# BUILDING HYPERMEDIA APIS WITH ASP.NET

## COURSE SUMMARY

So you've built your HTTP API, and now that it's live, you're suddenly dealing with a whole new set of problems. Do you really need to PUT the entire Customer just to change someone's email address? Why does it take you 25 API calls just to render a shopping cart? How do you find the bottlenecks when just drawing a web page requires fifty HTTP requests? What happens when one of your API consumers accidentally tries to GET your entire customer database? The architectural style known as REST can answer all these questions - and more - but even experienced developers often find it difficult to apply RESTful principles when building real-world applications. In this workshop, we'll explore the elements of REST related to hypermedia and the principle of "hypermedia as the engine of application state" (HATEOAS) - we'll talk about why they matter, what problems they solve, and when you might want to implement them in your own systems. During the workshop, we'll implement a RESTful HTTP API using C# and ASP.NET. Starting with the most basic "hello world" service, we'll cover patterns like content negotiation and resource expansion. We'll compare several popular formats for representing hypermedia in modern web APIs, we'll cover the semantics of PUT, POST and DELETE (and implement support for HTTP PATCH), and we'll look at some of the tools and frameworks that are available to help you design, build and monitor your HTTP APIs. Finally, we'll cover topics like monitoring and security, and we'll discuss the strategic value of building HTTP APIs in modern organisations - so you'll go home not just knowing how to build a great RESTful APIs, but also how to persuade your boss that it's a good idea.

### IN THIS COURSE, YOU CAN EXPECT TO…

- Learn how to design and implement RESTful APIs in C# and ASP.NET
- Learn how to use the full range of HTTP verbs and response codes effectively.
- Learn about the various hypermedia formats that are available for building HTTP APIs.
- Learn about the rich set of tools that can help you design, build and manage your APIs.

### WHAT'S NOT IN THIS COURSE:

To cover the material in the available time, we will be focusing exclusively on building hypermedia APIs in C# using ASP.NET and WebAPI. We won't be talking about VB.NET, F#, SQL, NoSQL, HTML, CSS or JavaScript. We'll be using a couple of small JS apps during the demonstrations and hands-on sessions, and talking about integrating your APIs with other systems, but if you want learn about using AngularJS with Suave.IO and Entity Framework, this is not the workshop for you. 😊

# COURSE PROGRAMME

The hands-on component of this course is broken down into nine modules, spread over two days.

The modules are designed so that every module builds on the code from the previous one – so if you want to start with module 1 and implement everything else yourself, go for it!

But we also want to encourage you to try things out for yourself and to experiment and play around with the various patterns and techniques we'll be talking about – and we also would hate for anybody to get left behind if they're having problems with one of the modules. So we've also included the starting code for every module – you can download ZIP files from the GitHub project releases page, or if you're happy with branching in Git you'll find the code for each module in a separate branch.

## MODULE 1: HYPERMEDIA AS THE ENGINE OF APPLICATION STATE

### WORKSHOP

1. Download the first release of the Herobook application code from http://www.github.com/dylanbeattie/Herobook/. You're welcome to fork this repository to your own GitHub account if you want to commit your own changes back to GitHub to refer back to later.
2. Get the project running locally. Verify that you can use the API explorer tool to make simple API GET requests, and get valid JSON back. Take a moment to review the project structure and see how everything fits together.
3. Modify the `/profiles` endpoint so that it accepts an `index` and a `count` parameter on the query string and returns the relevant subset of the data
4. Modify the `/profiles` endpoint so that the `index`, `count` and `total` values are included in the JSON response, and the values are exposed as an items property
5. Add the `_links` collection with the hypermedia links that will allow your API consumers to navigate through the data set.

### DISCUSSION POINTS

- Do linked resources have to be on the same server?
- How could you design a partition scheme that would split your user profile data across multiple database servers?

## MODULE 2: RESOURCE DECOMPOSITION

### WORKSHOP

- Modify the `/profiles/{username}` endpoint so that it converts the database response into a dynamic object.
- Add the _links collection, containing the canonical URI (self) of the profile resource.
- Add the additional friends, photos and statuses properties in the code, so that when you retrieve /profiles/{username}, you get back a JSON object graph containing a user profile and their friends, statuses and photos.

Now, use hypermedia to add navigation links for those associated properties, and verify that you can browse around a set of profile data using the API explorer.

### DISCUSSION POINTS

Are there situations where this model would be the right solution?

What are the similarities and differences between this model and the idea of 'eager loading' or 'deep loading' entities when working with an object-relational mapper?

## MODULE 3: RESOURCE EXPANSION

### WORKSHOP

Continuing from the code created during the 'resource decomposition' workshop, implement resource expansion, so that your API consumers can specify via the query string which properties should be included inline.

Take the time here to play around with some ideas for specifying multiple resources. How would you specify that you want to expand photos and friends? What if you wanted to expand friendships to include the full profile of the associated user?

### DISCUSSION POINTS

- Could you do inline expansion of resources that are hosted on a different server?
- Could you separate the elements of the implementation, so that the resource expansion happened on a front-end caching server or load balancer?
- Could you apply the 'donut caching' pattern to inline resource expansion?

