# TODO/Bugs

|  |  |
| --- | --- |
| Description | Assigned To |
| Here are the packages to list in CH4. Let’s wait until we’re farther along before we make final corrections.  DO THIS FIRST: update-package Microsoft.AspNet.WebApi WebApi2Book.Web.Api  install-package automapper WebApi2Book.Common  install-package log4net WebApi2Book.Common  install-package nhibernate WebApi2Book.Data.SqlServer  install-package fluentnhibernate WebApi2Book.Data.SqlServer  install-package automapper WebApi2Book.Web.Api  install-package log4net WebApi2Book.Web.Api  install-package nhibernate WebApi2Book.Web.Api  install-package fluentnhibernate WebApi2Book.Web.Api  ADD THIS: install-package Ninject.Web.Common.WebHost WebApi2Book.Web.Api  DELETE THIS: install-package ninject WebApi2Book.Web.Api  DELETE THIS: install-package ninject.web.common WebApi2Book.Web.Api  install-package log4net WebApi2Book.Web.Common  install-package nhibernate WebApi2Book.Web.Common  install-package ninject WebApi2Book.Web.Common  install-package ninject.web.common WebApi2Book.Web.Common | Jamie |
| In the Data Access section of ch3 we mention a separation between repositories and UOW. We aren’t separating these. Everything is done in the ISession. | Brian |
| Remove hack from AddTaskQueryProcessor added in CH5 once security is implemented. | Brian |
| Update TaskCreatedActionResult to support multiple content types, once we get to content negotiation. TaskCreatedActionResult was introduced in CH5. | Brian |
| Add location link so the TaskCreatedActionResult picks it up. Use a link service instead of the hack in AddTaskMaintenanceProcessor. | Brian |
| Some captions read, “POST Request (abbreviated)” and some lack the “(abbreviated)”. Make this consistent. All should have the “(abbreviated)”. | Brian |
| Table formatting – is it consistent enough? | Brian |

# Code Examples

|  |  |  |  |
| --- | --- | --- | --- |
| Feature | Comments | Code Complete | Covered in Book |
| Context-sensitive hyperlinks |  | X | TODO CH 7 |
| Paging of results |  | X | TODO CH 7 |
| partial updates |  | X | TODO CH 7 |
| api versioning |  | X | X |
| legacy SOAP support |  | X | TODO CH 8 |
| How to handle non-resource APIs using REST | TasksWorkflowController | X | X |
| How to best expose relationships between resources |  | X | TODO CH 7 |
| entities vs web models |  | X | X |
| IHttpActionResult |  | X | X |
| async filter | Audits task re-activation. | X | X |
| input validation | Use this as an intro:  We've made great progress, but we're still lacking input validation. Subject is a required field in the Task database table, and we're wasting valuable server resources by allowing messages with no subject to reach the database if they are only to be rejected by it. Let's take care of that problem next. | X | TODO CH7 |
| attribute-based routing | Including prefixes, constraints, and coexistence with global, convention-based routing | X | X |
| content negotiation |  | X | TODO CH8 |
| CORS | cors vs jsonp (jsonp is hacky). enablecors/disablecors attributes |  | TODO CH6 or CH9 |
| ASP.NET Identity | JAMIE!!! |  | TODO CH6 |
| Basic Auth |  | X | X |
| Bearer Auth |  |  | TODO CH6 |
| custom legacy auth | use route-specific handler |  | TODO CH8 |
| CSRF | Are we going to include this in code examples? JAMIE!!! |  | TODO CH6 |
| authorization attribute |  | X | X |
| require ssl attribute | Optional! |  | TODO CH6 |
| Removing sensitive data from response message |  | X | X |
| Automatic lifetime management for database connections and transactions |  | X | X |
| NHibernate |  | X | X |
| tracing |  | X | X |
| global error handling |  | X | X |
| web page to illustrate how to consume the api | Shane may help with this |  | TODO CH9 |
| testing (unit, integration) |  |  | TODO CH9 |
| comments/documentation | Code only |  | n/a |

# Original Book Proposal to Apress

### Brief Book Description (two sentence summary --in simple, non-technical language):

Learn how to leverage the features and capabilities of the ASP.NET Web API to build a RESTful web service from start to finish. This book will first explain the REST architecture, and then build on that knowledge, the Web API, and a few other patterns, tricks, and techniques, to go from a blank slate to a fully functional, secure, and versioned REST service.

