Angular in-memory-web-api

An in-memory web api for Angular demos and tests that emulates CRUD operations over a RESTy API.

It intercepts Angular `Http` and `HttpClient` requests that would otherwise go to the remote server and redirects them to an in-memory data store that you control.

See Austin McDaniel's article for a quick introduction.

This package used to live in its own repository.

It used to work and now it doesn't :-(

Perhaps you installed a new version of this library? Check the CHANGELOG.md for breaking changes that may have affected your app.

If that doesn't explain it, create an issue on github, preferably with a small repro.

Use cases

- Demo apps that need to simulate CRUD data persistence operations without a real server. You won't have to build and start a test server.
- Whip up prototypes and proofs of concept.
- Share examples with the community in a web coding environment such as Plunker or CodePen. Create Angular issues and StackOverflow answers supported by live code.
- Simulate operations against data collections that aren't yet implemented on your dev/test server. You can pass requests thru to the dev/test server for collections that are supported.
- Write unit test apps that read and write data. Avoid the hassle of intercepting multiple http calls and manufacturing sequences of responses. The in-memory data store resets for each test so there is no cross-test data pollution.
- End-to-end tests. If you can toggle the app into test mode using the in-memory web api, you won't disturb the real database. This can be especially useful for CI (continuous integration) builds.

  LIMITATIONS

  The *in-memory-web-api* exists primarily to support the Angular documentation. It is not supposed to emulate every possible real world web API and is not intended for production use.

> Most importantly, it is ***always experimental***. We will make breaking changes and we won't feel bad about it because this is a development tool, not a production product. We do try to tell you about such changes in the `CHANGELOG.md` and we fix bugs as fast as we can.

HTTP request handling

This in-memory web api service processes an HTTP request and returns an `Observable` of HTTP `Response` object in the manner of a RESTy web api. It natively handles URI patterns in the form `:base/:collectionName/:id?`

Examples:

```
// for requests to an `api` base URL that gets heroes from a 'heroes' collection
GET api/heroes          // all heroes
GET api/heroes/42       // the hero with id=42
GET api/heroes?name=^j  // 'j' is a regex; returns heroes whose name starting with 'j' or
GET api/heroes.json/42  // ignores the ".json"
```

The in-memory web api service processes these requests against a "database" - a set of named collections - that you define during setup.

Basic setup

Create an `InMemoryDataService` class that implements `InMemoryDbService`.

At minimum it must implement `createDb` which creates a "database" hash whose keys are collection names and whose values are arrays of collection objects to return or update. For example:

```
import { InMemoryDbService } from 'angular-in-memory-web-api';

export class InMemHeroService implements InMemoryDbService {
  createDb() {
    let heroes = [
      { id: 1, name: 'Windstorm' },
      { id: 2, name: 'Bombasto' },
      { id: 3, name: 'Magneta' },
      { id: 4, name: 'Tornado' }
    ];
    return {heroes};
  }
}
```

Notes

- The in-memory web api library *currently* assumes that every collection has a primary key called `id`.

- The `createDb` method can be synchronous or asynchronous. It would have to be asynchronous if you initialized your in-memory database service from a JSON file. Return the database *object*, an *observable* of that object, or a *promise* of that object. The tests include an example of all three.
- The in-memory web api calls your `InMemoryDbService` data service class's `createDb` method on two occasions.

  1. when it handles the *first* HTTP request
  2. when it receives a `resetdb` command.

  In the command case, the service passes in a `RequestInfo` object, enabling the `createDb` logic to adjust its behavior per the client request. See the tests for examples.