## MODULE 4: REPRESENTING ACTIONS WITH HYPERMEDIA

### WORKSHOP

Extend the API server code to support Richardson model level 2:

- Use hypermedia to add a **`DELETE`** action to the /profiles/{username} endpoint.
- Use hypermedia to add a **`PUT`** action to the **`/profiles/{username}`** endpoint. Clients should supply the complete JSON of the new version of the profile, and this should overwrite the existing profile data
- Use hypermedia to add a **`POST`** action to the **`/profiles`** endpoint, so hypermedia clients can add new users to the system

### DISCUSSION POINTS

What would you need to do to support Richardson maturity model level 3?

How could you expose an action that supports different content types? Say we wanted **`POST /profiles`** to accept both **`application/json`** and **`application/x-www-form-urlencoded`**?

## MODULE 5: IMPLEMENTING HTTP PATCH

### WORKSHOP

Implement PATCH support for /profiles/{username}, with the following specification:

- Any field included in the JSON post should replace the associated field in the profile resource.
- Fields that are not included in the JSON post should be ignored
- Fields that are included and set to null, or to an empty array, should replace the associated field, effectively removing a property from the profile resource.
- The PATCH action should be added to the _actions collection of the /profiles/{username} endpoint

### DISCUSSION POINTS

- What would RMM level 3 look like for a PATCH operation? How would you communicate the required data fields and format to the API client?

## MODULE 6: CONTENT NEGOTIATION

### WORKSHOP

Extend the /profiles/{username}/photos/{photoId} endpoint to support the following content types:

- image/png
- image/jpg

For image representations, pick your favourite superhero, find a cool picture of them online, and use this in your project. You can either convert the image to multiple formats and store these separately, or you can write some .NET code that will convert a 'master' image format into the required format in response to each request.

If the server does not have a suitable representation available, it should return an HTTP 406 Not Acceptable, and include in the response a set of available representations so that the client can decide whether to use an alternative representation.

### DISCUSSION POINTS

- In module 4, we talked about creating a POST method that could support both JSON and x-form submissions. How could you create generic HTTP actions – PUT, POST, DELETE – that support content negotiation?

## MODULE 7: CACHING AND LAYERING

### WORKSHOP

Extend GET /profiles/username endpoint to support HTTP `If-Modified-Since` and `If-None-Match` headers, and to return HTTP `304 Not Modified` if the client already has an up-to-date version of the resource.

Extend `PUT` and `PATCH` on `/profiles/username` to require an `If-Modified-Since` header, and to return an HTTP `409 Conflict` if one of these requests does not reflect the latest status of the underlying resource.

### DISCUSSION POINTS

- How does caching work with resource expansion? Could you design a system that supported cache invalidation for only one part of an expanded resource request? How would you decide the "last modified" date for the response to a request for an expanded resource?
- How would you implement If-Modified-Since and If-None-Match support for objects stored on a filesystem, like the photographs we used in the module on content negotiation? What if you wanted to avoid having to access the filesystem for every request?

## MODULE 8: SECURITY AND AUTHENTICATION

### WORKSHOP

Add authentication to the `PUT`, `PATCH` and `DELETE` methods on the `/profiles/{username}` endpoint, so that users have to be authenticated to make changes to their own profile. Use a default password for every user.

Use HTTP Basic authorization, so the username and password are supplied base64-encoded in a standard Authorization: header. Requests without this header should return a `401 Unauthorized`. Requests including the wrong credentials should return an HTTP `403 Forbidden`.

### DISCUSSION POINTS

- What's wrong with this model? What would you need to do to run a model like this in production?

## MODULE 9: API VERSIONING

### WORKSHOP

- Modify the Profile entity in the underlying data model so that firstName and lastName are exposed as separate properties. (You'll need to think carefully about how you handle this in persistence, otherwise the DemoDatabase will break – just like if you were making this change on a real production system.)
- Implement API versioning support for the /profiles/{username} and /profiles/ endpoints, so that clients that indicate they support version 2 of our API will see separate firstName/lastName properties, and clients that only support version 1 will still see a simple name property.

Use either of the RESTful methods of API versioning we described in the demo for this section.

### DISCUSSION POINTS

- How would you use hypermedia to communicate to API clients that a newer version of the API is available?
- How would you

## TESTING AND MONITORING APIS

### WORKSHOP

Add a /status endpoint to your API, that returns a JSON document showing:

- The number of user profiles currently stored in the system
- The number of profiles updated in the last five minutes

Using your preferred testing framework, create a test that you can run against a live API, which can double as both an integration test and a monitoring endpoint.

1. GET the root endpoint (/) of your hypermedia API
2. Find the link named "profiles"
3. GET the resource identified by the HREF in the profiles link
4. Find the first item in the items collection of the profiles resources
5. GET the first profile in the collection, using the _links/self/href property exposed by that profile item.
6. GET each property that's navigable from the profile resource (photos, statuses, friends)