### Long Description (300-350 words) statement that defines the benefits offered to the reader:

The ASP.NET MVC Framework has always been a good platform on which to implement REST-based services, but the introduction of the ASP.NET Web API Framework raised the bar to a whole new level. Now in release version 2.1, the Web API Framework has evolved into a powerful and refreshingly usable platform. This concise book provides technical background and guidance that will enable you to best use the ASP.NET Web API 2 Framework to build world-class REST services.

As with the Web API 2 Framework itself, ASP.NET Web API 2: Building a REST Service from Start to Finish improves upon its predecessor, ASP.NET MVC4 and the Web API. New content in this edition includes

* New capabilities in Web API 2 (currently version 2.1).
* Swappable data storage. NHibernate, Entity Framework, and Dapper with Web API are featured.
* Support for partial updates, or PATCH.
* API versioning.
* Support for legacy SOAP-based operations.
* How to handle non-resource APIs using REST
* How to best expose relationships between resources
* JSON Web Tokens, CORS, CSRF, and OAUTH

Also, in response to feedback we received on the first edition, we removed or relegated to an appendix the non Web API content that was deemed to be superfluous or distracting.

So get ready for authors Jamie Kurtz and Brian Wortman to take you from zero to REST service hero in no time at all. No prior experience with ASP.NET Web API is required; all Web API-related concepts are introduced from basic principles and developed to the point where you can use them in a production system. A good working knowledge of C# and the .NET Framework are the only prerequisites to best benefit from this book.

Although some examples include ASP.NET MVC features, the emphasis is on the Web API and not MVC. We recommend Pro ASP.NET MVC 5 if you need help on topics relating to MVC and/or CSS, HTML, JavaScript, jQuery, the Razor view engine, HTML Helpers, model binding, etc.

### Why is this an important topic? How large is the market? Please cite any market statistics or other relevant sources.

RESTful services are growing in importance at an incredible rate right now, mostly in support of the variety of devices out there today. REST style services are being used to provide backends for nearly all mobile applications, as well as, the newer style of web sites being built - i.e. single-page apps.

### Target audience: Level and Technical Focus

Beginner to Intermediate; REST, .NET, C#

What The Reader Will Learn (in 5 or fewer bullet points):

* Introduction to the REST architecture
* How to design a REST API
* New capabilities in Web API 2, including: better error handling, attributed routes, CORS, and better OData support
* Understanding Web API controller activation
* Automatic lifetime management for database connections and transactions
* Using NHibernate, Entity Framework, and Dapper with Web API
* Easily secure a REST service, using standards-based authentication and authorization, JSON Web Tokens, or a custom implementation
* Supporting legacy SOAP callers with Web API
* How to expose relationships between resources
* Supporting partial resource updates under REST
* Web API versioning

