NancyFX Succinctly

by Peter Shaw

Syncfusion | Technology Resource Portal

NancyFX Succinctly

By
Peter Shaw

Foreword by Daniel Jebaraj

Important licensing information. Please read.

Technical Reviewer: Stephen Haunts

Copy Editor: Courtney Wright

Acquisitions Coordinator: Hillary Bowling, online marketing manager, Syncfusion, Inc.

Proofreader: Morgan Cartier Weston, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

As an early adopter of IT back in the late 1970s to early 1980s, I started out with a humble little 1k Sinclair ZX81 home computer.

Within a very short space of time this small 1k machine became a 16k Tandy TRS-80, followed by an Acorn Electron, and eventually, after going through many other different machines, a 4mb arm powered Acorn A5000.

Eventually, after leaving school and getting involved with DOS-based PCs, I went on to train in many different disciplines in the computer networking and communications industry.

After returning to University in the mid-1990s and gaining a BSc in Computing for Industry, I now run my own consulting business in the north east of England called Digital Solutions Computer Software Ltd. I advise clients at both a hardware and software level in many different IT disciplines, covering a wide range of domain-specific knowledge from mobile communications and networks right through to geographic information systems, banking and finance, and web design.

With over 30 years of experience in the IT industry within many differing and varied platforms and operating systems, I have a lot of knowledge to share.

You can often find me hanging around in the Lidnug .NET users group on LinkedIn that I help run, and you can easily find me in the usual places such as Stack-Overflow (and its GIS specific board) and on Twitter as @shawty_ds, and now also on Pluralsight where my various videos are available. I also write for codeguru.com where I produce the regular .NET Nuts and Bolts column, which I invite you to drop by and leave a comment saying hello.

I hope you enjoy this book, and learn something new from it.

Please remember to thank Syncfusion (@Syncfusion on Twitter) for making this book (and others in the range) possible, allowing people like me to share our knowledge with the .NET community at large. The *Succinctly* series is a brilliant idea for busy programmers—if you enjoyed what you read, then tell your friends, if you didn't then please do tell me what needs to improve.

Introduction

Welcome to this Syncfusion *Succinctly* series e-book about NancyFX. Over the course of the next 100 or so pages, you'll become acquainted with the NancyFX microframework for .NET.

I'll explain what NancyFX is all about, where it came from, and what it can do for you as a .NET developer. NancyFX is more than just another web framework, and I'll teach you some of the many tricks that make it as easy as possible for you to create stunning “web-enabled” applications on the .NET platform.

The name “NancyFX” will be used interchangeably with “Nancy,” the latter being the more frequently used version of the name for current versions. Unless otherwise stated where I refer to Nancy or NancyFX, I'm referring to the framework.

The “Super-Duper-Happy-Path”

The creators of NancyFX even have their own catch phrase, and it's one that you'll hear often as you start to work with NancyFX, and for good reason.

NancyFX is Designed to "Just Work"

The "super-duper-happy-path" is the phrase used to describe the ease, simplicity, and ethos behind NancyFX (which you'll find out more about soon), and has become an almost religious expression among long-time users of the toolkit.

Code Samples

I will be presenting various snippets of code throughout this book, all of which are based around a sample NancyFX application I have written specifically to show off Nancy's capabilities.

We will start out quite basic, and gradually get more advanced as we work our way through the chapters, exploring further into the framework as we go.

Because NancyFX is such a modular framework, you will only need to go as far as you need to in order to learn the concepts you wish to understand. If all you need to learn is the basics of using the REST framework side of things, then you will not by any means need to read on to later chapters to gain further knowledge of the concepts being discussed.

All the code snippets presented in this book will be available in the final application, and where possible, I'll make a note of exactly which file to look in, and at which line. The application is available on my own GitHub repository at https://github.com/shawty/nancyfxbook.

Chapter 1 What is NancyFX?

Nancy is a microframework for the .NET platform. Heavily inspired by the Sinatra framework available in the ruby world, Nancy provides a low-ceremony, aggravation-free toolkit for "web enabling" your applications, irrespective of which .NET language you use to build them.

You'll notice that I use the term “web enabling,” and this is with good reason. Nancy is not just a web application framework designed to build web sites or API endpoints; it is a complete framework that can be used to add HTTP-based server facilities to everything from simple console mode applications right through to large-scale enterprise sites running on platforms such as Umbraco and ASP.NET MVC.

Nancy can be hosted in traditional scenarios, such as classic IIS7 with ASP.NET, and it can be self-hosted. Self-hosting is particularly interesting because it means normally invisible applications (such as Windows services) can now have a web interface attached to them for administration purposes.

Nancy can also be built on, and will happily run on, Linux and Unix systems under the Mono framework. This means that you can (or at least should be able to) run your Nancy-enabled code on Android and Mac OS. I'm going to stop short of saying you can definitely do this with those two operating systems, however, since I personally have never tried using Nancy on either Android or Mac OS.

This book will primarily focus on running Nancy on the Windows platform using Visual Studio. However, I’ve tried most of the code you'll be introduced to on Ubuntu Linux and on a Raspberry PI running the "Raspbian" version of Linux, and both have been shown to work without issue.

You might already by asking yourself whether you should use Nancy instead of another framework, such as MVC, ASP, Ember, or ServiceStack. How do you know which is best? The truth is that there is no "best framework" that suits every task you may have to address during your work as a software developer—you just choose the best framework for the task at hand.

What I do know, however, is that Nancy offers a large chunk of functionality that is incredibly easy to pick up and use.

As way of a comparison, when I first found out about Nancy (way back in 2011, about version 0.8 or 0.9), I learned everything I needed to know and got up and running with it in about half a day. Several years after I first used it, ASP.NET MVC is still an on-going learning process.

Nancy's capabilities at a glance

So what can Nancy do? Here is a short list of what’s built into the basic framework:

- Rest-based routing
- View page substitution with a built-in, simple view engine
- Static file serving

- Multitenancy service hosting
- Rest API authentication (basic, forms and token-based)
- Flexible model-binding
- Multi-format content negotiation
- Cryptographic key and phrase generation
- SSL certificate-handling
- Async task-handling
- Dependency injection

This list is just the tip of the iceberg; everything you see mentioned here can all be overridden with custom code because of the modular nature of the toolkit. You can easily plug-in your own dependency injection container, or provide your own extended authentication schemes.

With the extra third-party assemblies that are available to add to Nancy, you get the following too:

- OAuth-based authentication
- Connection to third-party diagnostic and logging providers (such as Glimpse and ELMAH)
- Cross-platform integration to things like NGINX, Apache, and other non-Windows application stacks
- Anything else you can imagine and write an adapter for

That last point is not just there for effect—you can easily override anything you want (or need) to. Every interface in Nancy is exposed publicly, and for the best part wired up automatically, meaning all you generally need to do is write a class that implements a given interface, and Nancy will find it and do the rest.

The last major trick that Nancy has is hosting. Nancy can be hosted in multiple scenarios; the basic list on the Nancy Wiki looks like this:

- ASP.NET
- Windows communication foundation
- Microsoft Azure
- OWIN
- Self-hosting (standalone apps)
- Umbraco
- NGINX
- Anything that supports FastCGI

Like with everything else, there's an interface you can implement to provide your own hosting providers if needed.

Summary

In this chapter, you learned what the NancyFX framework is and what its capabilities are. In the next chapter, you'll learn a bit more about the back story behind Nancy’s creation, as well as a little bit more about its ethos, ideals, and the "super-duper-happy-path."

Chapter 2 Nancy as a REST Framework

As you can see by now, Nancy has two primary use cases that define its existence.

The first of these is to use the framework as a general purpose, REST-based framework in place of toolkits such as ASP.NET Web API or REST.

By default, Nancy provides first-class routing and content negotiation facilities, which you'll learn more about in the coming chapters. It's not just about providing rest end points, however; a big part of the equation is the ease with which it allows you to support them.

Many toolkits claim to implement REST, when in reality they don't (at least not by the official W3C standard). While this is not a problem that prevents REST from being used as an architectural pattern, it does nevertheless detract somewhat from what it means to implement REST in a manner that satisfies the W3C standards.

For instance, given a URI pointing to a collection, like http://example.com/people/, many toolkits will implement a **`Get`** request to obtain the collection, and **`Post`** for anything that constitutes an action that potentially modifies data behind the scenes. Nothing more than **`Get`** and **`Post`** verbs are generally implemented, and even if other verbs are permitted, there's usually so many hoops that have to be jumped through to make it happen that it's often not worth the effort required to do so.

> ***Tip: If you need to catch up on the basics of HTTP and how it works, there's another Succinctly series book available on the subject: HTTP Succinctly, by Scott Allen, is well worth a read.***

Nancy flips this notion on its head and makes it easy to implement any verb you want by specifying it in your code. Later, I'll provide some full examples on routing. For now, however, the following code happily implements a REST-based interface to manage a collection that follows the W3C specification correctly:

Code Listing 1

```
public MP3Player()
  : base("mp3player")
{

  Get["/genres/"] = _ =>
  {
    // Implement code here to retrieve all genres from the database
    return HttpStatusCode.OK;
  };

  Put["/genres/"] = _ =>
  {
    // Implement code here to store all genres to the database
```

```
    return HttpStatusCode.OK;
  };

  Post["/genres/"] = _ =>
  {
    // Implement code here to add a new genre in the database
    return HttpStatusCode.OK;
  };

  Delete["/genres/"] = _ =>
  {
    // Implement code here to delete all genres from the database
    return HttpStatusCode.OK;
  };

}
```

The clarity with which routes are defined is quite startling when you're used to working with other toolkits that hide action methods behind obfuscated function names and depend on complex setup procedures to find where everything is.

The definitions are typically so clear and easy-to-see that you can almost read down a class implementing a set of routes and see at-a-glance exactly what responds to what, and how.

Nancy implements ALL the standard HTTP verbs in this manner, not just the four in the previous code listing. It implements, but is not restricted to, the following (which many other toolkits do not on the grounds of security):

- Head
- Trace
- Options
- Connect
- Patch

Given what you have learned already, it should also come as no surprise that it's a simple matter of adding a custom class that overrides one or more of Nancy's standard interfaces to easily implement your own verbs, making it even easier to implement custom functionality. In fact, one of Nancy's major strengths is its ability to define DSLs (domain-specific languages) using simple REST-based concepts and clearly written code.

As a REST toolkit, Nancy excels at letting you get on and implement what you need without ceremony or fuss.

Don't get me wrong—it's not all smooth sailing. There are one or two scenarios where you do have to do some extra work. Under IIS, for example, I am led to believe that IIS6 and earlier don't actually implement some of the verbs available, so if you're hosting using ASP.NET, you may not actually get the verb passed on to you. I am also led to believe that when hosting under IIS, you may experience conflicts with WebDAV where the only resolution is to either not use the conflicting verbs, or to disable WebDAV on the machine where the conflict arises.

In both of these cases, however, third-party conditions prevent Nancy from performing as intended; where no problems of the aforementioned types exist, implementing the exact interface you need is not a problem.

One of my own NancyFX projects does just this to drive a GPS module connected to a RaspberryPI single-board Linux computer.

I use the setup to get an accurate atomic clock using the GPS satellite network, from a GPS receiver attached to the roof of my house. My custom Nancy-based server implements the custom verbs **`Time`** and **`Drift`**, which I use to get the current time and time drift distance, respectively.

Nancy does everything it can to make implementing a sensible REST-based language as easy as possible, which means in the age of the "semantic web," you can now make HTTP endpoints that genuinely describe their own meaning, and it can be as easy or as difficult as you need it to be.

Should you immediately go out and replace your existing REST framework with Nancy? Well, that depends entirely what your objective is.

Because Nancy plays amazingly well alongside most (if not all) of the other REST frameworks available, you'll likely find that you can easily continue to use your existing technology while introducing Nancy-based technologies seamlessly.

I have upgraded projects using Nancy where the users of the associated interfaces had absolutely no idea that their interface had changed for the better. I achieved this by building out all new functionality from a defined point onwards using Nancy. As this was completed, new versions of the original functionality were also built using Nancy, but with slightly different URIs.

At a predefined low-use time, the new URIs were renamed to map to the old URIs, and the entire project deployed and activated in a very short space of time, resulting in identical functionality, but with much-improved performance.

