SAMPLE CHAPTER

Aurelia IN ACTION

Sean Hunter

MANNING

www.itbook.store/books/9781617294785

Aurelia in Action

by Sean Hunter

Chapter 8

brief contents

PART 1 INTRODUCTION TO AURELIA 1

1 ■ Introducing Aurelia 3
2 ■ Building your first Aurelia application 26

PART 2 EXPLORING AURELIA.....61

3 ■ View resources, custom elements, and custom attributes 63
4 ■ Aurelia templating and data binding 83
5 ■ Value converters and binding behaviors 104
6 ■ Intercomponent communication 119
7 ■ Working with forms 156
8 ■ Working with HTTP 188
9 ■ Routing 206
10 ■ Authentication 243
11 ■ Dynamic composition 264
12 ■ Web Components and Aurelia 275

13 ■ Extending Aurelia 305
14 ■ Animation 322

PART 3 AURELIA IN THE REAL WORLD 335

15 ■ Testing 337
16 ■ Deploying Aurelia applications 363

8 Working with HTTP

This chapter covers

- Examining the Fetch API browser
- Looking at the Aurelia fetch client
- Intercepting HTTP requests
- Using the Aurelia HTTP client

In the real world, no SPA lives in isolation. SPAs are typically part of an ecosystem that involves a multitude of components, such as REST APIs and other dependencies, both internal to your application and external. The first Aurelia application I built integrated with a backend REST API to fetch application data and statistics, the Octopus Deploy REST API to retrieve a list of servers that we were interested in, and other APIs with information pertinent to the application. Integrating with these kinds of external dependencies brings your application to life.

With the growing popularity of serverless architectures, it's increasingly common to host a simple SPA and use it to knit together a suite of external utilities—from cloud databases to SaaS (software as a service) offerings like Salesforce. The technology required to build these kinds of applications has existed for quite some time, starting with Microsoft's invention of AJAX, back in 2000. The built-in browser API for working with AJAX, the XMLHttpRequest API, is beginning to show its age,

and developers are starting to expect a cleaner and more modern HTTP browser API. Because of this, libraries like jQuery have created higher-level APIs to make it easier to work with HTTP requests in general, and AJAX specifically.

Recently, a more modern native API has started making its way into browsers. It's called the Fetch API and makes working with HTTP much simpler. It provides built-in support for concepts such as CORS- and HTTP-origin header semantics, which were not on the radar when the XMLHttpRequest API was invented. Aurelia provides two packages, `aurelia-http-client` and `aurelia-fetch-client` (which uses the new Fetch API under the hood), that simplify HTTP communication by providing a higher-level API on top of Fetch. The `aurelia-fetch-client` makes it significantly easier to use the Fetch API as part of the Aurelia application architecture due to its support for DI.

In this chapter, you'll learn how to build HTTP interactions into your Aurelia applications, connecting the my-books SPA to the my-books REST API by means of the `aurelia-fetch-client` and `aurelia-http-client` packages.

8.1 Overview of the Aurelia HTTP toolkit

The `fetch-client` and `http-client` plugins have a lot of overlap in terms of core functionality, each providing support for all HTTP verbs. The differentiating factor is the browser API that they're built on. `aurelia-http-client` aims to provide a simple-to-use abstraction over the traditional `XMLHttpRequest` object, which has been in browsers since 2000. This is the object that existing utilities, such as jQuery's AJAX library, are built on.

Because it was created so long ago, many things that we've come to expect from a modern HTTP client API aren't supported. Further, the functionality that is supported is verbose and requires a lot of boilerplate code. In contrast, the `aurelia-fetch-client` plugin is built on the recent Fetch API and provides access to some of the more recent browser APIs, such as service workers and the Cache API. It also provides support for concepts such as CORS.

The recommendation from the Aurelia core team is to use `aurelia-fetch-client` where possible and fall back to `aurelia-http-client` in cases where the functionality you need isn't supported. An example of this is if you require download progress or request cancelation.

Both Aurelia HTTP clients can be globally configured, avoiding the need to repeatedly specify options such as the base URL and credentials that should be applied to every request. Both clients also support *request interception*. Request interception is used to hook into an HTTP request on its way out to an HTTP endpoint (for header manipulation or request logging) and on its way back. Interception provides flexibility in the way HTTP requests are handled in your SPA. An overview of the Aurelia HTTP toolkit can be seen in figure 8.1.