### Detailed Table of Contents:

1. [ASP.NET](http://asp.net/) as a Service Framework

2. What is RESTful?

3. Designing Our Sample REST API

4. Building the Environment and Creating the Source Tree

5. Controllers, Dependencies, and Managing the Database Unit of Work

6. Securing the Service

7. Dealing with Relationships, Partial Updates, and Other Complexities

8. Leveraging new Web API Features to Simplify Your Code and Support Legacy Systems

9. Putting It All Together

### Time Frame for Completion:

Work on book throughout March and April and May

### Estimated Page Count:

Between 150 and 200

# Representative Feedback from Version 1

Comments from Amazon…

## Negative

* …degenerates into page after page of setting up NHibernate mappings, dependency injection, security and authentication, and, I kid you not, Log4Net logging. Page after page that could, and should have been used to show the ins and outs of the Web API, modeling a UI to utilize the RESTful links that are supposed to drive your application, crafting AJAX requests for PUT and DELETE, etc, etc, were instead used to fill up reams and reams of Log4Net XML setup, and DI mappings.
* You will spend most of your time learning about the authors favorite O/RM tools, Dependency Injection libraries and logging tools and how to configure them. You WILL NOT learn how to make REST calls using Ajax. I was looking to learning how to perform basic CRUD using a REST service without all the extraneous tools the authors covers. If you are looking to actually learn to use the Web API. DO NOT buy this book I repeat DO NOT by this book.
* Unfortunately, after stepping through a lot of complex project creation and configuration steps in Chapter 4, in Chapter 5 the author clearly gives up on the building, stops including steps with full sources, and instead refers you to download and explore his pre-created project.
* I was hoping to see a little more of a full fledge start to finish style book as the title as well as up to chapter 4 of the book suggest but that wasn't the case until you download the source code accompanying the book.
* Other than that though this book desperately needed someone to keep the author on track, and focused on what the book was supposed to be about: using the Web API.

Summary: Not focused on Web API, too many concepts, not enough step-by-step.

## Positive

* The book starts with a great and needed introduction of REST.
* This book is definitely absolutely utterly recommended for those who want to learn WebAPI MVC4 and also how to build a greatly architected WebAPI solution.

Summary: Keep the intro. Keep the good architectural practices.

## Suggestions

* I would also like to see how and where validation is done, and a chapter on how you would set up the same code for unit testing. I too would love to see an example of a simple html page consuming the newly built web service through a JavaScript Ajax call.
* And maybe a little more illustration of the authors examples might help out instead of relying on the book downloads
* This book would have been better if if was only 40 pages and just a step by step how to.
* I would be interested to see what the next book from this author would be like and hopefully it expands into some of the topics and concepts a little more in depth and sticks to the style that the first half of this book adheres to

Summary: Stick to highlighting the framework, show a client AJAX call, show [unit] testing, show step-by-step w/o *requiring* user to look at project code in github.

# Recommendation for Version 2

* Focus heavily on the framework. How to build on it. How to test what you’ve built.
* Show how to consume using an AJAX call(s). Don’t just rely on Fiddler. As we discovered with ccm, Fiddler hides issues you’ll encounter with real clients (e.g., CORS).
* Do NOT spend time on swappable data storage. Stick with one, probably NHibernate (though SQL Azure was requested by one guy).
* Do NOT spend time on OData. Nobody asked for it.
* Once we begin getting into discussing the implementation, let’s take vertical slices instead of horizontal slices (at least as much as possible). So, for example, show step-by-step how to GET a Task. Including testing and AJAX call.
* Similarly, let’s show the following in great detail, and only the following, rather than taking on explaining the whole task service. This focus will make it so readers don’t have to refer to the external project code to begin developing their own APIs. They can refer to that code to see all of the details if they want to, but most of the code is pretty much repetitive; after you’ve seen one GET you’ve seen enough to write another yourself.
  + GET Task(s)
  + POST Task(s)
  + PUT/PATCH Task(s)
  + DELETE Task(s)
  + GET Task User, to illustrate relationships
  + Perform a “non-resource” action (e.g., close a task).
  + Perform a legacy operation via the Web API.
    - Currently getting all Statuses and getting Status by id. May instead do this with Task, time permitting, b/c tasks are a bit more interesting.
* Drop Category and Priority. They may be good in a robust task tracking system, but they are overkill in illustrating Web API and good architectural practices. So get rid of any reference to them... globally.
* Table of Contents:
  + 1) ASP.NET as a Service Framework
    - Cut the ASP.NET MVC stuff. It’s confusing. For example, “This book will cover REST principles sufficiently for you to build services using ASP.NET MVC.” No, we’re not even going to use MVC. MVC and Web API are two different frameworks.
  + 2) What is RESTful?
    - Leave pretty much as-is, with the following changes. Readers rated this consistently well, even the ones who gave the book < 3 stars.
    - In CcmApi we're using 402 to indicate that a business processing rule has been violated. I think Adam Haskell or Jarrett recommended this, so I listened!
  + 3) Designing Our Sample REST API
    - Let's deal with handling non-resource APIs using REST. Specifically, starting, completing, or reopening a task. This time these will be different than simply updating or deleting it. There are business rules to follow. So the Status attribute is not editable via CRUD methods.
    - For collections, make the returned collection a "1st class object" with its own set of properties. Instead of putting the "All" link on each of the bare elements, put it on the collection which will now contain the elements. For the collections returning all Tasks or all Users (i.e., entities that can have a lot of records), add PageNumber, PageSize, and PageCount to the collection. Put pageNumber and pageSize in the query string so callers can page through results.
    - The new book and sample project code will use link services. This will help keep the list of links more dynamic than in the first book, where the mappers simply add a list of static links w/o any regard to current state.
    - ASP.NET Identity will be used instead of Membership Provider.
    - Good until the “Choosing Architecture Components” section. Stop there. Those details will move naturally into the text as needed, as we begin implementing vertical slices.
  + 4) Building the Environment and Creating the Source Tree
  + 5) Controllers, Dependencies, and Managing the Database Unit of Work
  + 6) Securing the Service
    - We should have 3 users with 3 different auth levels:
      * Do and see everything
      * Can't create or delete tasks, but can see and modify task data
      * Can't modify anything, and can't see who is assigned to the Task (e.g., for an external customer... don't want them to see which particular employee is working on their Task)
    - forms auth cookies not natural for apps and devices external to website, so will not cover it
  + 7) Dealing with Relationships, Partial Updates, and Other Complexities
  + 8) Leveraging new Web API Features to Simplify Your Code and Support Legacy Systems
  + 9) Putting It All Together

