Chapter 1 - HTTP Client

Objectives

Key objectives of this chapter

- What is the Angular HTTP Client
- Importing HttpModule
- Making Get/ Post Calls
- Working with Observables
- Request and Response Headers

1.1 The Angular HTTP Client

The Angular HTTP Client:

- Provides a simplified API for network communication
- Supports:
  - Making HTTP requests (GET, POST, etc.)
  - Working with request and response headers
  - Asynchronous programming
- Makes use of the rxjs async library Observable object

Notes

The Angular Http client offers a simplified API compared to the JavaScript XMLHttpRequest object.

1.2 Using The HTTP Client - Overview

Typical usage involves a service that is used by one or more components:

- Imports are setup in the application's Root Component:
  - Import Angular HttpModule
  - Import rxjs asynchronous libraries
- A Data Service is created that makes network requests:

- Required Http libraries and the Observable object are imported
- An Observable object is returned when network requests are executed
- Data is mapped from Response

- An Angular Component's view is used to display the data :
  - The data service is imported and injected into the constructor
  - A reference to the service's Observable object is obtained,
  - The component then "subscribes" to receive updates to the Observable object.
  - The component's view displays the data

Notes:

The initial use-case for nework requests involves the retrieval of data from a a server and the display of that data in a view.

In this case a GET request is used.

Mapping of data refers to extracting the data that will be passed to subscribers from the Response object.

1.3 Importing HttpModule

- In order to use the Http client throughout the application we need to import the Angular HttpModule in the module of the application
  - HttpModule is a collection of service providers from the Angular HTTP library
- Two steps are involved in 'app.module.ts' to import the HttpModule
  - Import HttpModule at the top of the file

```
import { HttpModule }    from '@angular/http';
```

- ◊ Add HttpModule to the imports in the @NgModule decorator

```
@NgModule({
  imports:       [ BrowserModule, HttpModule ],
```

- ■ When added in the application module, services from HttpModule become injectable in all components

1.4 Importing Individual Providers into Services

- ■ "`Http`" is one of several providers included in HttpModule:

`Http`	To make network calls
`Response`	Used to retrieve response data
`Headers`	Used to set request headers
`RequestOptions`	Used to set request parameters for Http calls

- ■ Only the APIs that are used need to be imported.
- ■ APIs are imported into services that use them like this:

```
import { Http } from '@angular/http';
```

- ■ This import is required *in addition* to the import of HttpModule in the application module

Notes:

If needed all required APIs can be imported using the same import statement:

```
import { Http, Response, Headers, RequestOptions } from '@angular/http';
```

1.5 Service Using Http Client

- Sample Service Code ("people.service.ts"):

```
import { Injectable } from '@angular/core';
import { Http } from '@angular/http';
import { Observable } from 'rxjs/Observable';

@Injectable()
export class PeopleService{
  people:Observable<Response>;
  constructor(http: Http){
   this.people = http
       .get('app/data.json')
       .catch(this.onError);
  }
  onError(res: any){
    let msg = res.status + ":" + res.statusText
    console.log(msg);
    return Observable.throw(msg);
  }
}
```

- This code is reviewed in the next few slides

Notes:

We review various lines of the code above in the next few slides including:

- Import statements
- The people object and its type
- Injection of the Http object
- Making a GET call and returning a response.

1.6 Service Imports

- There are three imports in total that we must add to the data service:

```
import { Injectable } from '@angular/core';
import { Http } from '@angular/http';
import { Observable } from 'rxjs/Observable';
```

- The first allows us to mark the class so that it can be injected into the components that need it
- The second allows us to use the Http object in the service to make network calls.
- The third allows the service to hold a reference to the Observable object that is returned when the network call is made

1.7 The Observable object type

- Network request calls made using the Http object return an Observable type object:

```
people:Observable<Response> = http.get(url);
```

- In the statement above:
  - Get is called on the `http` object
  - It immediately returns an object which is assigned to the variable `people`.
  - The `people` variable is of type `Observable<Response>`
  - `<Response>` refers to the type of data that the Observable returns, in this case an Angular Response object.

Notes:

As they are used with the Angular Http object Observable objects can be thought of as the *glue* that connects network requests with data consumers. There are several aspects to this relationship that our discusssion will cover one at a time.

From the current slide we see that http requests return Observable objects that we can assign to a variable for later use.

In the next slide we will take a look at how Observable objects are used.