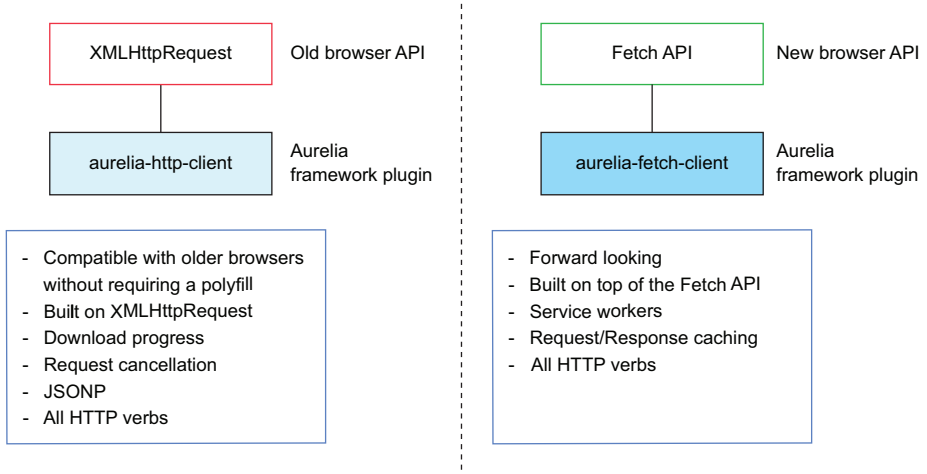

Figure 8.1 Overview of the Aurelia HTTP toolkit. This consists of `aurelia-http-client`, which is built on the XMLHttpRequest API, and `aurelia-fetch-client`, which is built on the Fetch API.

8.2 Using aurelia-fetch-client

`aurelia-fetch-client` is a wrapper over the browser's native Fetch API. This article on the Mozilla Developer Network is a great resource if you're interested in learning more about the Fetch API: https://developer.mozilla.org/en/docs/Web/API/Fetch_API. `aurelia-fetch-client` supports the same methods as the native Fetch API, but it also provides the following advantages:

- *Request tracking*—Every request sent from `aurelia-fetch-client` is tracked, giving you an overview of the HTTP interaction across your application.
- *HTTP interception*—You can intercept and optionally manipulate requests coming in and going out of your application. This is useful for cross-cutting tasks, such as logging or providing feedback to the user, when your application is performing HTTP communication.
- *Injection*—This module is injectable into the services and view-models across your Aurelia application.
- *Default value configuration*—You can set default values, such as a base URL for your API, request headers, or credentials, which are then applied to every request.

8.2.1 Adding fetch to my-books

In the previous chapters, you emulated HTTP interaction in my-books using the combination of an HTTP request to a seed JSON file and simulated backend calls to the `BookApi` service that return hardcoded data. In this section, swap out this fake

backend for a simple REST API built using Node.js, Express.js, and MongoDB. This REST API has the following endpoints:

- `/BOOKS (GET)`—Retrieves a list of books
- `/BOOKS (POST)`—Creates a new book
- `/BOOK/ID (GET)`—Retrieves a specific book by ID
- `/BOOK/ID (DELETE)`—Deletes a book by ID
- `/BOOK/ID (PUT)`—Updates a book by ID
- `/GENRES (GET)`—Lists all genres
- `/SHELVES (GET)`—Lists all shelves

The updated architecture of the my-books application encapsulates HTTP calls to the my-books REST API within the `BookApi` class, which depends on `aurelia-fetch-client`. You can see an overview of this architecture in figure 8.2.