# Legacy SOAP Support

On a recent project we were tasked with creating a REST-based Web API for an existing banking system. There were a couple of notable requirements:

* The new REST-based API and the legacy SOAP-based web service must be packaged together in a single application.
* Existing customers must be able to benefit from the new features of the new application without affecting any external integration points. So, for example, a bank should be able to upgrade without introducing any breaking changes to its message translation service that bridges the Visa financial network and the banking system.

Understanding that SOAP requests are merely HTTP POST messages with an XML body, we were able to easily meet these requirements. An example illustrating how we did it will be the focus of this chapter.

## The Controller

Not needed! Just use a route handler. As an additional benefit, this skips the overhead of model binding and controller, action selection.

## Security Using a Route-Specific Handler

Route-specific handler

* <http://www.strathweb.com/2013/08/message-handlers-scoped-per-route-in-asp-net-web-api/>
* Using this for custom legacy auth. Does not burden other routes with overhead. Returns 401 (Unauthorized) if auth fails.

# Raw Material for Appendix

A lot of this is stuff cut from the 1st edition text…

Data Access

There are quite a few options available in .NET when it comes to data access and object

persistence on SQL Server. Most of these options fall into one of two categories: using the

various SqlClient objects (e.g., SqlConnection, SqlDataAdapter, and SqlCommand) with stored

procedures or embedded SQL; or using an Object Relational Mapper (ORM). Sometimes the

two approaches are used together, but more often developers choose one or the other.

What I’ve found to work best most the time, on applications where the database

schema isn’t too crazy (and there isn’t any requirement to squeeze every last ounce out

of performance), is to use *NHibernate* for most of the basic Create-Read-Update-Delete

(CRUD) work—and then supplement that with stored procedure calls, as needed.

The separation between the Unit of Work object (i.e., the ISession) and your repository

objects is the main benefit, in my opinion. This is especially the case within web or

service applications, where you want a given call to execute within the context of a single

database session and transaction.

NHibernate is particularly well-suited for this scenario; in fact, it comes with built-in

support for associating Unit of Work instances with ASP.NET or WCF call contexts. This

benefit allows the developer to configure the lifetime and injection of ISession objects

just once—and then never have to mess with them again. As you’ll see, using an IoC

container along with NHibernate and its link with ASP.NET allows for near-transparent

yet very reliable database-transaction management.