1.8 What does an Observable Object do?

- The use of the Observable object by the Angular Http Client is fairly basic compared to its full capabilities.
- Angular uses the Reactive Extensions for JavaScript (RxJS) implementation of the Observable object.
- Observable objects provide a convenient way for applications to consume asynchronous data/event streams.
- Observable objects are exposed to events which they then make available to consumers.
- Applications consume events by subscribing to the Observable object.
- The Observable object can be used to transform data before returning it to a consumer if needed.

Notes:

Reactive programming and Observable objects can be used anywhere an asynchronous programming model is required. For more information on these topics see:

```
http://reactivex.io/intro.html
```

1.9 Making a Basic HTTP GET Call

- An HTTP GET call is made in the constructor as shown here:

```
constructor(http: Http){
      this.people = http.get('app/data.json')
      .catch(this.onError);
}
```

- First the Http object is injected into the constructor
- Then http.get() is called
- "app/data.json" points to a data file on the server
- .catch registers an onError function to be called if an error occurs.

Notes

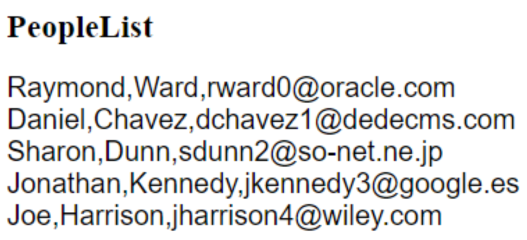
A copy of the full service for reference (same as show earlier):

```
// people.service.ts
import { Injectable } from '@angular/core';
import { Http } from '@angular/http';
import { Observable } from 'rxjs/Observable';

@Injectable()
export class PeopleService{
  people:Observable<Response>;
  constructor(http: Http){
   this.people = http
         .get('app/data.json')
         .catch(this.onError);
  }
  onError(res: any){
    let msg = res.status + ":" + res.statusText
    console.log(msg);
    return Observable.throw(msg);
  }
}
```

1.10 Using the Service in a Component

- Once the service has been created we can create an Angular component that uses it to retrieve and display data.
- Our component will have to import the service.
- In order to retrieve data the code in our component will have to work with the Observable object that was created in the service.
- The screen shot at right shows how the data will look when displayed.


PeopleList

Raymond,Ward,rward0@oracle.com
Daniel,Chavez,dchavez1@dedecms.com
Sharon,Dunn,sdunn2@so-net.ne.jp
Jonathan,Kennedy,jkennedy3@google.es
Joe,Harrison,jharrison4@wiley.com

1.11 The PeopleService Client Component

Code for the Component (explanation to follow):

```
import { Component } from '@angular/core';
import {PeopleService} from './people.service';

@Component({
  selector: 'people',
  template: `<h3>PeopleList</h3><ul>
            <li *ngFor="let x of list">
            {{x.fname}},{{x.lname}},{{x.email}}
            </li></ul>`,
  providers:[PeopleService]
})

export class PeopleList{
  list: Object[];
  constructor( srvPeople : PeopleService){
    srvPeople.people.subscribe(
       data => this.list = data.json(),
       error => console.error('Error: ' + error));
  }
}
```

1.12 Client Component Code Review

- `PeopleService` is imported
- The `ngFor` in the HTML template references the `list` variable
- The `list` variable is created as an Object array
- `PeopleService` is injected into the constructor
- `subscribe()` is called on the service's Observable `people` object
- The first parameter to subscribe calls `.json()` on the response object and assigns it to the `list` variable
- The second subscribe parameter is invoked if an error occurs. It is setup here to log the error to the JavaScript console.

1.13 Importing Observable Methods

- Observable has many methods supporting various capabilities.
- Some of these methods exist by default:

```
.subscribe()
```

- To save space other methods are only added if needed:

```
.catch()
```

```
.map()
.throw()
.toPromise()
```

- These methods can be added individually using import statements:

```
import 'rxjs/add/operator/catch';
```

- We will take a look at how these are used in the next few slides.

Notes:

We've used the subscribe() method in the component to pass callback functions for the Observable object to use when data is returned or when an error occurs.

We will take a look in the next slide at how catch() and map() can be used in the service to handle errors and convert data.

Details of the subscribe methods can be found at:

http://reactivex.io/documentation/operators/subscribe.html

Details of other methods are found here:

http://reactivex.io/documentation/operators.html

1.14 Enhancing the Service with .map() and .catch()

- We can transform data and catch errors in our service using .map() and .catch().
- The `people.service.ts` service.

```
@Injectable()
export class PeopleService{
  people:Observable<Object[]>;
  constructor(http: Http){
   this.people = http.get('app/data.json')
     .map(response => response.json())
```

```
                .catch(msg => Observable.throw(msg));
        }
    }
```

- This code will be reviewed in the next few slides

1.15 Using .map()

.map() can be used to transform values returned from the Http client.