Summary

In this chapter, you saw some of the ways Nancy works amazingly well as a replacement or supplemental addition to your current REST framework. In the next chapter, we will continue this exploration and see how Nancy makes an equally good addition to a general-purpose web framework.

Chapter 3 Nancy as a Web Framework

So what differs with Nancy for use as a web framework? Quite a lot, actually.

The process of adding restful endpoints for your pages and routes does not change when using Nancy as a web framework, so in this respect, you really don't need to learn anything new. Instead, like with everything else that Nancy does, there are modular parts that you can include into the mix that enable features suitable for building a full-blown web site or application.

Nancy will happily serve up static files and data-based views (just like ASP.NET MVC) based on the routes you define. In fact, you can even go so far as to define an alias or route that refers to a specific static file that already exists, rather than dynamically generating one.

Take the following example. Imagine I have the following in my Nancy bootstrapper (you'll learn more about these later in the book):

```
protected override void ConfigureConventions(NancyConventions
nancyConventions)
{
  Conventions.StaticContentsConventions.Add(
    StaticContentConventionBuilder.AddFile(
      "/jquery",
      "scripts/Jquery-2.1.1.min.js"));
}
```

I can then refer to this file in any HTML pages or views I produce using the following:

```
<script src="~/jquery"></script>
```

At first glance, this might not seem like anything special, but what happens if you decide to update the version of jQuery you use in your site?

Given the choice between going through every page, template, and view, and changing every occurrence of **"~/script/jquery-2.1.1.min.js"** to **"~/script/jquery-3.0.0.min.js"** (or whatever the new version might be), or replacing a single line in one class and re-compiling that class, I know straightaway which one I'd prefer.

Don't forget that because this file mapping is defined within regular C# code, you can also now define that mapping using the same code, rather than specifying the two strings.

You could, for example, add a simple entry to your **web.config** file, which you then read and populate, allowing you to easily change the version in one place without recompiling, or you could even store the mapping in a database—the possibilities for managing this are endless.

It's not just single files you can map, either; you can map entire directory structures. In the demo app I've written to accompany this book, you'll find the following in the bootstrapper code:

Code Listing 2

```
protected override void ConfigureConventions(NancyConventions
nancyConventions)
{
  Conventions.StaticContentsConventions.Add(
    StaticContentConventionBuilder.AddDirectory("/scripts", @"Scripts"));

  Conventions.StaticContentsConventions.Add(
    StaticContentConventionBuilder.AddDirectory("/fonts", @"fonts"));

  Conventions.StaticContentsConventions.Add(
    StaticContentConventionBuilder.AddDirectory("/images", @"Images"));

  Conventions.StaticContentsConventions.Add(
    StaticContentConventionBuilder.AddDirectory("/", @"Pages"));

}
```

What this does is set four global rules stating that any requests for **`"/scripts"`** go to a folder called **scripts** in the folder where the compiled Nancy binary is located. Likewise, the same is true for **`"/fonts"`** and **`"/images".`**

However, the rule to handle **`"/"`** is redirected to a folder called **pages**, so a request for **`"/home.html"`** will look in **./pages/home.html** for the HTML content.

I'll cover this in more detail later when we discuss **`Views`**, but for now, know that by default you don't have to do any of this. You can create a folder called **Content** and put everything in there; Nancy will automatically look for and use that folder and its contents as a static file store if it exists.

What does this mean for the developer looking to add Nancy to his or her project?

Well, it means you can install Nancy (and no other modules) via NuGet, create a folder called **Content** in your application, start adding HTML, JavaScript, style sheets, and anything else you need to that folder, press the F5 key, and start hosting that content.

Take note that this Content folder does not just work with ASP.NET hosting. The same holds true even if you are hosting using WCF, self-hosting in a Windows service, or hosting on Azure. The actual hosting platform does not matter at all; this is a default Nancy convention that “just works.”

This is not the only trick Nancy offers to help with a full-blown web app.

Nancy provides:

- Dynamic views via multiple view engines
- Simple model binding
- Content and data validation
- Basic session management

Out of the box, and just as is possible with the restful features, everything is open and extendable via one of the many public interfaces the framework exposes, should you wish to extend it.

Summary

In this chapter, you learned about some of the features that make Nancy an ideal choice for building entire websites and web applications. In fact, while writing this book so far, I've actually heard a number of people refer to NancyFX as the "NodeJS of .NET" because it brings many of the features found in Node to the .NET stack.

In the next chapter, you will get to write your first Nancy-based web application using yet more of the "super-duper-happy-path," via the use of the Nancy projects' pre-made Visual Studio templates.

Chapter 4 Quick Start (Using the Nancy Templates)

Before we get into the internal details of how Nancy works, I'm quickly going to introduce you to the Nancy templates available for use under Visual Studio.

It's very easy to build out the initial scaffolding and required NuGet additions just by making a few clicks, in exactly the same manner you might create an initial MVC application, or traditional Windows Forms application.

Install the templates pack

The Nancy templates come in the form of a Visual Studio extension, available from the Visual Studio online gallery. Navigate to **TOOLS** > **Extensions and Updates** in Visual Studio:

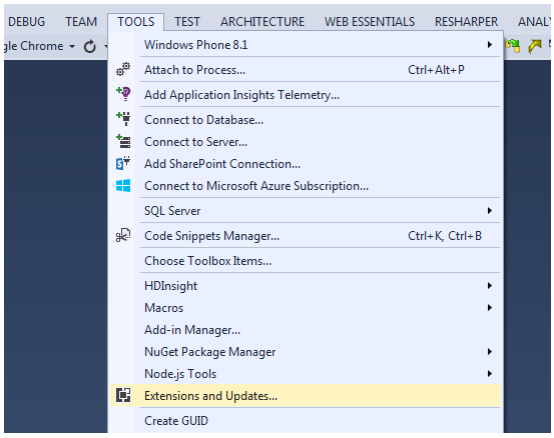


Figure 1: Tools menu required to get to extensions and updates.

This should open the Extensions and Updates dialog box:

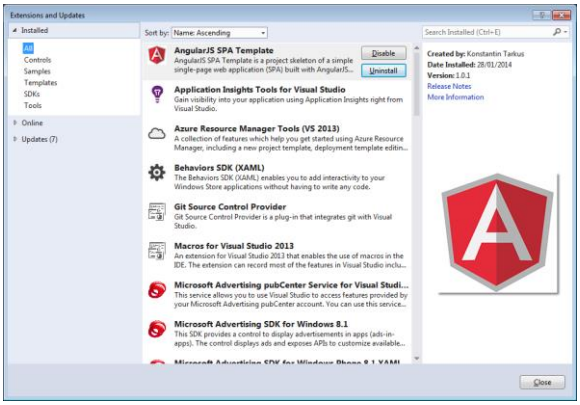


Figure 2: Extensions and Updates dialog box

Like with most of the other dialog boxes available in Visual Studio, there's a Search option in the upper-right corner.

Enter **Nancy** into the **Search** field and press return. After a few seconds, you should see the option to install the Nancy templates appear in the dialog box.

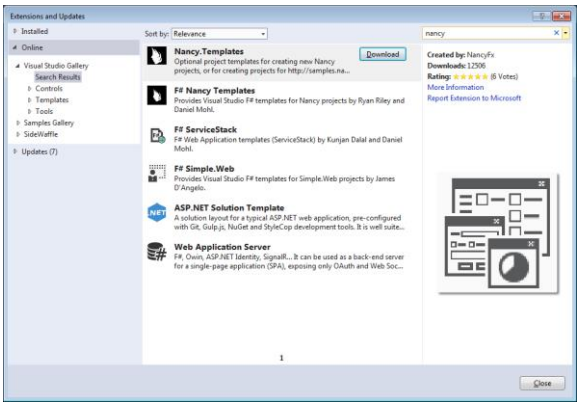


Figure 3: The Extensions and Updates dialog box showing the Nancy templates

Click **Download** and your default browser should then open and download the extension ready to install.

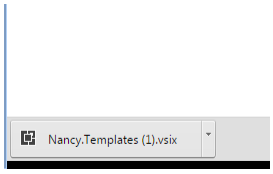


Figure 4: The Nancy templates downloaded in the browser

In my case, I'm using Chrome, so the template installer appears in the lower-left corner; this will be similar for other browsers. If you don't see it, try looking in the folder on your PC where downloads are usually saved.

Once you have the installer, the next step is to close Visual Studio and click on the installer to install the add-on. You don't always need to close down Visual Studio when installing add-ons like this, but I always find that it's usually better to do so, as it often means less problems.

If everything has gone okay so far, you should see the templates installer fire up:

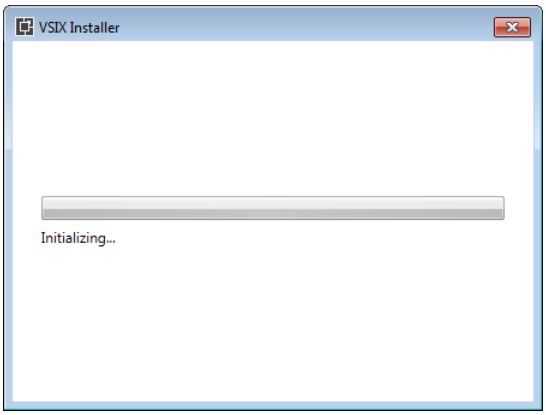


Figure 5: The installer for the templates starting up

In my case, since I already have the templates installed, I'll get the initialization screen, then a message telling me it's already installed.

Unless you've installed them before, you'll get a small wizard that will guide you through the process. Just follow the wizard's prompts, and when you restart Visual Studio, you'll find you have the templates available.

Using the templates

Depending on how your copy of Visual Studio is set up, you may or may not have a tree item on the left of your New Project dialog box, so the easiest way is usually to search by project type (just as you did previously when installing the add-on).

In the Search field in the top-right of your New Project dialog box, enter **Nancy** and press **Return**. You should get something like the following:

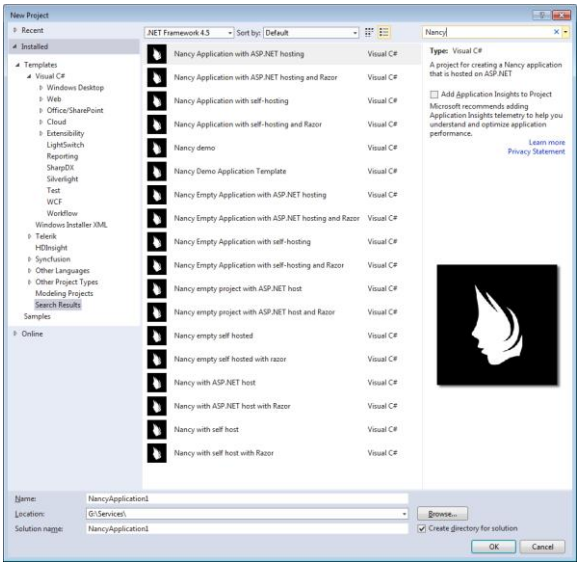


Figure 6: New Project dialog box showing the installed Nancy templates

At this point, it's just a matter of choosing the template you need, depending on what kind of start you want to make.

For example, if you're going to run your application using ASP.NET on an IIS server instance, then you need to choose a template that's set up for ASP.NET hosting. If you're going to self-host or use Nancy in a Windows service, WPF app, or other standalone application, then you need to choose the appropriate type of hosting. Whichever template you choose to work with, everything you do from this point on is transferable.

Nancy is completely independent of its underlying hosting technology, so if you define routes, bootstraps, and other bits of functionality in a standalone app using Razor, all of that is reusable (usually without change) in an ASP.NET app.

Remember that Nancy also works on Mono, and even though I won't be covering it in this book, any projects you create (especially standalone projects) should be easily portable to any platform that supports the Mono runtime.

Summary

In this chapter, you saw the "super-duper-happy-path" in effect once again, in the form of the template pack available for Visual Studio users to get started quickly and easily.

In addition to the official Nancy template pack, if you're a SideWaffle user, there are some Nancy templates contained within that, too.

Tip: Project SideWaffle is a web-based template extension pack for Visual Studio, containing templates for…well, everything. Take a look at the video on the SideWaffle website for an introduction. In the next chapter, we'll continue digging into the very core of Nancy, starting with its routing capabilities.