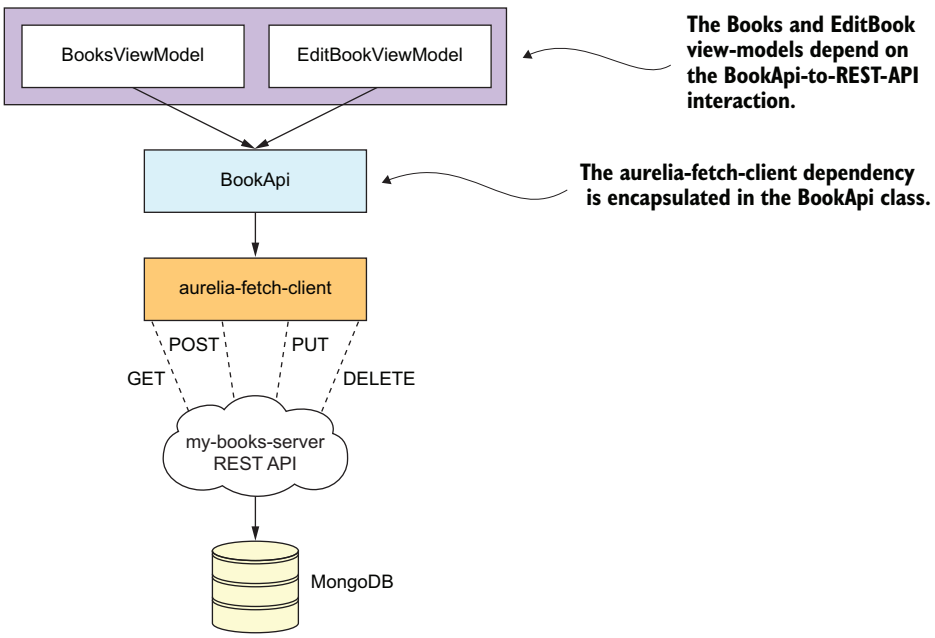

Figure 8.2 The my-books application modified to communicate with the my-books-server REST API via `aurelia-fetch-client`

Before making any changes to my-books, you need to set up the my-books-server, which is available from GitHub. Instructions for how to do this can be found in the appendix and in the GitHub repository at https://github.com/freshcutdevelopment/my-books-server. This is a simple Node.js-based REST API with a MongoDB backend. We won't cover how this server was built because it's outside the scope of this

book. But if you're interested, you can read more on this topic in Simon Holmes' terrific book, *Getting MEAN* (Manning, 2015) at http://mng.bz/87FF.

> **TIP** The Fetch API is a relatively recent addition to browsers. If you need to support users on Internet Explorer (any version other than Edge), or browser versions lower than Edge version 14, Firefox version 39, Chrome version 42, or Safari version 10.1, you'll need to use a polyfill to patch the missing functionality for these browsers. The polyfill fetch library from GitHub is a good option: https://github.com/github/fetch.

Now that you've set up the my-books-server REST API and have it running at http://localhost:8333/api, modify the `BookApi` class to make use of the new endpoints. To begin, import two modules from the `aurelia-fetch-client` package: the `HttpClient` module (which you'll use for HTTP communication) and the `json` module (which allows you to serialize book objects to JSON before they're sent to the backend). With the transition to a REST API rather than the simple books.json seed file, you'll need to configure the base URL that the HTTP client should use when making each of its requests. This is done using the `configure` method on the `HttpClient` module. This method takes a function that returns an `HttpClientConfiguration` object, `http.configure(config => {})`, which you configure to the following options:

- `config.withDefaults ({credentials:...}, {headers...})`—Allows you to configure default parameters to be passed with each HTTP request. Any configuration options available on the Fetch API are configurable here. This is most useful when you have headers or credential options that need to be the same with every HTTP request you send.
- `config.withBaseUrl (url)`—Allows you to configure the base URL used for HTTP requests.
- `config.useStandardConfiguration()`—Sets up reasonable defaults on the `httpClient` object, such as the same-origin CORS policy for credentials.
- `config.withIntercepter()`—Allows you to configure a pre-/post-request interceptor function. We'll cover this shortly.

After configuring the base URL, you need to modify each of the `BookApi` methods to call the my-books-server endpoints rather than sending back hardcoded values. The pattern is similar in each case. Use the `this.http.fetch(URL)` method to make the HTTP call, unwrap the promise, and deserialize the JSON message into an object with the `response.json()` method, returning the resulting value as a POJO to the caller. The HTTP `GET` verb is used by default when you call `httpClient.fetch()`, so in the cases where you're deleting, updating, or creating data, you need to specify the relevant verb (method) as a part of the fetch options.

Additionally, when creating or updating items, you need to serialize the message body using the `json` method. You'll also perform a little housekeeping, removing all simulated latency from the service, because you're no longer working with mock data.

Connecting to a real HTTP API introduces a chance for errors. To handle this, add a catch block to trap errors on each HTTP call, logging the error details to the console. Modify the ./src/services/book-api.js service class as shown in the following listing.

Listing 8.1 Using the my-books-server REST API in `BookApi` (book-api.js)

```
import {HttpClient, json} from 'aurelia-fetch-client';
import {inject} from 'aurelia-framework';

@inject(HttpClient)
export class BookApi{

    constructor(http){
        this.http = http;

        const baseUrl = 'http://localhost:8333/api/';

        http.configure(config => {
            config.withBaseUrl(baseUrl);
        })
    }

    getBooks(){

        return this.http.fetch('books')
                 .then(response => response.json())
                 .then(books => {
                    return books;
                 })
                .catch(error => {
                    console.log('Error retrieving books.');
                    return [];
                });

    }

    getShelves(){

        return this.http.fetch('shelves')
                 .then(response => response.json())
                 .then(shelves => {
                    return shelves;
                 })
                .catch(error => {
                    console.log('Error retrieving shelves.');
                    return [];
                });
    }

    getGenres(){

        return this.http.fetch('genres')
                 .then(response => response.json())
                 .then(genres => {
```

