Chapter 1 - Consuming REST Web Services in Angular

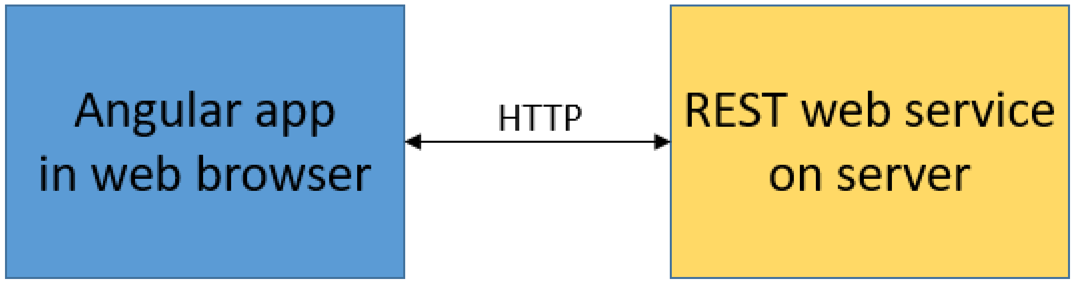
Objectives

Key objectives of this chapter

- REST Overview
- Common Angular tasks for REST communication
- Using Angular to send various HTTP requests

1.1 REST Web Services and Angular

- REST web services communicate using HTTP
- Although Angular doesn't have anything specific to "REST" web services, it does have the HTTP client which is the main tool to communicate with these web services from an Angular application running in the browser
- This chapter shows some things commonly needed when consuming REST web services and how to do this using the Angular HTTP client
- Although you need to know what kind of requests the REST web service will accept and what responses will be returned, what technology is used to implement it does not matter



1.2 Understanding REST

- REST applies the traditional, well-known architecture of the Web to Web Services
  - Everything is a resource or entity – for example, Orders, Customers, Price quote.
  - Each URI uniquely addresses an entity or set of entities

        - /orders/10025 – Order number 10025
        - /trains/BOM/DEL/02-23-2012 – All trains from Mumbai to New Delhi on 02-23-2012.
- Uses HTTP reply status code to indicate the outcome of an operation.
    - 200 or 204 (success), 404 (invalid URI), 500 (general error).
- Uses the **Content-Type** request and response header to indicate data format.

Understanding REST

REST heavily leverages the HTTP protocol. This creates a very familiar environment to provide and consume web services. For example, you can just enter a URL to issue a GET request. This simplicity and familiarity has driven the surge in popularity of this type of web services.

1.3 Understanding REST

- Leverages HTTP method for operating on resources (entities)
    - **GET** – Retrieves a resource or a collection of resources. A read-only operation
    - **DELETE** – Removes a resource
    - **PUT & POST** - "it depends"
- PUT & POST depend on where the resource identifier (primary key) is determined
    - Resource identifier is generated after the request reaches the server
        - eg. an order ID generated by a database insert
        - **POST:** Creates a resource
        - **PUT:** Updates a resource as a whole (e.g., entire Order resource)

- Resource identifier is part of the data sent by the client
  - eg. a flight number assigned outside the service
  - **PUT:** Creates a new resource
  - **POST:** Does a partial resource update (e.g., current flight status)

Understanding REST

PUT and POST are very similar methods. Both can create a new resource or update an existing resource. Updating a resource is easier to distinguish since a PUT is used when the entire content of the resource is being replaced and a POST is used when a partial update is performed. Which method to use when creating a new resource depends on how the resource identifier is determined. If the server-side REST service determines the resource identifier (perhaps auto-generated by a database) then a POST is used with a URI that does not include any resource identifier. If the client determines the resource identifier (perhaps by using a natural key like a social security number) then a POST is used and the URI has the resource identifier included (as if the resource already exists).

1.4 REST Example – Create

- If the entity identifier is created by the service a POST request is used to create
- Request

```
POST /RESTWeb/catalogs/products HTTP/1.1
Content-Type: text/xml
Content-Length: 142

<?xml version="1.0" encoding="UTF-8"?>
<product>
    <price>125.99</price>
    <name>Baseball bat</name>
</product>
```

- Response

```
HTTP/1.1 201 Created
Location: http://localhost/RESTWeb/catalogs/products/1029
Content-Length: 0
```

REST Example – Create

In this example the primary key, the product ID in this case, is generated by the service when inserting new data and is not part of the information sent in by the client with the initial request. Because of this the POST request is sent and the 'Location' header in the response is critical to know what address can be used to access the created entity later.