Chapter 5 Routing

Routing is where most of the magic happens in Nancy, and defining routing modules are the meat and bones that typically make up a Nancy application. Defining routes in a Nancy application is quite a bit different than defining them in something like an ASP.NET MVC application.

Taking ASP.NET MVC as an example, you typically create a class known as a controller. This class provides routing by convention in most cases. By defining your controller class name and the names of the methods within that class, you define the "route path" the code in the class will respond to.

Take the following example:

Code Listing 3

```
using System;
using System.Linq;
using System.Web.Mvc;

namespace Intranet.WebUi.Controllers
{
  public class HomeController : Controller
  {
    public ActionResult Index()
    {
      return View();
    }

  }
}
```

This snippet of code from a standard ASP.NET MVC application defines a result called **`index`** in a route called **`home`**, meaning that to route a request to this code, you would likely use

`/home/index`

as the request path. With Nancy, things are a little different.

First off, all routes defined in a Nancy application must inherit from the base class **`NancyModule`**. Secondly, rather than have a separate class for each route (as in the MVC example), you define your routes in the constructor of your module class, using restful verbs to define the route type.

Unfortunately, this can lead to classes built completely of large constructors, but there are various ways and means to get around that.

Before we go any further however, we need to cover the concept of…

Restful verbs

If you're used to seeing web requests made only from a web browser, you might not realize that underneath it all is quite a complex protocol. If you're doing any work at all in web development, then you've most likely heard of this referred to as the HTTP Protocol. HTTP works by using a set of verbs that represent certain actions the client wishes the server to take on its behalf.

There's a full W3C specification on how all the verbs are intended to work and what they represent, but I don't recommend reading it if you're new to the subject. Instead, I recommend *HTTP Succinctly* from the Syncfusion *Succinctly* series library.

In the context of using this with Nancy, however, we don't need to go into a lot of detail. In fact, in order to build an application that is considered REST-compliant, only the following verbs need to be used:

- GET
- POST
- PUT
- DELETE

GET, as the name suggests, is used to retrieve data from the service; likewise, **DELETE** is used to request data be deleted. **PUT** and **POST**, however, often cause confusion.

Following the spec, **PUT** is taken to mean "put an entire replacement for a resource in place," whereas **POST** is taken to mean "post an addition to a given resource." Many developers creating restful-based applications, however, don't use anything more than **GET** and **POST** for much of what they need to do.

Nancy takes this one step further, and unlike most REST/web-based frameworks, lets you define your own verbs, in a sense allowing you to create your own domain-specific language that operates over HTTP.

Let's dissect an example Nancy route module and see what makes it tick.

Our first Nancy route module

Take a look at the following piece of code:

Code Listing 4

```
using Nancy;

namespace nancybook.modules
{
  public class BaseRoutes : NancyModule
  {
    public BaseRoutes()
    {
      Get[@"/"] = _ => Response.AsFile("index.html", "text/html");
```

```
    }
  }
}
```

When it comes to Nancy, it really doesn't get much simpler than this.

That entire eleven-line class, when added to a Nancy project, will serve the standard plain HTML page called **"index.html"** and located in a folder called **"Content"** to anyone that makes a request to the **"/"** root path of the application it's contained in.

Content is the default place that Nancy will look to find any files you return using this method. We'll see a little more about this in the next chapter on views, but for now, just ensure that your project has a folder called Content, and that you place your HTML files for this chapter in there.

Putting aside the usual namespace and class parts of the code, the two main points that make this a Nancy route module are the addition of **: NancyModule** after the class definition, and the **Get** rule placed in the constructor.

The **Get** rule means that this will respond to HTTP calls made using the **GET** verb, as mentioned in the previous section, and the code following it would be executed in response to the path for that **GET** request being the root path.

If you wanted to expand the module to handle **GET**, **POST**, **PUT**, and **DELETE**, you would do something similar to the following:

Code Listing 5

```
using Nancy;

namespace nancybook.modules
{
  public class BaseRoutes : NancyModule
  {
    public BaseRoutes()
    {
      Get[@"/"] = _ => Response.AsFile("index.html", "text/html");
      Put[@"/"] = _ => Response.AsFile("index.html", "text/html");
      Post[@"/"] = _ => Response.AsFile("index.html", "text/html");
      Delete[@"/"] = _ => Response.AsFile("index.html", "text/html");
    }
  }
}
```

The path or route that the module needs to listen for is enclosed in the square parenthesis of the call to the verb being used. So, for example, you could modify the module to something like:

Code Listing 6

```
using Nancy;

namespace nancybook.modules
```

```
{
  public class BaseRoutes : NancyModule
  {
    public BaseRoutes()
    {
      Get[@"/allpeople"] = _ => Response.AsFile("index.html", "text/html");
      Put[@"/allpeople"] = _ => Response.AsFile("index.html", "text/html");
      Post[@"/newperson"] = _ => Response.AsFile("index.html",
"text/html");
      Delete[@"/singleperson"] = _ => Response.AsFile("index.html",
"text/html");
    }
  }
}
```

For the purposes of demonstrating the code, everything (at least for now) responds with a static page. In reality, you would probably want to respond to each with appropriate code and responses to perform the actual functions requested. You will see more on the subject of responses in a later chapter.

You can also specify a common path for your module. Imagine, for example, we had the following:

Code Listing 7

```
using Nancy;

namespace nancybook.modules
{
  public class BaseRoutes : NancyModule
  {
    public BaseRoutes()
    {
      Get[@"/single"] = _ => Response.AsFile("index.html", "text/html");
      Put[@"/single"] = _ => Response.AsFile("index.html", "text/html");
      Post[@"/new"] = _ => Response.AsFile("index.html", "text/html");
      Delete[@"/single"] = _ => Response.AsFile("index.html", "text/html");
    }
  }
}
```

You can quickly see that things become difficult to read. What's more, if you are trying to create a consistent approach to your API, you might want to reuse the **`single`** and **`new`** URL terminology for other resources your application may have to deal with.

Once again, Nancy makes this an easy problem to solve by calling the base constructor of the **`NancyModule`** parent class with the root path from which everything in your route module should continue.

Code Listing 8

```
namespace nancybook.modules
{
  public class BaseRoutes : NancyModule
  {
    public BaseRoutes() : base("/people")
    {
      Get[@"/single"] = _ => Response.AsFile("index.html", "text/html");
      Put[@"/single"] = _ => Response.AsFile("index.html", "text/html");
      Post[@"/new"] = _ => Response.AsFile("index.html", "text/html");
      Delete[@"/single"] = _ => Response.AsFile("index.html", "text/html");
    }
  }
}
```

By adding this base call, in one move you've changed all the URLs your module will respond to from:

- **GET /single**
- **PUT /single**
- **POST /new**
- **DELETE /single**

to:

- **GET /people/single**
- **PUT /people/single**
- **POST /people/new**
- **DELETE /people/single**

With a bit of creative thinking and some clever software construction, you could even reuse one controller in many different places.

Before we move on, a word of warning on "restful verbs" is needed. Some hosting platforms (including ASP.NET/IIS) do not enable some of the more uncommon verbs by default.

In the previous examples, if you're running on IIS, you'll find that **PUT** and **DELETE** will actually result in the server returning the following error: **503 Method not allowed**.

This is not a fault of Nancy, but is due to the fact that IIS only allows the two most common verbs (**GET** and **POST** by default, on the grounds of security. It's not difficult to fix it, however; take a look at this Stack Overflow post to find out how.

Most of the different resolutions can be found this post, and are easy enough to understand. You may also find that some methods cannot be called against the regular HTML file response we're using; this will be resolved using views, as we'll see in the next chapter.

Route parameters

Just as with other web toolkits, you can set your Nancy routing modules up to receive various items of data. This data can range from simple URL parameters such as a record ID or blog category on a **GET** request, right through to a complex object on a **POST** or **PUT** request.

In this chapter, we'll look at simple route-based data, and leave complex data objects for a later chapter.

You've already seen how easy it is to change a URL in a Nancy module to respond to a given request, but what if that request needs to pass in something like a record ID? In our previous examples, you saw a routing example that might be used with a single type of database record, the type that you might use to manage a single database record using a REST interface in your application. Getting access to something like an ID for this purpose is incredibly easy.

Let's take another look at our previous example, but this time just with the one get request available:

Code Listing 9

```
namespace nancybook.modules
{
  public class BaseRoutes : NancyModule
  {
    public BaseRoutes() : base("/people")
    {
      Get[@"/single"] = _ => Response.AsFile("Pages/index.html",
"text/html");
    }
  }
}
```

If you know C#, you might recognize that following our route definition, we're actually using a Lambda to provide the code we wish to run when Nancy calls that route. If you don't know what a Lambda is, don't worry; you don't need to know what they are in order to make use of them here.

The important thing to recognize is the underscore symbol placed between the two equal symbols.

While not a requirement, it's become somewhat of a self-imposed standard by many users of Nancy to use the underscore to mean "I don't care about" or "I'm not using" any parameters for this route. However, if you were to set the route up to accept a parameter, you could still use the underscore to access it. Many people like to use something like the word "parameters," as the following shows:

Code Listing 10

```
namespace nancybook.modules
{
  public class BaseRoutes : NancyModule
  {
```

```
    public BaseRoutes() : base("/people")
    {
      Get[@"/single"] = parameters => Response.AsFile("Pages/index.html",
"text/html");
    }
  }
}
```

When you do this, you can then access any parameters specified on the URL by using the name **parameters** and the name of the parameter you want to access.

To actually specify the parameters you wish to use, use the { } syntax (very similar to ASP.NET MVC) as follows:

Code Listing 11

```
using Nancy;

namespace nancybook.modules
{
  public class BaseRoutes : NancyModule
  {
    public BaseRoutes() : base("/people")
    {
      Get[@"/{id}"] = parameters =>
      {
        var myRecordId = parameters.id;

        return Response.AsFile("Pages/index.html", "text/html");
      };
    }
  }
}
```

You can also force Nancy to restrict only the parameters you supply to certain types. For example:

Code Listing 12

```
using Nancy;

namespace nancybook.modules
{
  public class BaseRoutes : NancyModule
  {
    public BaseRoutes() : base("/people")
    {
      Get[@"/{id:int}"] = parameters =>
      {
        var myRecordId = parameters.id;
```

```
        return Response.AsFile("Pages/index.html", "text/html");
      };
    }
  }
}
```

Notice the addition of **int** on the route parameter; any attempt to request that route with anything other than an integer-based number will result in a **404 not found** error.

Nancy defines many of these constraints, mostly tied to the .NET primitive types such as:

- **long**
- **decimal**
- **guid**
- **bool**
- **datetime**

There are others too, but I'll leave it as an exercise for you to take a look at the NancyFX Wiki (currently located on GitHub); it has a full list of every routing constraint available.

You can provide as many parameters as you wish in the URL—your imagination is the limit (and how complicated you want your URLs to be).

Code Listing 13

```
using Nancy;

namespace nancybook.modules
{
  public class BaseRoutes : NancyModule
  {
    public BaseRoutes() : base("/people")
    {
      Get[@"/person/{age:int}/{surname}/categories/{category}/{city}"] =
parameters =>
      {
        return Response.AsFile("Pages/index.html", "text/html");
      };
    }
  }
}
```

The route for this example might be long and unwieldy, but it's valid and will work without problem. A URL such as **/people/person/21/shaw/categories/developer/durham** would match the route definition without an issue, and would make the values **21**, **shaw**, **developer** and **durham** available in the parameters collection as properties named **age**, **surname**, **category** and **city**.

Summary

In this chapter, you learned how to make Nancy respond to routes set up in your application, and you learned that these routes are no different than any other URL-based routes that frameworks such as ASP.NET MVC might use.

You saw how Nancy defines them, and how you have full control of the route layout rather than the name of the class in which it's defined, and that you have full control over the verb used and the parameters and layout of the URL to match it.

In the next chapter, we'll take a look at view engines and see how to return something more than just plain HTML documents.

Chapter 6 View Engines

Nancy, just like many web toolkits, has a concept of using views to compose the output you see in your browser.