- In the code .map() is chained after .get()
- The call back arrow function passed to .map() returns an object array

```
.map(response => response.json())
```

- The type of the `people` property must be changed to align with the return value of the `.map()` function

```
people:Observable<Object[]>;
```

- Since the type returned from the Observable has changed we need to make a corresponding change in the component that uses it. The type of the return value now matches that of the list variable (`Object[]`) so it can be assigned directly (we no longer need to call `.json()` on it):

```
people => this.list = people
```

Notes:

.map() is a method on the Observable object. When used it needs to be imported like this:

```
import 'rxjs/add/operator/map';
```

Additional documentation on Observable.map() can be found here:

http://reactivex.io/documentation/operators/map.html

It is because of the fact that .get() returns an Observable object that .map() can be chained after it. In practice .map() can be chained after any method that returns Observable.

Previously the type of the people variabe was an Observable that returned a Response object.

```
people:Observable<Response>;
```

The Response object includes information about status and headers in addition to the returned data. See here for more information on the Response object:

https://angular.io/docs/ts/latest/api/http/index/Response-class.html#!%23json-anchor

We needed to call .json() on the Response object to get an Object[] we could assign to the list variable in the component:

```
.subscribe(data => this.list = data.json,...)
```

Using .map() lets us do the transformation in the service itself and simplify the component.

1.16 Using .catch()

.catch() is used to handle errors that occur during an Http call.

- In the code .catch() is chained after .map()
- The method passed to catch must return an Observable object

```
    .catch(msg => Observable.throw(msg));
```

- Here we use a call to the static throw method of Observable to create a new Observable encapsulating the error information.

```
    Observable.throw(msg)
```

- Instead of an arrow function we could pass a method from the current class that handles the error

  Observable.throw(this.onError)

- In onError we could do something to handle the error condition and to notifiy the user.

Notes:

.catch() and .throw() are methods on the Observable object.

When used they to be imported like this:

```
import 'rxjs/add/observable/throw';
import 'rxjs/add/operator/catch';
```

Here is an example onError method:

```
private handleError (error: any) {
  let errMsg = (error.message) ? error.message :
      error.status ? `${error.status} - ${error.statusText}` : 'Server error';
  console.error(errMsg); // log to console instead
  return Observable.throw(errMsg);
}
```

1.17 Using toPromise()

toPromise() allows you to convert an Observable object into a promise

- This allows you to use familiar promise methods like `.then()`
- A typical use-case would be to convert the Observable to a promise in the service and then let the component retrieve the promise instead.
- The service would include the following code:

  ```
  this.peoplePromise = http.get('app/data.json')
  .map(response => response.json()).toPromise();
  ```
- The component would use the following code:

  ```
  srvPeople.peoplePromise.then(
    data => this.list = data,
    error => console.error('Error: ' + error)
  );
  ```

Notes:

.toPromise() is a method on the Observable object.

When used it needs to be imported like this:

```
import 'rxjs/add/operator/toPromise';
```

The full `people.promise.service.ts` using toPromise():

```
import {Injectable} from '@angular/core';
import {Http, Response} from '@angular/http';
import {Observable} from 'rxjs/Observable';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/toPromise';

@Injectable()
export class PeoplePromiseService{

  peoplePromise: Promise<Object[]>;

  constructor(http: Http){
          this.peoplePromise = http.get('app/data.json')
          .map(response => response.json()).toPromise();
  }
}
```

The full `people.promise.component.ts` using the Promise object:

```
import { Component } from '@angular/core';
import {PeoplePromiseService} from './people.promise.service';

@Component({
  selector: 'people',
  template: `<h3>PeopleList</h3><ul>
            <li *ngFor="let x of list">
            {{x.fname}},{{x.lname}},{{x.email}}
            </li></ul>`,
  providers:[PeoplePromiseService]
})

export class PeopleList{

  list: Object[];

  constructor(srvPeople: PeoplePromiseService){
    srvPeople.peoplePromise.then(
           data => this.list = data,
           error => console.error('Error: ' + error)
    );
  }
}
```

1.18 GET Request

- GET Requests are typically used to retrieve data.
- Full syntax for an Http client GET request:

```
get(url: string, options?: RequestOptionsArgs) :
Observable<Response>
```

  - The first parameter is a URL string.
  - The second parameter is optional. When supplied it contains a RequestOptionsArgs object
  - By default the .get() call returns an Observable object that itself returns a Response type object.