Imports the HttpClient and JSON modules, and injects the http client into the BookApi class via constructor injection

Configures the base URL of the REST API

Switches the getBooks method to fetch from the API endpoint

Switches the getShelves method to fetch from the API endpoint

Switches the getGenres method to fetch from the API endpoint

```
                return genres;
             }).
            .catch(error => {
                console.log('Error retrieving genres.');
                   return [];
            });
    }

    addBook(book){
        return this.http.fetch('books', {
            method: 'post',
            body: json(book)
            })
            .then(response => response.json())
            .then(createdBook => {
                return createdBook;
            })
               .catch(error => {
                   console.log('Error adding book.');
            });

    }

    deleteBook(book){
        return this.http.fetch(`book/${book._id}`, {
                 method: 'delete'
                 })
                 .then(response => response.json())
                 .then(responseMessage => {
                     return responseMessage;
                 })
                   .catch(error => {
                        console.log('Error deleting book.');
                   });


    }

    saveBook(book){
        return this.http.fetch(`book/${book._id}`, {
                     method: 'put',
                     body: json(book)
                  })
                  .then(response => response.json())
                  .then(savedBook => {
                     return savedBook;
                     })
                     .catch(error => {
                        console.log('Error saving book.');
                   });


    }
}
```

Configures the addBook method with the HTTP POST verb

Configures the addBook method with the HTTP DELETE verb

Configures the saveBook method with the HTTP PUT verb

Catches errors and logs to console

TIP In this case, you've handled errors by catching them and logging an error message to the console. By contrast, in a real-world application, you'd want to notify the user that there was an issue connecting to the backend service and potentially retry the request. One way to implement this kind of error notification is to use the Aurelia Event Aggregator to raise an error event, passing along the error message. You can then listen for error events in a component higher up in the hierarchy and add global error-notification logic via an error-notification component or similar. You could also transmit the error to an error-tracking and reporting service, such as TrackJS or Raygun, to give you visibility into errors encountered by users.

With the `BookApi` changes in place, you need to modify the components in the application to work with the slightly revised data structure returned from the REST API and MongoDB. The revision to this structure incorporates the MongoDB database ID format (`"_id": "5991713f95fd5759604ffb7b"`) and refers to genres by ID reference rather than name (`"genre": "5991713f95fd5759604ffb70"`). Because the concepts in the following section have been covered already, feel free to skip these steps and download the complete version from chapter 7 at https://github.com/freshcut-development/Aurelia-in-Action. The changes you'll need to make are as follows:

- Modify the `EditBook` view-model to use the new books data structure.
- Modify the `edit-book.html` view to make use of the new books data structure.
- Modify the `Books` view-model to make use of the new books data structure.

STEP 1: MODIFY EDITBOOK VIEW-MODEL

Modify the contents of the `EditBook` view-model, ./src/resources/elements/edit-book.js.

Listing 8.2 Modifying `EditBook` to use the new data structure (edit-book.js)

```
...
export class EditBook{
    ...
    bind(){
        ...
        this.selectedGenre = this.genres
                                .find(g => g._id == this.book.genre);

        this.selectedShelves = this.shelves
                                .filter(shelf =>
                                        this.temporaryBook
                                            .shelves
                                            .indexOf(shelf.name) !== -1);
    }
    selectedGenreChanged(newValue, oldValue){
        if(!newValue) return;
        this.temporaryBook.genre = newValue._id;
    }
```

Replaces ID with _id to use the database ID returned from MongoDB

Replaces ID with _id to use the database ID returned from MongoDB

Populates the selectedShelves array; these are now objects rather than strings

```
    attached(){
        this.bookSaveCompleteSubscription =
            this.eventAggregator
            .subscribe(`book-save-complete-${this.book._id}`,
                        () =>  this.bookSaveComplete());
    }
    ...
}
...
```

Replaces ID with _id to use the database ID returned from MongoDB

STEP 2: MODIFY EDIT-BOOK VIEW

Modify the contents of the `edit-book` view, ./src/resources/elements/edit-book.html.

Listing 8.3 Modifying `edit-book` to use the new data structure (edit-book.html)

```
...
<label for="shelves" class="mb-2 mr-sm-2 mb-sm-0">Shelves</label>
<select show.bind="editingShelves"
        name="shelves"
        class="form-control mb-1 mr-sm-1 mb-sm-0"
        multiple
        value.bind="selectedShelves">
    <option repeat.for="shelf of shelves"
            model.bind="shelf">
        ${shelf.name}
    </option>
</select>
<button show.bind="editingShelves"...>ok</button>
...
```

Uses object-based multiselect binding rather than string-based

STEP 3: MODIFY BOOKS VIEW-MODEL

Modify the contents of the `Books` view-model, ./src/resources/elements/books.js.