Defining a view

To some of you, the concept of a "view" may be a concept you've never encountered before; to others, you may only know it from using toolkits such as ASP.NET MVC (where the “V” in MVC stands for “View”).

Did you ever stop to consider however just what a view actually is?

According to the Oxford English Dictionary, it's defined as the ability to see something, or to be seen from a particular place, and this fits in well with its meaning when used in a web context such as Nancy, or indeed any frame work that produces views.

A view is something that’s seen in the browser, from the viewpoint of the end user. There is, however, a little more to it than that, because according to our definition, anything that can be viewed as output in a web browser could be classed as a view.

What stands our definition apart is the fact that our view is the result of a “composable” output. By this we mean that our output is not static (such as a pre-made HTML file, or even a PDF document); instead, it’s output that is produced using a mixture of static content and application-generated data.

By this definition, a traditional Windows web application page (web form) that uses a template page to create the final output could be considered a view, which many would disagree with.

For the purposes of understanding Nancy, a view is an output from the framework generated using a mixture of static and program-based data as inputs. For those coming from JavaScript, a view is the output from a templating library such as Handlebars or Mustache.

So why is it necessary to define a view rather than just mention that it's what Nancy produces? As you've seen so far, Nancy is exceptionally modular, and by now it should be no surprise that you can plug different view engines into the framework.

View engines

Put simply, a view engine is a library that produces your view based on the inputs given to it.

Going back to our definition of a view, stop for a moment and think about the main view generator in ASP.NET MVC, Razor.

Razor is the view engine that produces views for that framework; technically, Handlebars is a JavaScript view engine for creating views that use the Handlebars template format to produce output.

Because there are a number of different engines available, you can choose to use whichever suits you best. If Razor (from MVC) is your thing, you can easily use Razor in your Nancy-based project. Some of the others you can use are NHaml, DotLiquid, and Markdown.

You can reliably use all three of these and others; you just need to add them to your project using NuGet. Once they have been added, it's a simple task to start defining the template files that your chosen view engine requires.

The view engines available with Nancy are all conventions-based, and will match their usual file type extension to be mapped through themselves. This means defining your view consists of no more than creating simple text files and giving those files an appropriate extension.

The last thing to know is that Nancy will always look in a folder called Views that is relative to the executing code for any files you ask it to use as view templates.

However, this can be changed, and we will cover that in more depth later on when we look at the Nancy bootstrapper classes.

Creating your first view

If you haven't already, create yourself a simple Nancy application; you can make a copy of my code, use one of the templates mentioned previously, or do the entire process by hand. I've created an empty web application with no authentication, no MVC, and no Web Forms support, and with no initial creation of any files.

For our purposes, we will use the view engine that comes with Nancy out of the box, called SSVE, or "The **S**uper **S**imple **V**iew **E**ngine."

Once you have a skeleton application, add a simple class containing a Nancy route module to it, containing the following code:

Code Listing 14

```
using Nancy;

namespace nancybook.modules
{
  public class BaseRoutes : NancyModule
  {
    public BaseRoutes()
    {
      Get[@"/"] = _ => View["firstView/hellonancy"];

    }
  }
}
```

This route will cause all requests to the **/** path in your application to request a view file called **hellonancy.<*ext*>** from a folder called **firstview** in the default views folder for your application.

Here I've put the file extension as **<ext>** to indicate that the file extension can vary.

In the case of SSVE, .sshtml, .html, and .htm are acceptable file extensions.

Putting together everything we've seen so far, create a folder called **Views** in your project, then inside that create a folder called **FirstView**. Inside that folder, create a regular HTML file called **Hellonancy.html** and put the following HTML code in it:

Code Listing 15

```
<!DOCTYPE html>
<html>

  <head>
    <title>Hello Nancy FX</title>
  </head>

  <body>
    <h1>Hello Nancy FX</h1>
  </body>

</html>
```

If you look in your Solution Explorer, you should have something like the following:

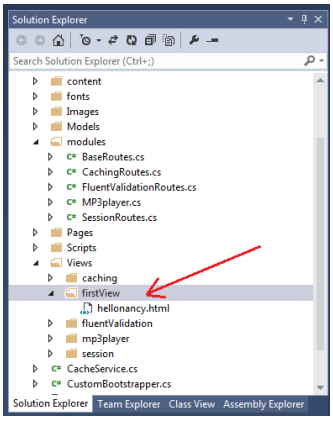


Figure 7: Solution Explorer showing the correct folders and file name for the view file

If you press **F5** at this point and run your project, you should see an HTML page saying “Hello Nancy FX” in your default browser.

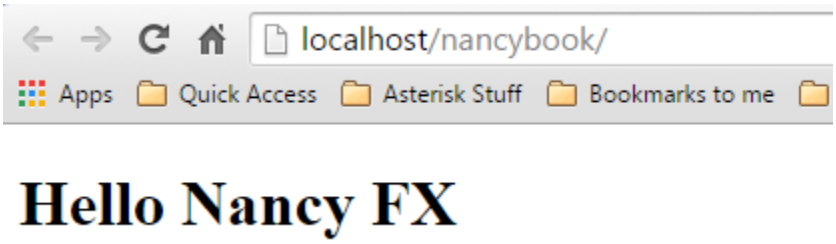


Figure 8: Our first view as seen in the browser

In case you're wondering what the **localhost/nancybook** URL is in Figure 8, that's just to do with the way I set up my development environment; for you, that will likely be different, maybe just showing **localhost**, **localhost:12345**, or something similar. As long as the page contents are the same, everything should be working ok.

A quick mention about static content

At this point, you might be wondering about including static files within your views, things such as images, JavaScript files, and other common web application files. Nancy makes this all very easy by creating a folder called **Content**, and placing your files in there, then using something along the lines of the following:

```
Response.AsFile("index.html", "text/html");
```

Everything that has no folder prefix will be looked for in a folder called Content, and then within any subfolders underneath that.

This convention can be changed in a custom bootstrapper file (which we'll see in a later chapter), and you can easily set up rules so that JavaScript files are in a scripts folder, styles are in a Styles folder, and so on.

For now, however, I am just going to assume that all static resources are within the Content folder. Feel free to organize this folder as you see fit; just remember to change the code samples as required.

Getting back to our discussion on views, you could, in theory, place your resources into separate view files, and just have your views return them. However, this would mean all your static resources would also go through the view engine being used, and that may add a very slight delay to the delivery of files that do not need to be processed.

Because you get the best of both worlds, however, it's not difficult to pass all JS, CSS, and other stuff through the view engine for combination, packing, size reduction, or any manner of other request-based processing. I'll leave you to experiment here.

Getting back to our view

As you've already seen, returning static content through your view engine is not difficult, but we shouldn't stop there. As I previously defined at the beginning of this chapter, using views is about much more than just returning a page; it's about combining known data with that page, and then having the page be presented as an amalgamation of the two inputs.

Let's add a new class to our project to act as a data model for our view. Use the Solution Explorer and add a new class called **`FirstModel.cs`** as follows:

Code Listing 16

```
using System;

namespace nancybook
{
  public class FirstModel
  {
    public string Name { get; set; }
    public DateTime TimeOfRequest { get; set; }
  }
}
```

Taking our previous example route, let's add a reference to the new object, and then return that as a data object for our page:

Code Listing 17

```
using System;
using Nancy;

namespace nancybook.modules
{
  public class BaseRoutes : NancyModule
  {
    public BaseRoutes()
    {
      FirstModel demoModel = new FirstModel
      {
        Name = "Shawty",
        TimeOfRequest = DateTime.Now
      };
      Get[@"/"] = _ => View["firstView/hellonancy", demoModel];
    }
  }
}
```

When run, this will do exactly the same thing as before, as we've not yet changed our view to act on the data. However, you now have the data in **demoModel** passed to your view, and just as with other frameworks, you can use whatever processing rules your chosen view engine uses to display that data.

Change the HTML code in your **hellonancy.html** to read as follows:

Code Listing 18

```
<!DOCTYPE html>
<html>

  <head>
    <title>Hello Nancy FX</title>
  </head>

  <body>
    <h1>Hello @Model.Name</h1>
    <h2>The time is @Model.TimeOfRequest</h2>
  </body>

</html>
```

Make sure you rebuild your project, then re-run it, either by pressing **F5** or refreshing your browser. You should see the following in your browser:

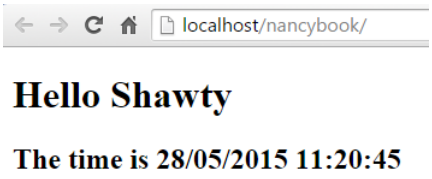


Figure 9: Our browser output showing a view containing our data

SSVE is a very simple engine, but perfectly fine for most use cases—because it is simple; the processing instructions available are not as varied as other engines.

However, this does not stop it from being useful.

For example, you can loop over a collection of items by using the **@Each** operator. When using **@Each** you refer to the currently iterated Model using **@Current**, and then close the loop using **@EndEach**, as the following view shows:

Code Listing 19

```
<!DOCTYPE html>
<html>

  <head>
    <title>Hello Nancy FX</title>
```

```
  </head>

  <body>
    <!-- If we passed our data in as List<FirstModel> we could do the
following -->
    @Each.Model
      <h1>Hello @Current.Name</h1>
    @EndEach

  </body>

</html>
```

As you might expect, this will display the list of names passed in, in your data object.

You can also perform very basic decision-making in your views by using the **@If** processing instruction. **If** comes in two versions: **@If** and **@IfNot**, which will branch if a condition matches or if it doesn't match, depending on which one you use. There is a "gotcha" here though: the **If** condition must be a Boolean (true or false), meaning you cannot perform checks such as **if count < 10** or **if name == 'joe'**. If you plan your objects properly, however, this does not really present a problem.

There's a third way of invoking **If**, using **Has**, which allows you to base your conditionals on a property collection existing in your data object. To demonstrate this, alter your **FirstModel.cs** class so it looks as follows:

Code Listing 20

```
using System;
using System.Collections.Generic;

namespace nancybook
{
  public class FirstModel
  {
    public List<string> Names { get; set; }
    public DateTime TimeOfRequest { get; set; }
    public bool IsError { get; set; }
    public string ErrorMessage { get; set; }

  }

}
```

Now change your route module so it has the following code:

Code Listing 21

```
using System;
using Nancy;
```

```
namespace nancybook.modules
{
  public class BaseRoutes : NancyModule
  {
    public BaseRoutes()
    {
      var demoModel = new FirstModel
      {
        Names = new List<string> { "Shawty", "Frank", "Alice", "Bob",
"Julie", "Samantha"},
        TimeOfRequest = DateTime.Now,
        IsError = false,
        ErrorMessage = "This is an Error message...."
      };

      Get[@"/"] = _ => View["firstView/hellonancy", demoModel];
    }
  }
}
```

Rebuild your project, then make the following changes to the HTML code for your view:

Code Listing 22

```
<!DOCTYPE html>
<html>

  <head>
    <title>Hello Nancy FX</title>
  </head>

  <body>

  @If.Model.HasNames
    <h1>Names</h1>
    @Each.Model.Names
      <h2>@Current</h2>
    @EndEach
  @EndIf

  @If.IsError
    <h2 style="color: Red;">Oh Noes: @Model.ErrorMessage</h2>
  @EndIf

  @IfNot.IsError
    <h2 style="color: Green;">Phew, no errors...</h2>
  @EndIf

  </body>
```

```
</html>
```

When you run this, you should get a list of names, and a message in green stating **"No Errors"**.

Now try changing the **IsError** property to **true**, and setting the **Names** property to **null**. When you rebuild and re-run your project, you should now find you get no list (but importantly no error) and a message in red containing the text you placed in the **ErrorMessage** property. If you set the names collection to an empty list rather than null, you should also find that everything works just the same.

While we're on the subject of handling empty collections, what actually happens if you ask SSVE to display a property in the model that doesn't exist?