As you work with NHibernate in your task management sample REST service, be

sure to note the way in which the code manages the lifetime and injection of ISession

instances. This is arguably the key to much of the value of using NHibernate within

ASP.NET MVC applications. Also, if you use NHibernate and maintain a separation of the

data model classes from the actual repositories and mapping files, then you can create a

true provider-agnostic data access interface layer. I’m a huge fan of code and architecture

cleanliness and simplicity, so the fact that you can completely separate the caller from

anything SQL Server–related is a very important to me (even if I never intend to support

any other platform beyond SQL Server).

IoC Container

These days, whether working in .NET or in Java, not using an IoC container of some

sort can almost be considered foolish. Again, there are certainly special circumstances

that might require you to manage dependencies yourself. But generally speaking, using one of the available frameworks is pretty much a no-brainer. If you’re one of the many

developers who don’t know what an IoC container is used for, I suspect this method of

managing dependencies and using such a tool might be the most valuable thing you will

learn from this book. Seriously, dependency injection tends to be the anchor on which

most other good patterns and practices hang.

I’ve used a few different IoC containers, and the one I like best is *Ninject*. It is

extremely simple to use, contained within a single DLL, and configured with a fluent-like

syntax (e.g., when registering the various type mappings and object instances). It also has

the ability to register a callback method for type resolution, which will come in handy

when you want to make NHibernate ISession objects available for constructor injection

into your repository objects.

Logger

If you ask 10 people for their opinion on the best logger, you will likely get 11 different

answers. There are many ways to write log messages from within an application,

including built-in .NET tracing. What’s most important with loggers is that you can

configure them via configuration files—*not* by changing code and recompiling. As such,

any logger worth considering will offer some degree of the following capabilities—all

configurable at runtime (or during deployment):

·· Filtering

·· Log levels

·· Routing

·· Formatting

Filtering allows you to write code using certain tags or categories, and then filter them

out at runtime. For example, a filter value might be a certain section of the application called

authentication, or a certain class called Mvc4ServicesBook.Web.Api.TasksController.

So the code itself would have this category either hard-coded or inserted by the logger,

and then the configuration file could be used to turn those categories on or off. In this

manner, you can decide at run time to log certain types of messages. Obviously, you

don’t want to be required to update and recompile your code in order to log more

information.

Log levels are essentially a special case of filtering, where the different log levels

are typically DEBUG, VERBOSE, INFO, WARNING, ERROR, CRITICAL, and FATAL (or some

combination thereof). This particular filter is used often enough that most loggers make it

a prominent part of the API. For example, *log4net*’s logger class contains methods named

after those levels: Debug(), Verbose(), Info(), and Error().

Routing describes the ability for log messages to be sent to different targets by simply

updating the configuration file. In other words, the code itself knows nothing about

where the log data will eventually end up—it just sends log messages to the logger. Then,

at deployment or runtime, the configuration file is updated to route log data to one or

more providers. Examples of some providers include text file, XML file, Windows event log, SMTP server/e-mail, and SQL Server databases. Typically, you can also configure

different routes for different filters, as in these examples:

·· Send info and debug messages to a log file

·· Send warning and error messages to the event log

·· Send fatal messages to an ops team via email

Again, these types of routes and filters should be updatable via updates to a

configuration file.

Finally, a logger should offer the ability to use format strings for log messages. This

means that—again, via the configuration file—you can tell the logger exactly how you want

the log messages to look. These log format strings are usually similar to what you might

use when specifying the format string in a call to the .NET function, String.Format().

Further, it should be possible to utilize predefined tokens for logger-provided pieces of

information. For example, the following should be available to you: the current date and

time (including the ability to specify a date/time format string), logger name, ThreadId, log

level, currently executing method, class name, and so on. Here’s an example from log4net:

%date %-5level [%thread] %logger - %message%newline%exception

My logger of choice has been log4net for quite a while now. Beyond the capabilities

I just described, log4net’s logger is an ILog interface. This means you can use

dependency injection for supplying logger instances to any class in the application.