Listing 8.4 Modifying the `Books` view-model (books.js)

```
...
export class Books {
...
  addBook () {
    this.bookApi.addBook({title : this.bookTitle}).then(createdBook => {
        this.books.push(createdBook);
        this.bookTitle = "";
    });
  }

  removeBook(toRemove){

    this.bookApi.deleteBook(toRemove).then(() => {
      let bookIndex = _.findIndex(this.books, book => {
        return book._id === toRemove._id;
      });
```

Modifies the addBook method to call book-api to POST book

Modifies the removeBook method to hit the API endpoint and updates the books array with the result

```
        this.books.splice(bookIndex, 1);
        });
    }
    ...
    bookSaved(updatedBook){
        this.bookApi
            .saveBook(updatedBook)
            .then((savedBook) => {
                let index = this.books
                                .findIndex(book =>
                                        book._id == savedBook._id);

                Object.assign(this.books[index], savedBook);

                this.eventAggregator
                      .publish(`book-save-complete-${savedBook._id}`);
            });
    }
    ...
}
```

Modifies the bookSaved callback method to hit the API PUT endpoint and updates the books array with the result

With this housekeeping taken care of, the application should look the same as it did before, but if you lift the hood and take a look, you can see that the data is now retrieved from the REST API, as shown in figure 8.3.

By adding `aurelia-fetch-client` to my-books, you've seen how a default configuration can be configured on the `httpClient` object and used for each request. You used this default configuration to minimize duplicate code across the HTTP calls, isolating the base URL so that you could define it in one place. You've also seen how to modify the HTTP verb on requests, which allows you to easily create, delete, and update data on any REST API you want to use.

Next, we'll look at another configuration option available on `aurelia-fetch-client`: *interceptors*. You'll use interceptors to log each of the HTTP interactions that you've added to the Aurelia application and observe the requests being made under the hood.

8.2.2 Intercepting and manipulating requests

Interceptors give you a straightforward way to manipulate requests coming to and from your Aurelia application via `aurelia-fetch-client`. You can think of a interceptor like a middleman for your requests, allowing you to take the request or response and modify it in any way (including swapping it out and replacing it with a new request/response entirely, if that suits your purpose). Most commonly, interceptors are used for tasks such as appending request headers and logging. The interceptor flow is depicted in figure 8.4.