You might expect that an exception would be thrown, and you'd get a yellow screen of death (like in other frameworks), but you won't. Instead, you'll get something like this:

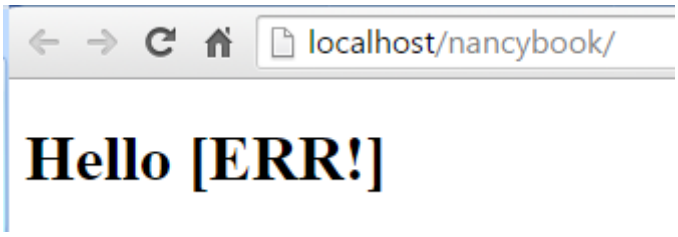


Figure 10: This is what Nancy does when you ask it to display something that doesn't exist

Where you would expect to see the contents of your model property, you'll instead see "[ERR!]". In my mind at least (others will disagree), this is a much better way, because it allows me to still see my layout while developing, without having to constantly make sure I at least have dummy variables in my objects, and it allows QA testing to see missing values immediately.

For a simple view engine, there are a few more things SSVE can do.

By placing an exclamation point (**!**) after the at symbol (@), you can automatically encode the output in HTML:

Code Listing 23

```
var demoModel = new FirstModel
{
  //...
  ErrorMessage = "<strong>This is an Error message....</strong>"
};
```

This will give the following output:

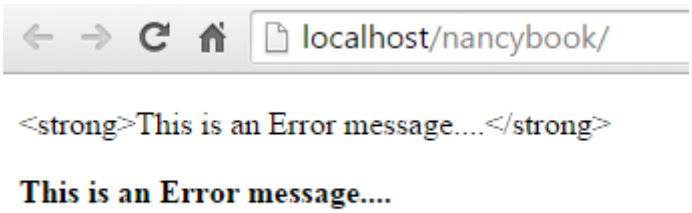


Figure 11: Output using SSVE @! syntax

When used with this HTML:

Code Listing 24

```
<p>@!Model.ErrorMessage</p>
<p>@Model.ErrorMessage</p>
```

SSVE can also do basic partials and master pages with sections by using the **@Partial**, **@Master**, and **@Section** processing instructions.

Add a new HTML file to your **firstview** folder called **master.html**, and then add the following HTML to it:

Code Listing 25

```
<!DOCTYPE html>
<html>

  <head>
    <title>Hello Nancy FX</title>
  </head>

  <body>

    <h1>This header is in the master page</h1>
    <hr />
    @Section['Body']

  </body>

</html>
```

Create a second HTML file in the same folder; we'll call this **partial.html** and add the following to it:

Code Listing 26

```
<div style="width: 200px; height: 200px; border: 1px solid black;">
  <h1>Oh Hai....</h1>
  <p>I'm your new partial content.</p>
```

```
</div>
```

Now change your **hellonancy.html** file so that it looks as follows:

Code Listing 27

```
@Master['firstview/master.html']

@Section['Body']
  <h2>I'm some content that's defined in hellonancy.html</h2>
  <p>and I've brought along a friend to the party:</p>
  @Partial['firstview/Partial.html']
@EndSection
```

If you now re-run or refresh your project in the browser, you should see the following:

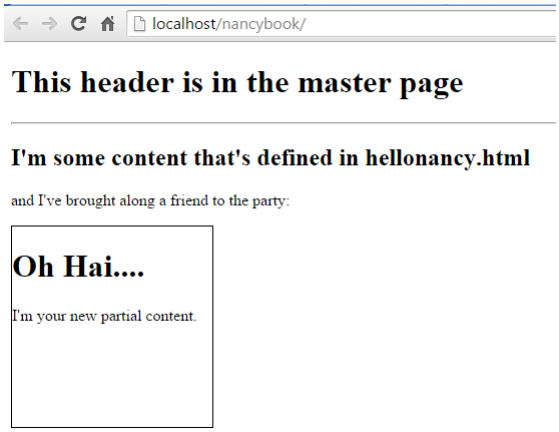


Figure 12: Output produced from the master page/partial example

The final two processing instructions in SSVE's toolbox are the anti-forgery token and the path expansion operator.

Simply put, adding the **@AntiForgeryToken** instruction will generate a hidden input at that location in your output. That is then verified when the page is posted back to a controller. Note, however, that CSRF protection is not enabled by default; you need to enable it in a custom bootstrapper (which you will learn in a later chapter) in order for it to be checked.

> ***Tip: CSRF is a common form of attack used by hackers to gain access to a website. Most toolkits and frameworks (Nancy included) have some form of protection against this type of attack. If you'd like to find out more, do a web search for "Cross Site Request Forgery" or the "OWASP Top 10." The path operator expands any paths you have in your HTML code so that they correctly reference the correct path in your application.***

For example, if you specify:

@Path['~/mypath/myfile.ext']

and your site is rooted at:

`http://myserver/mysite/`

The view will be rendered in the browser with:

`http://myserver/mysite/mypath/myfile.ext`

However, as of version 1.2 of SSVE, any path found that is preceded with a **~** is automatically expanded, so **`@Path`** is now only really needed in a handful of cases, and should be mostly considered deprecated.

Summary

In this chapter, you learned how to combine your data and HTML to produce data-driven output in your applications using the built-in view engine, SSVE. You learned that SSVE, while quite simplistic, includes more than enough under the hood for basic tasks related to producing views.

You learned about other view engines such as Razor and NHaml, as well as learning how to define and use the concept of a view when compared to static content.

In the next chapter, we'll look at bringing data back into your Nancy routes from your HTML pages using model binding and validation, giving you a full two-way, data-driven web application.

Chapter 7 Model Binding and Validation

With any good framework, you need to be able to pass data back to it. Back in the chapter on routing, we saw how easy it is to pass simple values into your routes using the URL parameter routing available in Nancy.

Being able to pass in simple values like this is fine, especially if all you're doing is building an API that looks up and returns database entries. However, in many cases you'll want to provide complex objects of data back to your route.

This might be something as simple as a set of login credentials for the admin section of your admin panel, or it could be a large, flattened record representing several entries in a database table. Either way, you need to be able to pass in more than simple values, and not have a URL that's longer than the average hard drive's deeply nested folder path.

This is where Nancy's model-binding features come into play.

If you've used ASP.NET MVC at all, then you'll already know what model binding is all about: it's the process of looking at the incoming request and any payload it may have, then attempting to match it to the object variables you're looking for.

A picture speaks a thousand words

For simplicity's sake, imagine that we have an address book, and as part of this address book, we want to save an entry into our address database. The first thing we'll likely do is build an object/class that represents what we want an address in our database to look like. It might look something like this:

Code Listing 28

```
namespace nancybook.Models
{
  public class Address
  {
    public int RecordId { get; set; }
    public string Name { get; set; }
    public string BuildingAddress { get; set; }
    public string Town { get; set; }
    public string County { get; set; }
    public string PostalCode { get; set; }
  }
}
```

Looking at this class, you can immediately make an assumption that when populating it, we'll use an integer to identify the record, and that each record will include five pieces of information that make up the address.

When this data is posted from a form in your web page or AJAX action, you'll likely have either a standard form post containing those five pieces of information, or a single JSON object sent from JavaScript to your Nancy route:

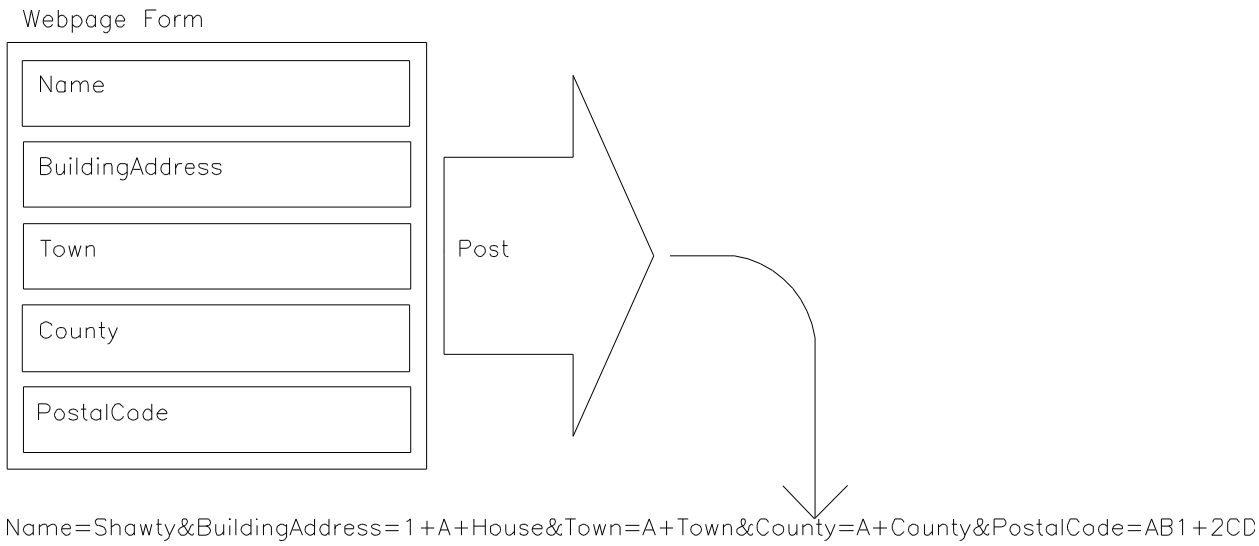


Figure 13: Diagram showing post variables being posted

As you can see from Figure 13, when you post a complex object using a web form, the names of each element in your input form are concatenated with any values entered into those text fields. That large string is then posted to your Nancy endpoint as a single string in the body of the request. This also applies to requests sending JSON and XML, except the data is a single, concatenated JSON- or XML-formatted string.

The model binder's job is to take this string apart and match each of the input fields to the names of the properties in your object.

If only some of the fields are present, then only those fields will be matched, and the others will remain at their default. If no fields can be matched to properties, then the result is an object with all default values in your Nancy route handler when the request is received.

A simple example

Create a folder in your views called **aAddress**, and add an initial HTML page called **index.html** into this new folder, containing the following HTML code:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Nancy Demo | Data Binding Example</title>
```

```
    <link href="~/content/bootstrap.min.css" rel="stylesheet"
type="text/css" />
  </head>

  <body>

    <div class="container">
      <div class="page-header">
        <h1 style="display: inline-block">Nancy Demo <small>Data Binding
Example</small></h1>
        <h1 style="display: inline-block" class="pull-right"><small><a
href="~/" title="Click to return to demo home page">home <span
class="glyphicon glyphicon-home"></span></a></small></h1>
      </div>

      <p class="lead">Please fill in the form below and click 'Send Data'
to perform a data bind to the post action.</p>

        <input id="RecordId" name="RecordId" type="hidden" value="1"/>

        <div class="form-group">
          <label for="Name">Name</label>
          <input type="text" class="form-control" id="Name" name="Name"
placeholder="Enter Persons Name here"/>
        </div>

        <div class="form-group">
          <label for="BuildingAddress">Building Address</label>
          <input type="text" class="form-control" id="BuildingAddress"
name="BuildingAddress" placeholder="Enter Building Address here" />
        </div>

        <div class="form-group">
          <label for="Town">Town</label>
          <input type="text" class="form-control" id="Town" name="Town"
placeholder="Enter Town here" />
        </div>

        <div class="form-group">
          <label for="County">County</label>
          <input type="text" class="form-control" id="County" name="County"
placeholder="Enter County here" />
        </div>

        <div class="form-group">
          <label for="PostalCode">Postal Code</label>
          <input type="text" class="form-control" id="PostalCode"
name="PostalCode" placeholder="Enter Postal Code here" />
        </div>
```

```
        <button type="submit" class="btn btn-primary">Send Data</button>
      </form>

    </div>

    <script src="~/scripts/jquery-2.1.3.min.js"></script>
    <script src="~/scripts/bootstrap.min.js"></script>

  </body>

</html>
```

Add a new class to your modules folder called **`AddressRoutes.cs`** with the following C# code in it:

Code Listing 29