Notes:

Up until now we have focused on a simple get request taking a single URL parameter.

Additional options are available when making Http get requests with multiple parameters.

Although the default return type of the created Observable is Response we can change that as seen earlier by using the .map() command.

Complete information about the RequestOptions object can be found here:

https://angular.io/docs/ts/latest/api/http/index/RequestOptions-class.html

1.19 GET Request with Options

- A typical .get() call using both parameters looks like this:

```
constructor(http: Http){
  let headers = new Headers();
  headers.append("Content-Type",
"application/json");
  let requestOptions =
  new RequestOptions({headers: headers });
  this.peopleOptions =
  http.get('app/data.json', requestOptions)
  .map(response => response.json());
}
```

- The objects, Headers and RequestOptions need to be imported for this code to compile

Notes

Full service example (people.get.options.service.ts):

```
import {Injectable} from '@angular/core';
import {Http, Response, Headers, RequestOptions} from '@angular/http';
import { Observable } from 'rxjs/Observable';

@Injectable()
export class PeopleService{

  peopleOptions: Observable<Object[]>;

  constructor(http: Http){
          let headers = new Headers();
          headers.append("Content-Type", "application/json");
          let requestOptions = new RequestOptions({headers: headers });
          this.peopleOptions = http.get('app/data.json', requestOptions)
          .map(response => response.json());
  }
}
```

1.20 POST Request

- POST requests are typically used to send data to a server.
- Full syntax for an Http client POST request:

```
post(url: string, body: any,
options?: RequestOptionsArgs)
: Observable<Response>
```

- ◊ The first parameter is a URL string.
- ◊ The second parameter is used to supply the HTTP request's body content.
- ◊ The third parameter is optional: When supplied it contains a RequestOptionsArgs object
- ◊ By default the .post() call returns an Observable object that itself returns a Response type object.

1.21 POST Request Example

- Example POST Request:

```
doPost(){
  var data = {id:25,name:"Steve"};
  var body = JSON.stringify(data);
  let headers = new Headers();
  headers.append("Content-Type",
               "application/json");
  let requestOptions = new RequestOptions({
     headers: headers
  });
  this.peopleOptions = this.http.post(
    'app/data.json', body, requestOptions)
  .map(response => response.json());
}
```

- The example sets body content and headers for the request.

Notes:

Full service example (people.post.service.ts):

```
import {Injectable} from '@angular/core';
import {Http, Response, Headers, RequestOptions} from '@angular/http';
import { Observable } from 'rxjs/Observable';

@Injectable()
export class PeopleService{

  peopleOptions: Observable<Object[]>;

  constructor(private http: Http){this.doPost();}

  doPost(){
    var data = {id:25,name:"Steve"};
          var body = JSON.stringify(data);
          let headers = new Headers();
          headers.append("Content-Type", "application/json");
          let requestOptions = new RequestOptions({headers: headers});
          this.peopleOptions = this.http.post('app/data.json', body,
requestOptions)
          .map(response => response.json());
  }
}
```

1.22 Reading HTTP Response Headers

- By default `.get()` and `.post()` return Response type objects that include header information.
- This code retrieves and displays a single named header:

```
logDateHeader(response: Response){
  console.log(response.headers.get("Date"));
}
```

- The following code reads response headers and logs them to the console:

```
logHeaders(response: Response){
  response.headers.forEach(function(value, key, map)
{
    console.log(key + ": " + value);
  });
}
```

- Other Response object data can be retrieved in a similar fashion.

Notes

Documentation for the Reponse object can be found here:

https://angular.io/docs/ts/latest/api/http/index/Response-class.html

Full Service Example ("people.headers.service.ts"):

```
import {Injectable} from '@angular/core';
import {Http, Response} from '@angular/http';
import { Observable } from 'rxjs/Observable';

@Injectable()
export class PeopleHeadersService{
  people:Observable<Object[]>;

  constructor(private http: Http){
         this.people = http.get('app/data.json')
         .map(response => { this.logHeaders(response)); response.json(); }
  }

  logHeaders(response: Response){
    response.headers.forEach(function(value, key, map){
    console.log(key + ": " + value);
   });
  }

  logDateHeader(response: Response){
    console.log(response.headers.get("Date"));
  }

}
```

1.23 Summary

In this chapter we covered:

- What is the Angular HTTP Client
- Importing HttpModule
- Making Get/ Post Calls
- Working with Observables
- Request and Response Headers