1.5 REST Example – Retrieve

- A GET request should always only retrieve data
  - A '404 Not Found' error should be returned if the data doesn't exist
- Request

```
GET /RESTWeb/catalogs/products/1029 HTTP/1.1
Accept: text/xml
```

- Response

```
HTTP/1.1 200 OK
Content-Type: text/xml
Content-Length: 137

<?xml version="1.0" encoding="UTF-8"?>
<product>
  <price>125.99</price>
  <name>Baseball bat</name>
  <code>1029</code>
</product>
```

1.6 REST Example – Update

- A PUT request can update existing data
- Request

```
PUT /RESTWeb/catalogs/products/1029 HTTP/1.1
Content-Type: text/xml
Content-Length: 144

<?xml version="1.0" encoding="UTF-8"?>
<product>
    <price>75.99</price>
    <name>Baseball bat</name>
    <code>1029</code>
</product>
```

- Response

```
HTTP/1.1 204 No Content
Content-Length: 0
```

REST Example – Update

Since for this service, new entities are created with a POST request, a PUT request will update an existing entity. Note that the identifier is part of the address the request is sent to.

1.7 REST Example – Delete

- The DELETE request is the simplest with no body to request or response
-

- Request

```
DELETE /RESTWeb/catalogs/products/1029 HTTP/1.1
```

- Response

```
HTTP/1.1 204 No Content
Content-Length: 0
```

1.8 REST Example – Client Generated ID

- If the identifier for data is provided by the client, a PUT request is used to create a new entity with the service
  - It can also update if the entity already exists
- Request

```
PUT /RESTWeb/flights/AA1215 HTTP/1.1
Content-Type: text/xml
Content-Length: 150

<?xml version="1.0" encoding="UTF-8"?>
<flight>
    <origin>MCO</origin>
    <dest>ORD</dest>
    <status>On Time</status>
    <code>AA1215</code>
</flight>
```

- Response

```
HTTP/1.1 201 Created
Location: http://localhost/RESTWeb/flights/AA1215
Content-Length: 0
```

REST Example – Client Generated ID

In the above example, if the request were updating existing data instead of creating a new entity the request would be the same. The response would likely be a '204 No Content' status code instead of '201 Created' to indicate the data was updated successfully. There would also be no need for a 'Location' header in the response.

1.9 REST Example – JSON

- It is very common for REST services to support communication with JSON, JavaScript Object Notation
  - This is much easier than XML for JavaScript clients that have been very common with REST services
- JSON uses curly brackets for the boundaries of objects along with comma-separated name/value pairs for properties
- Request

```
GET /RESTWeb/catalogs/products/1029 HTTP/1.1
Accept: application/json
```

- Response

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 62

{
  “price” : 125.99,
  “name” : “Baseball bat”,
  “code” : 1029
}
```

1.10 Knowledge of the REST API

- Knowledge of the REST web service being communicated with will be critical to writing any Angular service to simplify this communication
- Knowledge of the following will be needed:
  - What URLs a REST service will respond to
  - What HTTP methods are supported
  - What options or parameters need to be supplied with the requests
  - What data will be returned as a response

1.11 Common Angular Tasks for REST Communication

- Using the Angular HTTP service, there are several common tasks Angular applications need to do for REST communication
- An Angular service should be created to provide this functionality so visual Angular components are unaware of the HTTP communication details
- Most common requirement is to send requests using various HTTP methods depending on what the REST service accepts
  - GET, PUT, POST, and DELETE are most common
- You may need to supply various parameters as part of the request
  - Query and Path parameters are most common
- You may need to set various headers for the request
  - 'Content-Type' and 'Accept' are most common
- You will need to process the response to the request
  - This is done with the RxJs Observable API of the HTTP client service

1.12 Angular Service Class Using HTTP Client

- The following is the basic declaration of an Angular service class used in the rest of the examples of this chapter
    - It imports various elements of the Angular HTTP client and RxJs Observable API