```
using Nancy;

namespace nancybook.modules
{
  public class AddressRoutes : NancyModule
  {
    public AddressRoutes() : base("/address")
    {
      Get[@"/"] = _ => View["address/index"];
    }
  }
}
```

Compile and run your project, then request **/address** in your browser. You should see the following:

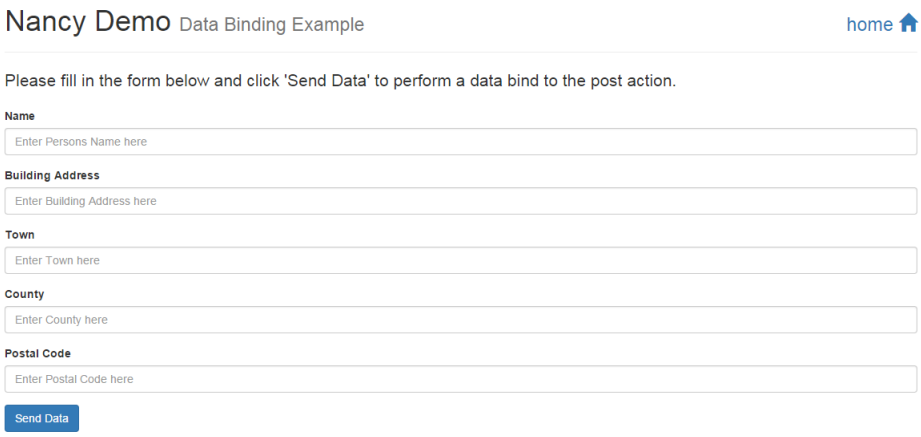


Figure 14: Form generated by the HTML code and route used to return our test address form

Note: I'm using an ASP.NET web application, and I have Bootstrap, jQuery, and Font Awesome installed in the project, so if your form doesn't look exactly the same as mine, don't worry. The important part is that it's a form, and that it's set up to post to a Nancy Route action; how it looks has no bearing on how it works.

Now we have a way of adding our data; it's just a simple matter of binding to that request. There are currently three different ways of binding to your model in Nancy, and which you choose is entirely up to you. You can use type-binding directly:

```
Address myAddress = this.Bind();
```

Or you can use **var**-based binding, passing in the model type as a generic parameter:

```
var myAddress = this.Bind<Address>();
```

Finally, if you have an existing instance of a TModel already allocated, you can use **BindTo** to attach to it:

```
var myAddress = this.BindTo(existingModelInstance);
```

This last form is generally used only in special circumstances, such as in extensions (this is the only circumstance in which I've seen it used), with the first two forms being the more popular.

Personally, I tend to use the second form, **var**-based binding.

Whichever form you use, the result should be the same: an object containing any properties and fields bound from the incoming data, or an object containing default values if nothing could be bound. There is a small chance that the object you're binding to could also come out as null, so it's often a good idea to try and handle that by checking the model for null.

Extend your address route's module, so that it now looks like this:

Code Listing 30

```
using nancybook.Models;
using Nancy;
using Nancy.ModelBinding;

namespace nancybook.modules
{
  public class AddressRoutes : NancyModule
  {
    public AddressRoutes() : base("/address")
    {
      Get[@"/"] = _ => View["address/index"];

      Post[@"/save"] = _ =>
      {
```

```
        var myAddress = this.Bind<Address>();
        if (myAddress != null)
        {
          return View["address/display", myAddress];
        }

        return View["address/error"];
      };

    }
  }

}
```

Save this route module, then add two more files to your **address** folder in views as follows:

display.html:

Code Listing 31

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Nancy Demo | Data Binding Example</title>
    <link href="~/content/bootstrap.min.css" rel="stylesheet"
type="text/css" />
  </head>

  <body>

    <div class="container">
      <div class="page-header">
        <h1 style="display: inline-block">Nancy Demo <small>Data Binding
Example</small></h1>
        <h1 style="display: inline-block" class="pull-right"><small><a
href="~/" title="Click to return to demo home page">home <span
class="glyphicon glyphicon-home"></span></a></small></h1>
      </div>

      <p class="lead">The results from your address form are as
follows...</p>

      <p>Record ID : <strong class="text-
success">@Model.RecordId</strong></p>
      <p>Name : <strong class="text-success">@Model.Name</strong></p>
      <p>Address : <strong class="text-
success">@Model.BuildingAddress</strong></p>
```

```
      <p>Town : <strong class="text-success">@Model.Town</strong></p>
      <p>County : <strong class="text-success">@Model.County</strong></p>
      <p>Post Code : <strong class="text-
success">@Model.PostalCode</strong></p>

      <a href="~/address" class="btn btn-primary">Go back to the input
form</a>

    </div>

    <script src="~/scripts/jquery-2.1.3.min.js"></script>
    <script src="~/scripts/bootstrap.min.js"></script>

  </body>

</html>
```

error.html:

Code Listing 32

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Nancy Demo | Data Binding Example</title>
    <link href="~/content/bootstrap.min.css" rel="stylesheet"
type="text/css" />
  </head>

  <body>

    <div class="container">
      <div class="page-header">
        <h1 style="display: inline-block">Nancy Demo <small>Data Binding
Example</small>
        </h1>
        <h1 style="display: inline-block" class="pull-right"><small><a
href="~/" title="Click to return to demo home page">home <span
class="glyphicon glyphicon-home"></span></a></small>
        </h1>
      </div>

      <h3 class="text-danger">Unfortunately something went wrong trying to
bind to the inbound data, and Nancy was unable to bind to your object.</h3>

      <a href="~/address" class="btn btn-primary">Go back to the input
form</a>
```

```
    </div>

  </body>
</html>
```

Recompile your application, then request **/address** in your browser. Fill in the form and click **Send Data**. If everything worked, you should see something like the following:

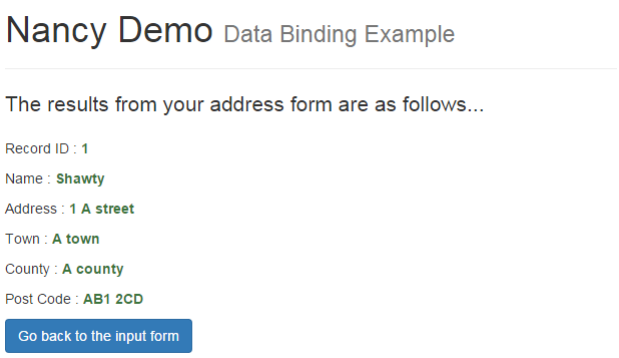


Figure 15: The result of posting an address entry to the form binder

In your input form, you could change all the name attributes on each of the input elements to something that's different to the property names in your address model EG:

```
<input id="RecordId" name="RecordIdDONOTUSE" type="hidden" value="1" />

<div class="form-group">
  <label for="Name">Name</label>
  <input type="text" class="form-control" id="Name" name="NameDONOTUSE"
placeholder="Enter Persons Name here" />
</div>
```

Then, press **F5** to reload your form, and try entering data again. You should observe that the object just contains default values (in this case, **`0`** for **`RecordId`** and an empty string for **`Name`**).

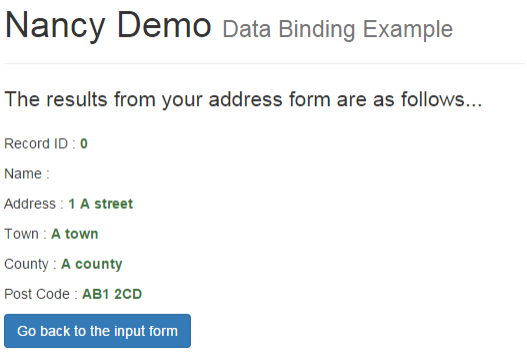


Figure 16: The result of sending data to our endpoint with missing data

As you can see, if any fields or properties are missing from your inbound data, Nancy doesn't throw an exception or cause any kind of show-stopping error, as some frameworks do.

There may, however, be times when you specifically need to force a property or field to be ignored. For example, you might want to make sure that a form has a field for a second password (so you can validate a password entry on the client), but you might not want that property bound to your object. Or, you might want to use the same input form for a user and an admin input, but have certain fields ignored when the form is used by a user, and not ignored when it's used by an admin.

Whatever the reason, it's easy to ask the binding system to ignore any properties you want it to, by providing either a list of lambdas or just simple property names when performing the binding.

Make sure that your form is changed back to the original version (so that all the fields are submitted to the post endpoint), then change your Nancy module so that it looks like the following:

```
using nancybook.Models;
using Nancy;
using Nancy.ModelBinding;

namespace nancybook.modules
{
  public class AddressRoutes : NancyModule
  {
    public AddressRoutes() : base("/address")
    {
      Get[@"/"] = _ => View["address/index"];

      Post[@"/save"] = _ =>
      {
        var myAddress = this.Bind<Address>("BuildingAddress");
        if (myAddress != null)
        {
          return View["address/display", myAddress];
        }

        return View["address/error"];
      };

    }
  }
}
```

I've highlighted the line that's changed in red; you should be able to see that I've added a parameter to it that matches the name of a property in my object.

If you build and run your application now, then submit the form, you'll find that even though the **`BuildingAddress`** property is being supplied, the binding system openly ignores it.

You can also specify the field names as follows:

```
var myAddress = this.Bind<Address>(f => f.BuildingAddress);
```

You can specify multiple property lists in both cases by separating each name entry with a comma.

Binding to lists and arrays

At this point you might be wondering, “What about adding checkboxes to my form?” The good news is that you don't need to do anything special, except for ensuring that your checkbox has a name, just like any other form parameter (just an ID won't do). When you post it, any **bool** property in your object that matches the checkbox name will be populated with **true** or **false** as needed.

Binding to arrays is just as simple. If you have a regular text field that contains a text representation of your list items separated by commas, the Nancy model-binder will split this input into an array automatically, should one be provided in the model you’re binding to. For example, if you had the following input field in your form:

```
<div class="form-group">
  <label for="AliasList">Alias List</label>
  <input type="text" class="form-control" id="AliasList" name="AliasList"
placeholder="Enter List of aliases here" />
</div>
```

Then you updated your address model so it looked like the following:

```
namespace nancybook.Models
{
  public class Address
  {
    public int RecordId { get; set; }
    public string Name { get; set; }
    public string BuildingAddress { get; set; }
    public string Town { get; set; }
    public string County { get; set; }
    public string PostalCode { get; set; }
    public string[] AliasList { get; set; }
  }
}
```

You would then find that if you entered something similar to **“**Peter,Shawty,Pete,Mr Shaw” in that field and submitted it, your model property would then include each entry in a separate array slot:

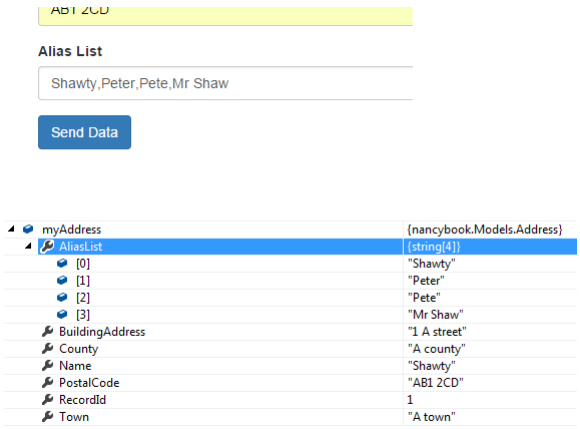


Figure 17: Entering a comma separated list in the binder will bind as an array

This list binding works with any simple value types such integers, decimals, and Booleans.

The automatic parsing of multiple items into an array works with multiple inputs, too. For example, say you had a list of check boxes, formatted as follows:

Code Listing 33

```
Option 1 <input type="checkbox" name="OptionsList" value="One" /><br />
Option 2 <input type="checkbox" name="OptionsList" value="Two" /><br />
Option 3 <input type="checkbox" name="OptionsList" value="Three" /><br />
Option 4 <input type="checkbox" name="OptionsList" value="Four" /><br />
Option 5 <input type="checkbox" name="OptionsList" value="Five" /><br />
```

If you render and submit the form containing them, the model binder will add an entry to the array for each item that's checked, just as it did for any items separated by a comma.

Lastly, if you add a multiple-item select list in your form using the following:

Code Listing 34

```
<select multiple="multiple" name="AliasList">
  <option>Option 1</option>
  <option>Option 2</option>
  <option>Option 3</option>
  <option>Option 4</option>
  <option>Option 5</option>
</select>
```

That, too, will bind and populate the array with the selected options from your list.

It's worth noting that we've used a standard **`array[]`** so far in the binding examples, but anything that derives from an **`IEnumerable`** will bind in exactly the same manner. For example:

Code Listing 35