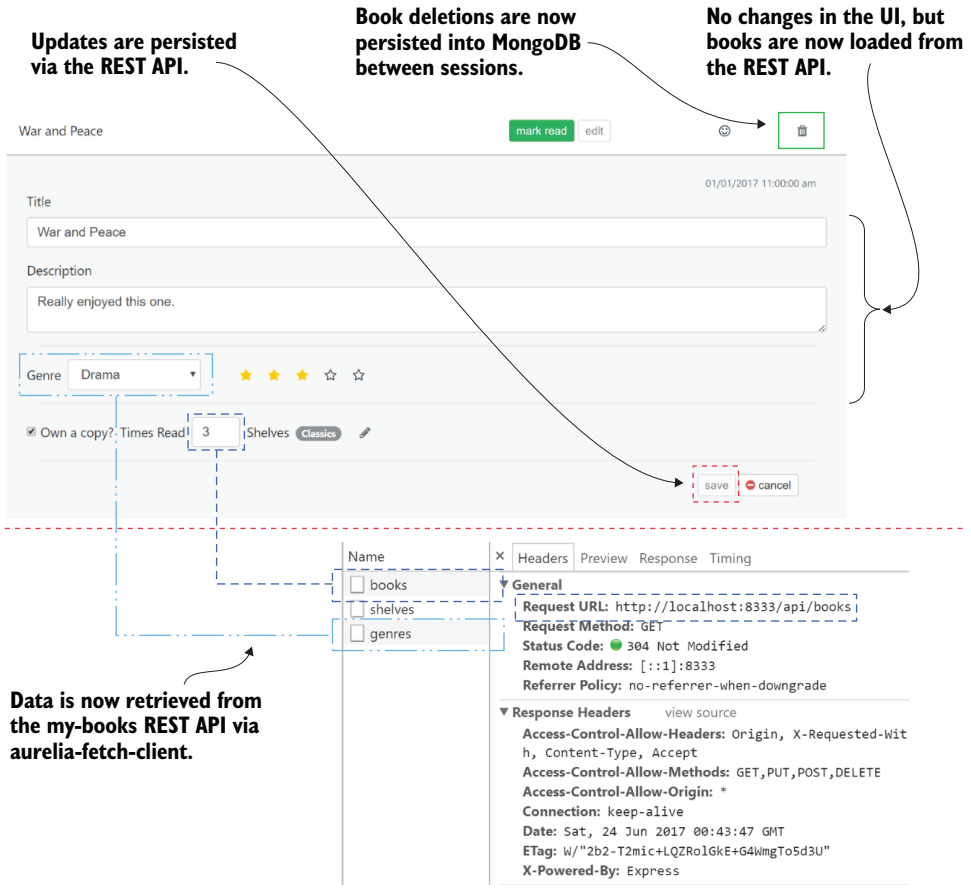

Figure 8.3 Fetching, adding, updating, and deleting books is now achieved via REST calls to the my-books-server API using `aurelia-fetch-client`.

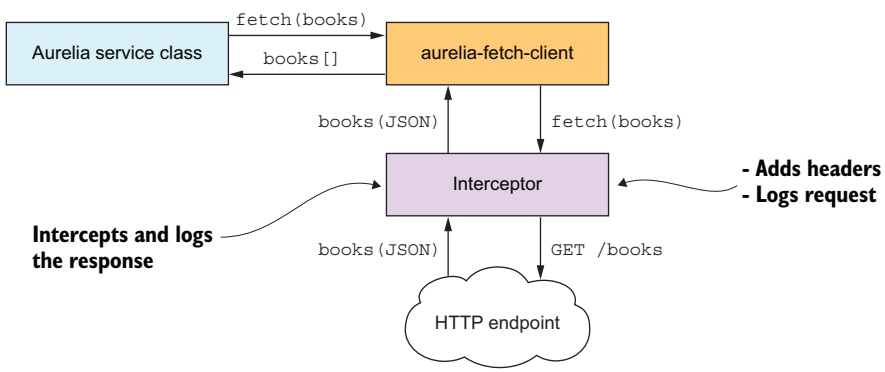

Figure 8.4 Requests made via `aurelia-fetch-client` can be intercepted and manipulated.

To see it in action, modify the `BookApi` class to intercept and log all my-books HTTP interactions. Modify the ./src/services/book-api.js class.

Listing 8.5 Logging HTTP requests from `BookApi` (book-api.js)

```
...
export class BookApi{

    constructor(http){
        this.http = http;

        const baseUrl = 'http://localhost:8333/api/';

        http.configure(config => {
            config.withBaseUrl(baseUrl)
                .withInterceptor({
                    request(request) {
                        console.log("request", request);
                        return request;
                    },
                    response(response) {
                        console.log("response", response);
                        return response;
                    }
                });
        });
    }
    ...
}
```

Adds an interceptor to the http client configuration

Adds a callback request to log the content of each request

Adds a callback response to log the content of each response

Refresh the browser, and then you should see the request and response bodies logged into the Chrome Developer Tools (F12). In this example (figure 8.5), I reloaded the page, causing the initial set of books, genres, and shelves to be loaded. I then deleted and created a book. The pertinent details of the requests are logged, including the type (CORS, in this case, because of calling an HTTP endpoint with a different URL than the Aurelia site), the request URL, the method (or HTTP verb), and, in the case of the response, a status code and a Boolean value to indicate whether the request was redirected. Figure 8.5 depicts a log of each of these interactions.

Aside from using them to analyze and debug HTTP communications in your Aurelia application, you can use interceptors to manipulate HTTP requests. For example, say you wanted to add an `awesome` custom header to each request. One option is to include it in the base configuration, but the drawback is that it would then be included in every request. What if you wanted to include the header only on a `POST`? Easy, let's give it a try. To do this, conditionally add a header to the `awesome-custom-header` HTTP requests, but only when the method is of the POST type. Modify the interceptor in ./src/services/book-api.js and add a custom header for POST requests only.

Figure 8.5 Intercepting and logging requests between the my-books Aurelia client and the Node.js server

Listing 8.6 Adding a custom header to POST requests (book-api.js)

```
export class BookApi{

    constructor(http){
        this.http = http;

        const baseUrl = 'http://localhost:8333/api/';

        http.configure(config => {
            config.withBaseUrl(baseUrl)
                .withInterceptor({
                    request(request) {
```

Adds custom header for POST requests

```
            if(request.method == 'POST'){
              request
              .headers['awesome-custom-header']
                 = 'aurelia-in-action';
            }
            console.log("request", request);
            return request;
        },
        response(response) {
            console.log("response", response);
            return response;
        }
    });
  });
 }
 ...
}
```

You can use the logging that you set up to see this new interception logic in action. Navigate back to the browser, add and delete a book, and check the developer console log. You'll see that the custom header is applied only for the POST request, as shown in figure 8.6.