```
// Imports
import { Injectable }    from '@angular/core';
import { Http,Response,Headers,RequestOptions } from
'@angular/http';
import { Observable } from 'rxjs/Rx';
import { Product }    from '../model/product';
// Import RxJs required methods
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/catch';

@Injectable()
export class CommentService {
     constructor (private http: Http) {}
     // private instance variable to hold base url
     private productsUrl =
'http://localhost:3000/catalogs/products';
     // other methods included here
}
```

Angular Service Class Using HTTP Client

One of the only things above that is not a "standard" is the 'Product' import. This is a custom TypeScript class that is part of the Angular application that defines the structure of data the service will be using to communicate with the REST service. This class could be something like that below:

```
export class Product {
    constructor(
        public id: number,
        public name: string,
        public price: number,
        public category: string,
        public description: string
        ){}
}
```

Notice that the only thing required in the constructor for the custom service is a reference to the Angular HTTP service.

1.13 RequestOptions

- Although the URL a request is sent to is the most important property of sending a request, many other options may need to be provided
  - This depends on the request being sent and what will be expected by a REST web service
- Angular provides a '**RequestOptions**' object so these various options can be provided as properties of a single object
- The various methods of the Angular HTTP service will accept a RequestOptions object as an optional parameter

```
let options = new RequestOptions({ // properties to set });
this.http.get(url, options)...
```

- The most commonly used properties of this object are:
  - headers – request headers to be set
  - search – URL search parameters (query parameters) to be added to the URL

RequestOptions

The RequestOptions object is not always passed in as the second parameter. In some methods it is the third parameter.

There are some properties of the RequestOptions object that are rarely if ever used:

- url – This is already a required parameter of the HTTP client methods
- method – The HTTP method of the request is almost always determined by the method called on the HTTP client service
- body – Any request type that requires a body is supplied as a parameter of the HTTP client methods

1.14 URL Path Parameters

- Path parameters are one of the more common ways to get data for REST services
- Path parameters are part of the main URL the request is sent to
  - This often adds some type of unique ID to some base URL for the service
- Requests for different data will have different values but will put this in the same location in the URL "path"

```
/RESTWeb/catalogs/products/1029
```

- These types of parameters would be used simply by combining some base URL with the value of a dynamic property in the URL the request is sent to

```
'${this.productsUrl}/${product['id']}'
```

1.15 Query Parameters

- Many searches that aren't by ID will pass in query parameters
  - These often supply optional parameters to narrow a search or specify things like sorting
- These are name/value pairs in the URL after a '?' and separated by '&'

```
/RESTWeb/catalogs/products?maxPrice=20&category=LEGO
```

- Query parameters must be "encoded" so that all of the characters used are legal for being used in a URL
  - The mechanism provided with Angular automatically encodes parameters
- To work with query parameters in Angular, you construct an object of the **URLSearchParams** type, set parameters, and then set it as the '**search**' property of a **RequestOptions** object passed as a parameter

```
let params = new URLSearchParams();
params.set('maxPrice', 20);
params.set('category', 'LEGO');
let options = new RequestOptions({ search: params });
this.http.get(url, options)...
```

Query Parameters

Obviously most of the time the values of the search parameters might be set using variables so the value can be more dynamic. The example above is used to show how the example URL provided previously could be constructed for the request.

1.16 Common HTTP Request Headers

- Although there are lots of standard headers that can be sent as part of HTTP requests, a few are most common:
  - An 'Accept' header indicates the data format a client would accept for the response
    - You can use the '.json()' method on the response object to parse the body into a JavaScript object

```
Accept: application/json
```

  - When the request contains a body with data, the 'Content-Type' header indicates the data format of the request body

```
Content-Type: application/json
```

- Angular applications mostly use 'application/json' as the value of these headers since JSON, or JavaScript Object Notation, is easy to consume in Angular applications
- To work with headers in Angular, you construct an object of the **Headers** type, append headers, and then set it as the '**headers**' property of a **RequestOptions** object passed as a parameter

```
let headers = new Headers();
headers.append('Content-Type', 'application/json');
headers.append('Accept', 'application/json');
let options = new RequestOptions({ headers: headers });
this.http.get(url, options)...
```

Common HTTP Request Headers

An alternate way to construct the Headers object is to pass an object to the constructor that has as property names the names of headers to set and the property values the values for the header.