```
namespace nancybook.Models
{
```

```
  public class Address
  {
    public int RecordId { get; set; }
    public string Name { get; set; }
    public string BuildingAddress { get; set; }
    public string Town { get; set; }
    public string County { get; set; }
    public string PostalCode { get; set; }
    public List<string> AliasList { get; set; }
  }
}
```

The outcome will be exactly the same as with the array syntax, except that now you can use all that lovely LINQ goodness to manipulate your data as you see fit.

The last trick that Nancy has when it comes to model binding is the ability to bind to entire lists of data. So far, we've only uses primitive lists, embedded in our objects, alongside other singular items of data. But what if we wanted to pass a list of addresses to our address module? It's probably no surprise by now that Nancy has a "super-duper-happy-path" for that, too.

If you format your input items in your form using a hidden index, and array specified fields for each of your properties, you can then bind the entire inbound data model to an **`IEnumerable`** list. If you have a form that looks like the following:

Code Listing 36

```
<form method="POST" action="~/address/save">

  <!-- Address item 0 -->
  <input id="RecordId" name="RecordId[0]" type="hidden" value="1"/>

  <div class="form-group">
    <label for="Name">Name</label>
    <input type="text" class="form-control" id="Name" name="Name[0]"
placeholder="Enter Persons Name here" />
  </div>

  <div class="form-group">
    <label for="BuildingAddress">Building Address</label>
    <input type="text" class="form-control" id="BuildingAddress"
name="BuildingAddress[0]" placeholder="Enter Building Address here" />
  </div>

  <div class="form-group">
    <label for="Town">Town</label>
    <input type="text" class="form-control" id="Town" name="Town[0]"
placeholder="Enter Town here" />
  </div>

  <div class="form-group">
    <label for="County">County</label>
```

```
    <input type="text" class="form-control" id="County" name="County[0]"
placeholder="Enter County here" />
  </div>

  <div class="form-group">
    <label for="PostalCode">Postal Code</label>
    <input type="text" class="form-control" id="PostalCode"
name="PostalCode[0]" placeholder="Enter Postal Code here" />
  </div>

  <!-- Address item 1 -->
  <input id="RecordId1" name="RecordId[1]" type="hidden" value="2" />

  <div class="form-group">
    <label for="Name">Name</label>
    <input type="text" class="form-control" id="Name1" name="Name[1]"
placeholder="Enter Persons Name here" />
  </div>

  <div class="form-group">
    <label for="BuildingAddress">Building Address</label>
    <input type="text" class="form-control" id="BuildingAddress1"
name="BuildingAddress[1]" placeholder="Enter Building Address here" />
  </div>

  <div class="form-group">
    <label for="Town">Town</label>
    <input type="text" class="form-control" id="Town1" name="Town[1]"
placeholder="Enter Town here" />
  </div>

  <div class="form-group">
    <label for="County">County</label>
    <input type="text" class="form-control" id="County1" name="County[1]"
placeholder="Enter County here" />
  </div>

  <div class="form-group">
    <label for="PostalCode">Postal Code</label>
    <input type="text" class="form-control" id="PostalCode1"
name="PostalCode[1]" placeholder="Enter Postal Code here" />
  </div>

  <button type="submit" class="btn btn-primary">Send Data</button>
</form>
```

If you alter your **`Address`** module so that it binds your address object as a list, as follows (I've highlighted the changed line):

Code Listing 37

```
using System.Collections.Generic;
using nancybook.Models;
using Nancy;
using Nancy.ModelBinding;

namespace nancybook.modules
{
  public class AddressRoutes : NancyModule
  {
    public AddressRoutes() : base("/address")
    {
      Get[@"/"] = _ => View["address/index"];

      Post[@"/save"] = _ =>
      {
        var myAddress = this.Bind<List<Address>>();
        if (myAddress != null)
        {
          return View["address/display", myAddress];
        }

        return View["address/error"];
      };

    }
  }
}
```

You'll find that when you examine the data in the Visual Studio debugger, you get a full list of separate object types:

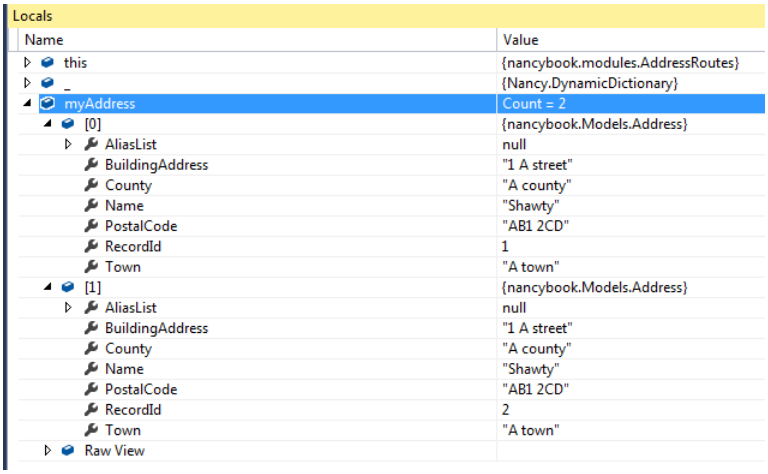


Figure 18: Visual Studio debugger showing list binding

As long as the form contents are laid out as shown, and each input element name has an array index appended to it, then the Nancy model binder will allow you to bind it.

A word of caution: you must make sure that your indexes are sequential. I've read that the method Phil Haack suggests for ASP.NET MVC, using a hidden index field, also works for Nancy. I've not actually tested this, however, and when I generate forms dynamically to take advantage of this feature, I always try to make sure I keep the indexes in sync.

Validation

The last thing we'll take a look at in this chapter is validation. Just like ASP.NET MVC and other .NET-based frameworks, Nancy has the ability to validate your objects when you bind to them, allowing you to easily reject any binds that contain incomplete or incorrect data.

There are two packages currently available on NuGet that provide validation support: **Nancy.Validation.FluentValidation** and **Nancy.Validation.DataAnnotations**.

For the purposes of this book, I'm only going to demonstrate the data annotations approach. If you're used to using fluent validations, or already have that deployed elsewhere, you may want to use it. The concept is the same in that you need to create a separate class that derives from **`AbstractValidator`**. There's an example of how to do this in the NancyFX demos in the source code on GitHub.

Validating with data annotations

If you've used ASP.NET MVC, you may be familiar with the way models are validated using the framework. Data annotations for NancyFX work in exactly the same way.

You attach attributes to your data objects, and these attributes are used to generate rules, which the rest of the framework checks. Because the same data annotation assemblies used in the ASP.NET product are used in Nancy, any rules you may have already developed to work under ASP.NET MVC will also work without change under a NancyFX-based project.

To give you an example of how to use them, take your **`Address`** class from the previous demonstration and change it so that it looks like the following code:

```
using System.ComponentModel.DataAnnotations;

namespace nancybook.Models
{
  public class Address
  {
    [Required]
    public int RecordId { get; set; }

    [Required]
    [StringLength(20)]
    public string Name { get; set; }
```

```
    [RegularExpression(@"^[0-9]{1,3}\s.*")]
    public string BuildingAddress { get; set; }

    [Range(18, 150)]
    public int Age { get; set; }

    [DataType(DataType.EmailAddress)]
    public string Email { get; set; }
  }
}
```

You can see I've added a few of the available validations in here. **Required** ensures that the property has a value and will reject the validation if not; and **StringLength** ensures the string has no more that the specified number of characters (20 in this case). The **RegularExpression** attribute allows you to specify a custom pattern to match, too. I'm not going to delve into the subject of regular expressions here, but if you're interested, there is a book in the *Succinctly* series that covers this topic.

The last two, **Range** and **DataType**, allow you to specify a numerical or date range into which a value must fall, and the type of data expected as the format of the contents.

The latter on **DataType** may be a little confusing to some, but the **DataAnnotations** assembly has a number of pre-made patterns that can be used to check the format of the supplied string. In this demo, I've chosen to use the **Email** format, which will check that the data is formatted correctly for an email address. If you use IntelliSense in Visual Studio, you'll see there are many others:

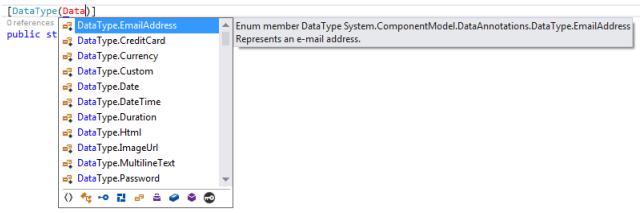


Figure 19: Visual Studio IntelliSense showing the different data types available

Once you've added the required attributes to your model, validating it is easy.

```
using nancybook.Models;
using Nancy;
using Nancy.ModelBinding;
using Nancy.Validation;

namespace nancybook.modules
{
  public class AddressRoutes : NancyModule
  {
    public AddressRoutes() : base("/address")
    {
```

```
      Get[@"/"] = _ => View["address/index"];

      Post[@"/save"] = _ =>
      {
        var myAddress = this.Bind<Address>();
        var result = this.Validate(myAddress);

        return result.IsValid
          ? View["address/display", myAddress]
          : View["address/error"];
      };