8.3 Working with aurelia-http-client

`aurelia-fetch-client` solves most of the HTTP communication requirements that you'll come across in your Aurelia development, but you may run into the occasional scenario where the functionality that you need hasn't made it into the Fetch API browser yet. In these cases, you'll need to fall back to the `aurelia-http-client` package. As mentioned earlier, a common case for this is if you need to communicate with a JSONP API.

JSONP provides a mechanism for sharing data between different domains, so it's a common requirement of third-party REST APIs. This makes it a useful tool to keep in your back pocket. To see how this can be used in practice, add a new service class to my-books that retrieves the books API using a JSONP request. Then, add a reference to this new service in the `Books` view-model class to log the results.

Begin by importing the `HttpClient` and configuring the base URL, as you did with the `fetch-client` example. The difference in this case is that you import the module from the `aurelia-http-client` package instead of the `aurelia-fetch-client` package. Then, in a new method, `getBooksJsonp`, make a JSONP call using the `jsonp` method on the `HttpClient` class, `this.http.jsonp('booksjsonp')`. This method takes a URL and a callback-parameter name—the name of the URL parameter that specifies the function used to wrap the JSONP response (which, in your case, is set to `'callback'` in the my-books-server response). If you're interested in learning more about cross-site requests and JSONP, I recommend checking out *CORS in Action* by Monsur Hossain (Manning, 2014), which delves into these concepts in much greater depth, at http://mng.bz/BASc.

Figure 8.6 Custom headers added to POST requests sent via `aurelia-fetch-client`

The shape of the response object is slightly different than the JSON requests that you made with `fetch-client`. In this case, the response body is already deserialized into an array for you, so all you need to do is retrieve the response body from the `responseMessage.response` message. To implement this change, add a new `BookApiJSONP` class under ./src/services/book-api-jsonp.js.

Listing 8.7 Adding the `BookApiJSONP` class (book-api-jsonp.js)

```
import {HttpClient} from 'aurelia-http-client';
import {inject} from 'aurelia-framework';

@inject(HttpClient)
export class BookApiJSONP{
```

Imports the HttpClient class

```
    constructor(http){
        this.http = http;

        this.baseUrl = 'http://localhost:8333/api/';

        this.http.configure(config => {
            config.withBaseUrl(this.baseUrl);
        });
    }

    getBooksJsonp(){
       return this.http.jsonp('booksjsonp', 'callback')
          .then(responseMessage => {
              return responseMessage.response;
          })
          .then(books => {
            return books;
          });
    }
}
```

Configures the base URL

Makes an HTTP JSONP call specifying the URL and the callback

Retrieves the response body

To see the results of this JSONP call, wire up the new service class in the `Books` view-model and log the response, as shown in the following listing, importing the newly created service and loading the books array from the `bind` lifecycle callback.

Listing 8.8 Including the JSONP request (books.js)

```
...
import {BookApiJSONP} from '../../services/book-api-jsonp';

@inject(BookApi, EventAggregator, BookApiJSONP)
export class Books {

  constructor(bookApi, eventAggregator,bookApiJSONP){
    ...
    this.bookApiJSONP = bookApiJSONP;
  }
  ...
  bind(){
    ...
    this.loadBooksJsonp();
  }

   loadBooksJsonp(){
      this.bookApiJSONP.getBooksJsonp()
              .then(savedBooks => console
                                  .log("jsonp books", savedBooks));
  }
  ...
}
```

Imports the BookApiJSONP service class

Loads the books array in the bind() component-lifecycle callback

Logs the result to the console

Figure 8.7 depicts the JSONP-network request, the autogenerated callback-function name injected into the URL callback parameter, and the wrapped JSONP-network response.