```
let headers = new Headers({'Content-Type': 'application/json',
            'Accept': 'application/json'});
```

1.17 Override Default Request Options

- Rather than setting certain request options for every request, Angular provides a way to override or set default request options
- You override the default request options by providing a class that extends the **BaseRequestOptions** class and exports a custom provider

```
import { Injectable } from '@angular/core';
import { BaseRequestOptions, RequestOptions } from
'@angular/http';
```

```
@Injectable()
export class DefaultRequestOptions extends
BaseRequestOptions {
  constructor() {
    super();
    this.headers.set('Content-Type', 'application/json');
    this.headers.set('Accept', 'application/json');
  }
}
export const requestOptionsProvider = { provide:
RequestOptions, useClass: DefaultRequestOptions };
```

- You then register the provider in the root AppModule

```
providers: [ requestOptionsProvider ],
```

Override Default Request Options

To simplify code examples, the rest of this chapter will assume the above defaults for the 'Content-Type' and 'Accept' headers have been set.

The BaseRequestOptions class defines the defaults for a request to be to use the GET HTTP method and an empty set of headers. Since the HTTP method used is already often set by calling a specific method on the HTTP client API, the headers is the most common thing that would have the defaults changed in this way.

If you override the default request options in this way, make sure to include this provider when setting up the application for any testing configuration. If you don't, different request options would be used in production compared to testing which would likely cause a change in behavior.

1.18 Returning Response Data

- Angular uses a **Response** class with various details about the response
- Depending on the REST service, different return types could be returned from an Angular service that performs the REST communication
  - You could simply return the entire body of the response after parsing it

into a JavaScript object

```
resp => resp.json()
```

- You could also extract various response details into a custom "response wrapper" object

```
resp => {
      status: resp.status,
      statusText: resp.statusText,
      ok: resp.ok
    }
```

- The REST service may wrap the response itself, requiring extraction

```
resp => (resp.json()).result
```

Returning Response Data

In the last example above, the REST service returns a "wrapped" response where the data needed is provided as a 'result' property in the response. The response must be parsed from JSON and then the 'result' property extracted as the data of the response. This 'result' property is not a standard property of the Angular Response class.

Some of the important properties of the Response class are:

ok – boolean indicating if the response status code is between 200-299

status – status code returned by the server

statusText – Text corresponding to the status

headers – a **Headers** object with all of the response headers

1.19 DELETE

- DELETE requests are generally the easiest to implement with no request or response body to deal with
- The main information required is the unique URL for the data to be deleted

- You might often return an object with the status of the request

```
deleteProduct(product: Product) : Observable<any> {
   return this.http.delete(
        '${this.productsUrl}/${product.id}')
        .map(this.mapResponseStatus).catch(...);

mapResponseStatus(resp: any): any {
  return {
    status: resp.status,
    statusText: resp.statusText,
    ok: resp.ok
  };
}
```

1.20 GET

- An HTTP GET request is one of the most common requests to a REST service
- Important elements – URL with optional search (query) parameters for data to retrieve and the response body with the requested data
- Sending a GET request to obtain all products

```
getProducts() : Observable<Product[]> {
     return this.http.get(this.productsUrl)
          .map((res:Response) => res.json())
          .catch((error:any) =>
Observable.throw(error.json().error));
}
```

- Sending search parameters for products

```
let params = new URLSearchParams();
params.set('maxPrice', 20);
```

```
     params.set('category', 'LEGO');
     let options = new RequestOptions({ search: params });
     return this.http.get(this.productsUrl, options)...
```

- Requesting a specific product adding an ID to the path

```
getProductByID(id: number) : Observable<Product> {
     return this.http.get(
          '${this.productsUrl}/${id}')...
```

GET

The above example assumes that the 'Accept' header was already set by the request defaults code shown earlier in the chapter.

It is important to know exactly what format will be used to return the response body. It may be simply a case of calling the '.json()' method on the response object or needing to access a particular property in the object that is returned (like 'data'), to extract the response data desired. If it is more than just calling '.json()', it might be good to define a method that extracts the data and then refer to this method in the mapping of the response. You could also define a separate method for more complex error processing. For example:

