Retrieved all requested << resourcePathName | !uppercamelcase>>

      body:
        application/json:
          type: <<resourcePathName | !singularize>>.<< resourcePathName | !singularize |
!uppercamelcase>>Out[]

    503:
      description: |
        Failed to retrieve all requested << resourcePathName | !uppercamelcase>>
        due to DB connection failure
```

Then, create a resourceType fragment for your members similar to the following:

```
#%RAML 1.0 ResourceType

get?:
  is: [ traits.hasAcceptHeader, traits.cacheable, traits.has404And503Responses ]
  displayName: Retrieve a <<resourcePathName | !singularize | !uppercamelcase>>
  description: Retrieve one **<<resourcePathName | !singularize | !uppercamelcase>> by <<resourcePathName |
!singularize>>ID**
  responses:
    200:
      description: Retrieved requested <<resourcePathName | !uppercamelcase>>
      body:
        application/json:
          type: <<resourcePathName | !singularize>>.<<resourcePathName | !singularize | !uppercamelcase>>Out
put?:
  displayName: Update an entire <<resourcePathName | !singularize | !uppercamelcase>>
  description: Repleace one <<resourcePathName | !singularize | !uppercamelcase>> by <<resourcePathName |
!singularize>>ID
  is: [ traits.hasAcceptHeader, traits.has400Response, traits.has404And503Responses ]
  body:
    type: <<resourcePathName | !singularize>>.<<resourcePathName | !singularize | !uppercamelcase>>In
  responses:
    204:
      description: Updated requested <<resourcePathName | !singularize | !uppercamelcase>>
patch?:
  is: [ traits.has400Response, traits.has404And503Responses ]
  displayName: Update part of a <<resourcePathName | !singularize | !uppercamelcase>>
  description: Update parts of one <<resourcePathName | !singularize | !uppercamelcase>> by <<resourcePathName |
!singularize>>ID
  body:
    type: <<resourcePathName | !singularize>>.<<resourcePathName | !singularize | !uppercamelcase>>Patch
  responses:
    204:
      description: Updated requested <<resourcePathName | !singularize | !uppercamelcase>>
delete?:
  displayName: Delete a <<resourcePathName | !singularize | !uppercamelcase>>
  description: Delete one <<resourcePathName | !singularize | !uppercamelcase>> by <<resourcePathName |
!singularize>>ID
  is: traits.has404And503Responses
  responses:
    204:
      description: Deleted requested <<resourcePathName | !singularize | !uppercamelcase>>
options?:
  displayName: Retrieve methods
  responses:
    200:
      description: Returned operations on /<<resourcePathName>>/{<<resourcePathName | !singularize>>ID}
```

Now to create the endpoints for a given collection, such as cars, define them in your root file similar to the following:

```
#%RAML 1.0

Uses:
```

```
        cars: carsLib.raml

resourceTypes:
        collection: !include collection.raml
        member: !include member.raml

/cars:
        type: collection
        post:
        get:
        /{carID}:
                type: member
                get:
                put:
                patch:
                delete:
```

NOTE: since each method was defined with a question-mark (?) in the resourceType files, each method is optional for each type of resource. So, if you want that method, you have to list it, as we did here.

The code in this section results in the following endpoints, each with a displayName, data type, and description defined in the resourceType:

- POST /cars – datatype is from the carsLib's CarIN object.
- GET /cars – dataType is from the carLib's CarOUT object, as an array.
- GET /cars/{carID} – dataType is from carLib's CarOUT object
- PUT /cars/{carID} – dataType is from carLib's CarIN object just like POST
- PATCH /cars/{carID} – dataType is from carLib's carPatch
- DELETE /cars/{carID} – no dataType needed for delete's.

Give it a try.

The next time you are defining an API with RAML, start by:

1) Define the two resourceTypes (collection, member), which establishes your taxonomy, too.
2) For each resource (noun):
    a. Defining the 3 versions of the objects used in your CRUD endpoins with xxIN, xxOUT, and xxPATCH as described here.
    b. Define the collection and member resources and give each their respective resourceType (collection, member)

It becomes that simple. Any questions? Please send an email to info @ compclass.com and we'll try to give you a hand.