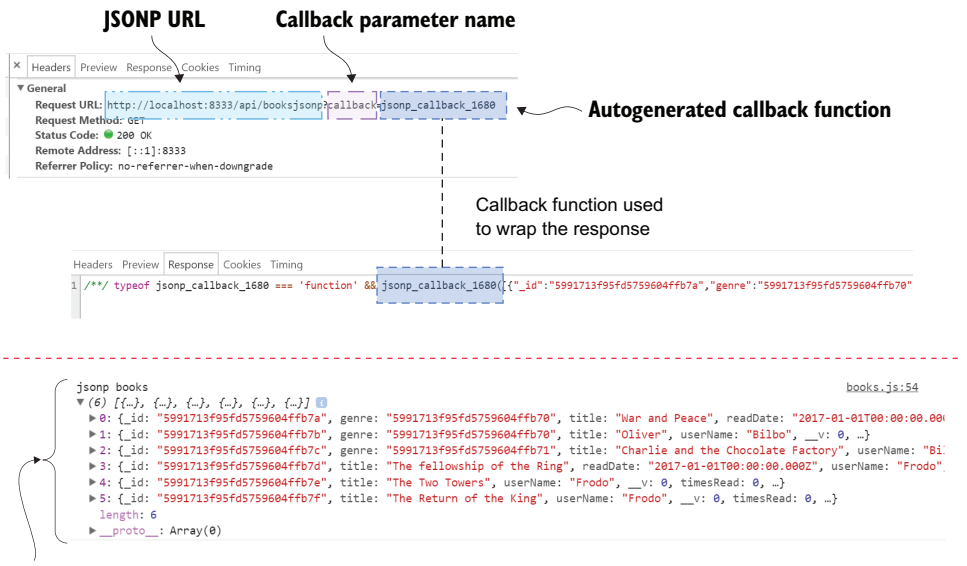

Figure 8.7 JSONP-network call with `json_callback_1680` autogenerated callback method used to wrap the network response

This gives you a taste of the functionality available with `aurelia-http-client`. To see the full API and additional configuration options, such as fluent configuration, you can check out the latest documentation on the Aurelia Hub at http://aurelia.io/docs/plugins/http-services.

With an Express.js backend added to your Aurelia project, you're one step closer to creating a real-world, usable web application, but it still has a long way to go. Two major shortcomings are navigation and authentication. Currently, the site consists of two pages—the homepage and the books page—but the user doesn't have a great deal of indication as to which is active. Additionally, the application is devoid of authentication, allowing any Tom, Dick, or Harry to view any book collection. In the next chapter, you'll remedy both shortcomings, adding a navigation bar and a much-needed authentication system to the my-books application. In doing so, you'll become familiar with the ins and outs of Aurelia's router.

Summary

- Two HTTP modules are provided with the Aurelia framework. These modules sit on top of the `XMLHttpRequest` object (`aurelia-http-client`) and the new Fetch API (`aurelia-fetch-client`).
- `aurelia-fetch-client` is a simple wrapper on top of the Fetch API. It's injectable and, though the name may not make it obvious, it supports the full range of HTTP verbs.

- Because `aurelia-fetch-client` is so new, some features, such as JSONP, aren't supported yet, and you'll need to drop down to the alternative HTTP package: `aurelia-http-client`.
- The combination of these two packages gives you the power and flexibility that you need to meet any HTTP-related challenge when developing your own applications.
- If you run into issues and need to diagnose your HTTP logic, interceptors can save the day. These give you visibility into your Aurelia HTTP pipeline, allowing you not only to trace incoming and outgoing requests, but also to manipulate them on the way through.

WEB DEVELOPMENT

Aurelia IN ACTION

Sean Hunter

Try Aurelia, and you may not go back to your old web framework. Flexible and efficient, Aurelia enforces modern design practices and a modular architecture based on web components. It's perfect for hybrid web + mobile apps, with hot features like dynamic routes, pluggable pipelines, and APIs for nearly every flavor of JavaScript.

Aurelia in Action teaches you how to build extraordinary web applications using the Aurelia framework. You'll immediately take advantage of key elements like web components and decorators when you start to explore the book's running example: a virtual bookshelf. As the app unfolds, you'll dig into templating and data binding the Aurelia way. To complete the project, you'll take on routing and HTTP, along with tuning, securing, and deploying your finished product.

What's Inside

- Templating and data binding
- Communication between components
- Server-side and SPA design techniques
- View composition

Written for developers comfortable with JavaScript and MVC-style web development.

Sean Hunter is a software developer in Melbourne, Australia, with nine years of web-development experience.

"Makes building single-page applications easy and fun."
—Alessandro Campeis, Vimar

"The perfect way to transform your applications into successes."
—Philippe Charrière, Clever Cloud

"All navigators need a map, and this book provides a path into the jungle of production-ready web-development frameworks."
—Joseph Tingsanchali, Netspend

"Learn how to build a single-page app that aims for simplicity, modularity, and convention over configuration."
—Peter Perlepes, Growth

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/aurelia-in-action

ISBN-13: 978-1-61729-478-5
ISBN-10: 1-61729-478-0
54999
9 781617 294785

MANNING $49.99 / Can $65.99 [INCLUDING eBOOK]