Import the in-memory web api module

Register your data store service implementation with the `HttpClientInMemoryWebApiModule` in your root `AppModule.imports` calling the `forRoot` static method with this service class and an optional configuration object:

```
import { HttpClientModule } from '@angular/common/http';
import { HttpClientInMemoryWebApiModule } from 'angular-in-memory-web-api';

import { InMemHeroService } from '../app/hero.service';

@NgModule({
 imports: [
   HttpClientModule,
   HttpClientInMemoryWebApiModule.forRoot(InMemHeroService),
   ...
 ],
 ...
})
export class AppModule { ... }
```

Notes

- Always import the `HttpClientInMemoryWebApiModule` *after* the `HttpClientModule` to ensure that the in-memory backend provider supersedes the Angular version.
- You can setup the in-memory web api within a lazy loaded feature module by calling the `.forFeature` method as you would `.forRoot`.
- In production, you want HTTP requests to go to the real server and probably have no need for the *in-memory* provider. CLI-based apps can exclude the provider in production builds like this:

  ```
  imports: [
  ```

```
    HttpClientModule,
    environment.production ?
      [] : HttpClientInMemoryWebApiModule.forRoot(InMemHeroService)
    ...
]
```

Examples

The tests are a good place to learn how to setup and use this in-memory web api library.

See also the example source code in the official Angular.io documentation such as the HttpClient guide and the Tour of Heroes.

Advanced Features

Some features are not readily apparent in the basic usage described above.

Configuration arguments

The `InMemoryBackendConfigArgs` defines a set of options. Add them as the second `forRoot` argument:

```
InMemoryWebApiModule.forRoot(InMemHeroService, { delay: 500 }),
```

Read the `InMemoryBackendConfigArgs` interface to learn about these options.

Request evaluation order

This service can evaluate requests in multiple ways depending upon the configuration. Here's how it reasons: 1. If it looks like a command, process as a command. 2. If the HTTP method is overridden, try the override. 3. If the resource name (after the api base path) matches one of the configured collections, process that. 4. If not but the `Config.passThruUnknownUrl` flag is `true`, try to pass the request along to a real *XHR*. 5. Return a 404.

See the `handleRequest` method implementation for details.

Default delayed response

By default this service adds a 500ms delay to all data requests to simulate round-trip latency.

> Command requests have zero added delay as they concern in-memory service configuration and do not emulate real data requests.

You can change or eliminate the latency by setting a different `delay` value:

```
InMemoryWebApiModule.forRoot(InMemHeroService, { delay: 0 }),    // no delay
InMemoryWebApiModule.forRoot(InMemHeroService, { delay: 1500 }), // 1.5 second delay
```

Simple query strings

Pass custom filters as a regex pattern via query string. The query string defines which property and value to match.

Format: `/app/heroes/?propertyName=regexPattern`

The following example matches all names that start with the letter 'j' or 'J' in the heroes collection.

`/app/heroes/?name=^j`

> Search pattern matches are case insensitive by default. Set `config.caseSensitiveSearch = true` if needed.

Pass thru to a live server

If an existing, running remote server should handle requests for collections that are not in the in-memory database, set `Config.passThruUnknownUrl: true`. Then this service will forward unrecognized requests to the remote server via the Angular default `XHR` backend (it depends on whether your using `Http` or `HttpClient`).

Commands

The client may issue a command request to get configuration state from the in-memory web api service, reconfigure it, or reset the in-memory database.

When the last segment of the *api base path* is "commands", the `collectionName` is treated as the *command*.

Example URLs:

```
commands/resetdb   // Reset the "database" to its original state
commands/config    // Get or update this service's config object
```

Usage:

```
http.post('commands/resetdb', undefined);
http.get('commands/config');
http.post('commands/config', '{"delay":1000}');
```

Command requests do not simulate real remote data access. They ignore the latency delay and respond as quickly as possible.

The `resetDb` command calls your `InMemoryDbService` data service's `createDb` method with the `RequestInfo` object, enabling the `createDb` logic to adjust its behavior per the client request.

In the following example, the client includes a reset option in the command request body:

```
http
  // Reset the database collections with the `clear` option
  .post('commands/resetDb', { clear: true }))

  // when command finishes, get heroes
  .concatMap(
    ()=> http.get<Data>('api/heroes')
        .map(data => data.data as Hero[])
  )

  // execute the request sequence and
  // do something with the heroes
  .subscribe(...)
```

See the tests for other examples.

parseRequestUrl

The `parseRequestUrl` parses the request URL into a `ParsedRequestUrl` object. `ParsedRequestUrl` is a public interface whose properties guide the in-memory web api as it processes the request.