```
private extractData(res: Response) {
    let body = res.json();
    return body.data || { };  // returns an empty object if body.data is undefined
}

private handleError (error: Response | any) {
    let errMsg: string;
    if (error instanceof Response) {
      const body = error.json() || '';
      const err = body.error || JSON.stringify(body);
      errMsg = `${error.status} - ${error.statusText || ''} ${err}`;
    } else {
      errMsg = error.message ? error.message : error.toString();
    }
    console.error(errMsg);
    return Observable.throw(errMsg);
}
```

```
getProducts() : Observable<Product[]> {
     return this.http.get(this.productsUrl)
          .map(this.extractData)
          .catch(this.handleError);
}
```

1.21 PUT

- PUT requests are used to update data or create data when the unique resource identifier is determined by the client
- Important elements
  - A unique URL for the data being created/updated
  - Request body with the data itself
  - 'Content-Type' header with the content type of the request body
- The HTTP client '.put' method requires a second parameter which would be the body of the request
  - An optional third parameter for the request options can be supplied if the default request options are not already sufficient

```
addEditProduct(product: Product) : Observable<any> {
     return this.http.put(
          '${this.productsUrl}/${product.id}',
          JSON.stringify(product)).map(...).catch(...);
```

PUT

The above example assumes that the 'Content-Type' header was already set by the request defaults code shown earlier in the chapter.

A version of the above example that explicitly sets the 'Content-Type' header would be:

```
let headers = new Headers({ 'Content-Type': 'application/json' });
let options = new RequestOptions({ headers: headers });
return this.http.put(
          '${this.productsUrl}/${product.id}',
          JSON.stringify(product),
          options).map(...).catch(...);
```

Since the response does not contain any data retrieved from the server, you might extract the response status information into an object returned from the service.

```
  mapResponseStatus(resp: any): any {
    return {
      status: resp.status,
      statusText: resp.statusText,
      ok: resp.ok
    };
  }

  addEditProduct(product: Product) : Observable<any> {
     return this.http.put(
          '${this.productsUrl}/${product.id}',
          JSON.stringify(product)).map(this.mapResponseStatus).catch(...);
```

1.22 POST

- POST requests are also used to create/update data similar to PUT requests with a few important differences
  - The URL doesn't have a URL unique to the data since the entity identifier is determined by the REST service processing the request
  - The response has a 'Location' header that has the URL used to access the created/modified data
- You could either return just the location header value or a "wrapper" object that contains the location and other response details

```
addProduct(product: Product) : Observable<any> {
     return this.http.post(
          this.productsUrl,
          JSON.stringify(product))
          .map(resp => resp.headers.get("location"))
          .catch(...);
```

POST

The above example assumes that the 'Content-Type' header was already set by the request defaults code shown earlier in the chapter.

Returning a "wrapper" object that includes the location header value could be done as below:

```
mapResponse(resp: any): any {
  return {
    location: resp.headers.get("location"),
    status: resp.status,
    statusText: resp.statusText,
    ok: resp.ok
  };
}

addProduct(product: Product) : Observable<any> {
   return this.http.post(
        this.productsUrl,
        JSON.stringify(product)).map(this.mapResponse).catch(...);
```

1.23 Security of REST APIs

- Any REST API that is exposed to web applications will likely have some type of security mechanism that must be considered
- This can vary greatly between organizations but the most common is some type of "security access token" that must be sent with a request
  - The token could be unique for each end user or generated for the application as a whole
- One of the most common for REST APIs is the use of JSON Web Tokens (or JWT)
  - These are JSON objects that contain arbitrary "claims" as the main payload that can have information like username, token expiration, etc
  - These are encoded and signed to provide security for the token itself
- JWT tokens are typically sent as part of a request as an HTTP

'Authorization' header

- OAuth (or OAuth 2) is another common topic with modern web security
  - ◇ It provides a broader infrastructure for securely issuing and distributing security tokens and can be used with arbitrary token types including JWT

1.24 Summary

- Communicating with REST web services leverages standard patterns of HTTP requests and responses
- Knowledge of exact details of the requests expected and the responses provided from a specific REST service is critical
- You can use the Angular HTTP client API in fairly standard ways to send and receive data from REST services
- It is suggested to create an Angular service as part of the application to hide the details of the HTTP communication and simplify the process