Some loggers (e.g., Microsoft’s Enterprise Application Library Logging Block) don’t offer

an interface-based logger. So you either have to wrap the entire logging API with an

adapter; or, you are forced to statically bind your class code to a specific logger. At least,

that was my experience.

Bottom line: The log4net logging framework is simple to use, provides a logger

interface that can be used with IoC containers, comes with numerous options for routing

and filtering, and has been used all around the world in thousands of .NET applications

for many years.

DateTimeAdapter

I’m a firm believer in avoiding static calls at all costs—and that includes static calls

against .NET Framework classes. The only place a static call should be made is within

an adapter or factory class. The DateTime.Now property in .NET is a perfect example of

something that seems so trivial to use, yet can get you tied up in knots many times over if

you’re not careful; this is especially so when it comes to writing unit tests.

Instead, you need to use the Adapter pattern, wrap the DateTime class in an

appropriate injectable interface, and create a corresponding adapter implementation.

Then, anytime a class needs to get the current system time, it will use dependency

injection to obtain an implementation of your IDateTime interface and call Now

(or, UtcNow) on it. That way, the unit test code can force the “current time” without having

to resort to setting the Windows system clock during test execution.

In the MVC4ServicesBook.Common project, add the following interface and

corresponding implementation:

public interface IDateTime

{

DateTime UtcNow { get; }

}

public class DateTimeAdapter : IDateTime

{

public DateTime UtcNow

{

get { return DateTime.UtcNow; }

}

}

For the task-management service, you’ll use UTC time. However, you are free to add

other adapted properties, as well. Even so, this adapter is the only place in the entire code

base that you see a call to DateTime.Now (or DateTime.UtcNow).

Composition Over Inheritance

There have been many books written in the last 10 or 15 years talking about various evolutions in the practice of object-oriented programming. One of the major shifts of the past 15 years is captured in the oft-quoted phrase, “composition over inheritance.” In short, an application’s code will be much less coupled—and much more testable and maintainable—if it utilizes composition of many smaller classes instead of relying on a base class for desired behaviors. When a base class is packed with a wide variety of functionality, your class—even if it’s small—implicitly takes on the base class's full surface. Changes to the base class require full test regression of all classes that inherit from it—even if those classes aren’t using the base functionality that was modified.

Further, relying on a base class for behaviors forces the base class to take on more than one responsibility—and this is in direct conflict with the Single Responsibility Principle. You see, if you limit a base class’s responsibility (i.e., its functionality) to only one thing, but child classes require the use of more than a single behavior, then you are forced to use composition in providing these behaviors. And at that point, using a base class loses most of its intended appeal.

If the concept of “composition over inheritance” is new to you, I encourage you to read Head First Design Patterns by Eric Freeman et al (O’Reilly, 2004). I think you will find its principles and presentation rather freeing, especially if you come from a strong 1990’s style OO background.

# Raw notes…

Upgrading

<http://www.asp.net/mvc/tutorials/mvc-5/how-to-upgrade-an-aspnet-mvc-4-and-web-api-project-to-aspnet-mvc-5-and-web-api-2>

CORS

Also, note that according to the CORS spec, setting origins to "\*" is not valid if **SupportsCredentials** is true (<http://www.asp.net/web-api/overview/security/enabling-cross-origin-requests-in-web-api)>

Partial updates thru patching

Idea from <http://weblog.west-wind.com/posts/2012/Aug/30/Using-JSONNET-for-dynamic-JSON-parsing>

Formatters

* Applying security masks using custom formatter
* Look at global.asax ConfigureFormatters. We use JsonMediaTypeFormatter, CamelCasePropertyNamesContractResolver, StringEnumConverter. Can also enforce json-only by replacing the IContentNegotiator (e.g., JsonOnlyContentNegotiator).

Caching

<http://byterot.blogspot.com/2012/06/aspnet-web-api-caching-handler.html>