Default *parseRequestUrl*

Default parsing depends upon certain values of `config`: `apiBase`, `host`, and `urlRoot`. Read the source code for the complete story.

Configuring the `apiBase` yields the most interesting changes to `parseRequestUrl` behavior:

- For `apiBase=undefined` and `url='http://localhost/api/customers/42'` `ts {apiBase: 'api/', collectionName: 'customers', id: '42', ...}`
- For `apiBase='some/api/root/'` and `url='http://localhost/some/api/root/customers'` `ts { apiBase: 'some/api/root/', collectionName: 'customers', id: undefined, ... }`
- For `apiBase='/'` and `url='http://localhost/customers' ts { apiBase: '/', collectionName: 'customers', id: undefined, ... }`

The actual api base segment values are ignored. Only the number of segments matters. The following api base strings are considered identical: 'a/b' ~ 'some/api/' ~ 'two/segments'

This means that URLs that work with the in-memory web api may be rejected by the real server.

Custom *parseRequestUrl*

You can override the default parser by implementing a `parseRequestUrl` method in your `InMemoryDbService`.

The service calls your method with two arguments. 1. `url` - the request URL string 1. `requestInfoUtils` - utility methods in a `RequestInfoUtilities` object, including the default parser. Note that some values have not yet been set as they depend on the outcome of parsing.

Your method must either return a `ParsedRequestUrl` object or `null`|`undefined`, in which case the service uses the default parser. In this way you can intercept and parse some URLs and leave the others to the default parser.

Custom *genId*

Collection items are presumed to have a primary key property called `id`.

You can specify the `id` while adding a new item. The service will blindly use that `id`; it does not check for uniqueness.

If you do not specify the `id`, the service generates one via the `genId` method.

You can override the default id generator with a method called `genId` in your `InMemoryDbService`. Your method receives the new item's collection and collection name. It should return the generated id. If your generator returns `null`|`undefined`, the service uses the default generator.

responseInterceptor

You can change the response returned by the service's default HTTP methods. A typical reason to intercept is to add a header that your application is expecting.

To intercept responses, add a `responseInterceptor` method to your `InMemoryDbService` class. The service calls your interceptor like this:

```
responseOptions = this.responseInterceptor(responseOptions, requestInfo);
```

HTTP method interceptors

You may have HTTP requests that the in-memory web api can't handle properly.

You can override any HTTP method by implementing a method of that name in your `InMemoryDbService`.

Your method's name must be the same as the HTTP method name but **all lowercase**. The in-memory web api calls it with a `RequestInfo` object that contains request data and utility methods.

For example, if you implemented a `get` method, the web api would be called like this: `yourInMemDbService["get"](requestInfo)`.

Your custom HTTP method must return either:

- `Observable<Response>` - you handled the request and the response is available from this observable. It *should be "cold"*.
- `null`/`undefined` - you decided not to intervene, perhaps because you wish to intercept only certain paths for the given HTTP method. The service continues with its default processing of the HTTP request.

The `RequestInfo` is an interface defined in `src/in-mem/interfaces.ts`. Its members include:

```
req: Request;           // the request object from the client
collectionName: string; // calculated from the request url
collection: any[];      // the corresponding collection (if found)
id: any;                // the item `id` (if specified)
url: string;            // the url in the request
utils: RequestInfoUtilities; // helper functions
```

The functions in `utils` can help you analyze the request and compose a response.

In-memory Web Api Examples

The test fixtures demonstrates library usage with tested examples.

The `HeroInMemDataService` class (in `test/fixtures/hero-in-mem-data-service.ts`) is a Hero-oriented `InMemoryDbService` such as you might see in an HTTP sample in the Angular documentation.

The `HeroInMemDataOverrideService` class (in `test/fixtures/hero-in-mem-data-override-service.ts`) demonstrates a few ways to override methods of the base `HeroInMemDataService`.