    }
  }
}
```

You can see that you bind the object just as you did before, but now, you immediately make a call to **`this.Validate`**.

The validation returns a result object that, among other things, has a property called **`IsValid`**. This property is a regular true or false Boolean value that you can test to see if your object passed validation.

If **`IsValid`** is set to **`false`**, then the **`Errors`** property will contain a list of errors and error messages telling you exactly what failed in the validation.

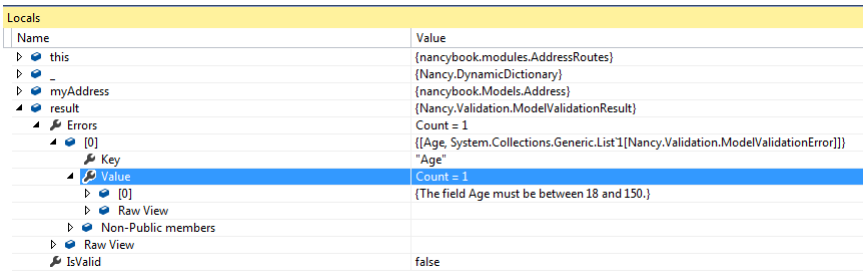


Figure 20: Visual Studio debugger showing validation errors

How you display those errors is entirely up to you; Nancy has no specific way that you must deal with them, leaving you free to do what is best for your application.

Summary

In this chapter, we took a closer look at Nancy's model binder. We saw firsthand how to get our view models from our HTML pages back into our Nancy module.

We also saw how Nancy smartly takes multiple inputs, or single inputs with a delimiter, and makes them easily accessible to us in the form of standard lists and arrays.

Finally, we looked at how Nancy provides the same experience as other .NET frameworks and allows you to easily provide validation for your view-bound data objects.

In the next chapter, we will move away from plain HTML and explore content negotiation, Nancy's “super” weapon for powering API-based URLs.

Chapter 8 Content Negotiation

In the world of web frameworks, everything is REST-based these days—from the routes that return rich, nicely formatted HTML views full of CSS and JavaScript, right through to those boring URLs that just return a bunch of JSON.

Figure 21: A rather plain JSON feed as viewed directly in the browser

Whichever way you look at it, both are required. We use one set of URLs to deliver our user interface, and then we use a second set of URLs to deliver the dynamic data to that user interface. Sometimes we might even have a third set of URLs that deliver data to our mobile applications.

All this duplication can seem like a lot of work to maintain, and it is. I've been there, and it's certainly not pretty. To provide a solution to this problem, many frameworks (Nancy included) have something called *content negotiation* added to them.

Content what?

When you make a request using a browser (or some code that knows how to speak to a web server), you provide various bits of information behind the scenes to describe the request.

These pieces of information are known as **request headers**:

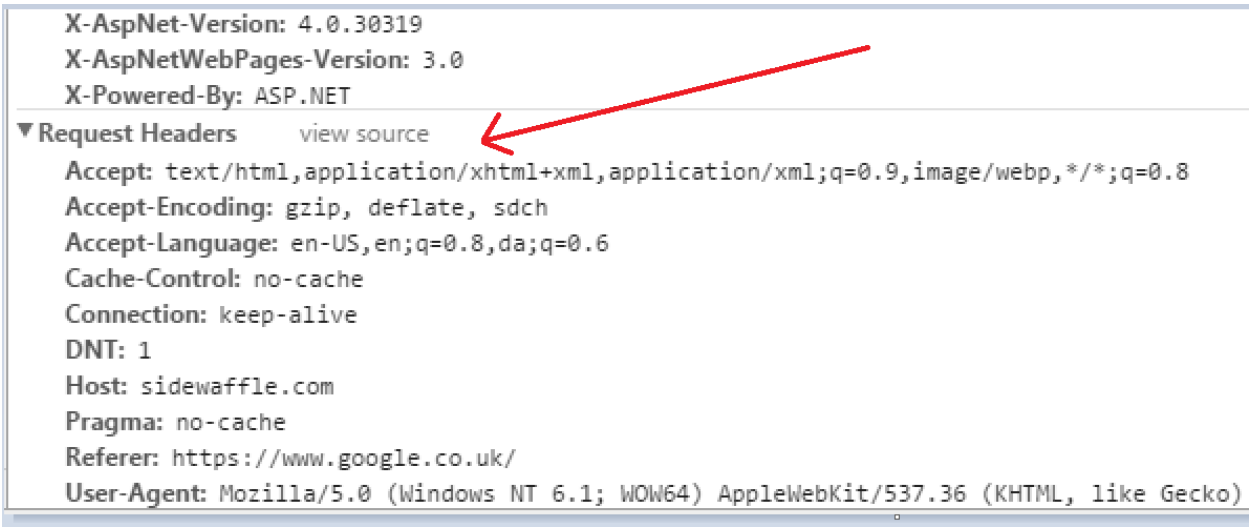


Figure 22: Chrome debugger showing the request headers for the JSON in the previous image

You might not be able to see it in the screenshot, but one of these headers is called the *Accept* header, which tells the web server what kind of response the browser or calling program is willing to accept, in our example it's set to:

`Accept: text/html,application/xhtml+xml,application/xml,image/webp,*/*`

Which says that, in order of preference, I would like you to send me **`html`**, **`xml`**, or a **`webp`** image, and if you can't provide any of those in that order, then just send me what you have.

Nancy pays attention to this header when it receives a request. Based on what it finds, it can change the output of a given URL.

If our **`Accept`** header read:

`Accept: application/json`

Nancy will immediately look to see if it can send its response using JSON-based data, and if it can't, it will throw an exception and fail. If we don't want it to fail, then we could add **`*/*`** onto the end, just as in the previous version, and that would cause Nancy to just send the default response that it's able to generate.

By default, Nancy can make a decision between HTML, XML, and JSON without you having to provide any extra configuration. If you build an API and then use that API to return a simple model, as we have already seen in previous chapters, Nancy will try to return the most appropriate response.

If you call your URL from a browser, then Nancy will look in your views folder for a view with a name that matches the object type.

Taking the **`Address`** object we were using in the previous chapter, Nancy would look for:

'~/Views/Address.Html'

That small HTML snippet could then use any of the available view engines to generate the output without you ever actually having to explicitly tell your module to return a view.

If the URL were called from some JavaScript inside your pages, and the **Accept** header in the call to the same URL-specified JSON, Nancy would just serialize your object into JSON syntax and return that.

To allow Nancy free reign in deciding which content type to return, you return nothing more than a simple object model from your Nancy routing modules. The methods we've seen so far for returning strings, pages, and view content are no use here; by using any of those, we stop Nancy from performing a correct analysis of the accept header. Instead, the quickest way is as follows:

Code Listing 38

```
Get[@"/{id}"] = parameters =>
{
  int recId = parameters.id;
  var record = Database.Find(recId);
  return record;
};
```

By either returning a model using **new** or grabbing an object back from some backend store, you are letting Nancy act on the header information and decide exactly how to format the response.

You can also use the fluent **Negotiate** response API to perform the same task. The difference here is that you have much more control over the response, such as changing the view name that the HTML response will look for, or adding extra headers. You use the fluent response API in the following fashion:

Code Listing 39

```
Get[@"/{id}"] = parameters =>
{
  int recId = parameters.id;
  var record = Database.Find(recId);
  return Negotiate
    .WithModel(record)
    .WithHeader("X-CustomHeader", "Some custom value");
};
```

As before, this finds a record using our pretend **Database**, but instead of returning that model directly, it returns it while adding a custom heading to the response. There's a whole list of additions that can be made onto the response; Visual Studio's IntelliSense will show you them all, if you just pause after entering the first period:

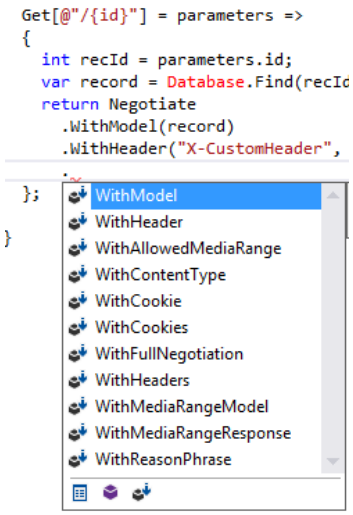


Figure 23: Visual Studio showing other methods in the fluent negotiation API

Finally, you can extend the serialization base classes and add in different or custom content types. If you take a look at some of the links on the NancyFX Wiki page, you'll find there are additions to return PD's and other special types of content, based solely on what the calling client asks for.

If you look in NuGet using **Nancy.Serialization**, you'll quickly find that there are content serializers available for Google Protocol Buffers, JSON .NET, ServiceStack, and a few others.

Unless you're using a custom serializer or doing things in a non-standard way, taking advantage of Nancy's ability to serve the correct content is automatic, as long as you make sure the correct headers are in your request.

Summary

In this chapter, you got a brief introduction to content negotiation in Nancy—it's such a simple concept that it requires next-to-no configuration to make it work correctly.

You saw that by employing content negotiation correctly, you can reduce the surface area of your API to a single URL that serves requests for everything using it, and in the process, reduce the complexity and maintenance time required to manage that API.

In the next chapter, we'll take a closer look at Nancy's response object, and see the different ways we return data and pages to a Nancy client.

Chapter 9 Responses

Following on closely from content negotiation is the Nancy **`Response`** object.

You've already seen the **`Response`** object in action in the first couple of chapters in this book, when we used it to return a simple file using **`AsFile`**:

Code Listing 40

```
using Nancy;

namespace nancybook.modules
{
  public class BaseRoutes : NancyModule
  {
    public BaseRoutes()
    {
      Get[@"/"] = _ => Response.AsFile("index.html", "text/html");
    }
  }

}
```

`AsFile` can be used to return any file you have access to in the server's file system and set its specific mime type, forcing the client to download or open the content in a specific way.

In recent versions of Chrome, for example, a PDF reader is built-in, so it's very easy to define a route module that can send PDF documents directly to the browser:

Code Listing 41

```
using Nancy;

namespace nancybook.modules
{
  public class BaseRoutes : NancyModule
  {
    public BaseRoutes()
    {
      Get[@"/"] = _ => Response.AsFile("myreport.pdf", "application/pdf");
    }
  }
}
```

As you can see, this results in the PDF displaying in the browser:

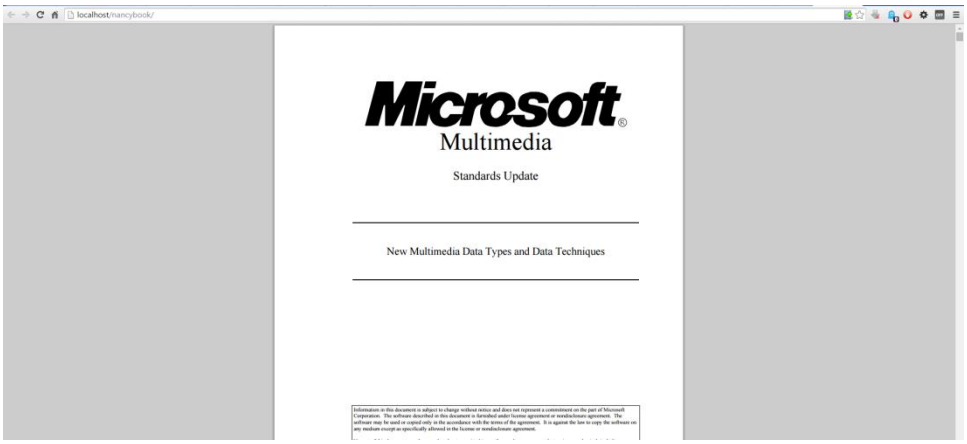


Figure 24: Our PDF report being displayed in the Chrome browser

In general, when returning files this way, you're ensuring that the end user's browser choices are being taken into consideration. Some users might have their PCs set to open a PDF in a standalone viewer, or they might have it set to download the file to disk; by returning the file in this manner and setting the mime type correctly, you're telling the browser exactly what you're sending and letting it decide how to handle it.

Sending files with the correct mime type is not the only thing the response object can do. If you type **`Response.`** and then pause, you'll see that IntelliSense will show the following:

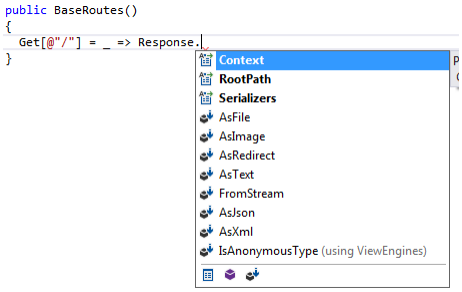


Figure 25: Response object methods as seen in IntelliSense

`AsImage` will take the file path passed to it and force the response to be an appropriate image type; this is basically a shortcut for using **`AsFile`** with an appropriate image mime type.

`AsRedirect` will redirect to another page or site; the first parameter will be either a relative or absolute URL to which the client will be sent. This method also has an optional second value, which can be one of the following enumerations:

- **`RedirectResponse.RedirectType.Permanent`** (generates response code 301)
- **`RedirectResponse.RedirectType.SeeOther`** (generates response code 303)
- **`RedirectResponse.RedirectType.Temporary`** (generates response code 307)

The default is to send a **`303`** response, which is essentially a temporary redirect.

AsText, **AsJson**, and **AsXml** are all shortcuts to allow the returning of pre-rendered XML. TEXT and JSON data stored on disk. This is slightly different from content negotiation as discussed in the previous chapter, as the data is not generated on the fly, but instead is grabbed from an existing file and sent with the appropriate mime type headers.

The final method, **FromStream**, allows you to stream a file to the browser. This is typically used when you want to send a very long file (such as an audio or video file), but want it to start being available immediately. For example, imagine you had the following audio element in an HTML page that you'd delivered to the browser:

Code Listing 42

```
<audio autoplay="autoplay" controls="controls">
  <source src="~/mp3player/playtrack" />
</audio>
```

You could very easily use the following Nancy module to play the track in a streaming fashion:

Code Listing 43

```
using System.IO;
using System.Linq;
using Nancy;

namespace nancybook.modules
{
  public class MP3Player : NancyModule
  {
      Get[@"/playtrack/{id}"] = parameters =>
      {
        Stream fileStream = File.OpenRead("mymusic.mp3");
        return Response.FromStream(fileStream, "audio/mp3");
      };
    }
  }
}
```

You can serve the file from a database stream, a memory stream, or a stream from another server. It doesn't matter where the stream comes from; Nancy will open it and begin sending it more or less in real time to the client requesting it.

You can also create a new response object, and populate the properties in it yourself. For example, let's imagine that you had your images in a database, stored as BLOB objects. You could easily send image responses from your Nancy modules using something like this:

Code Listing 44

```
using System.IO;
using Nancy;

namespace nancybook.modules
{
```

```
  public class BaseRoutes : NancyModule
  {
    public BaseRoutes()
    {
      Get[@"/"] = _ =>
      {
        Response myResponse = new Response();

        byte[] contentBuffer = new byte[1000];
        // do something here to load your image into the buffer

        Stream contentStream = new MemoryStream(contentBuffer);

        myResponse.ContentType = "image/jpg";
        myResponse.Contents = stream => contentStream.CopyTo(stream);

        return myResponse;
      };
    }
  }
}
```

You can also use the response object directly to control the HTTP response code you want to send back to the client; Nancy has an enumeration built in that provides an easy way to select the one you want:

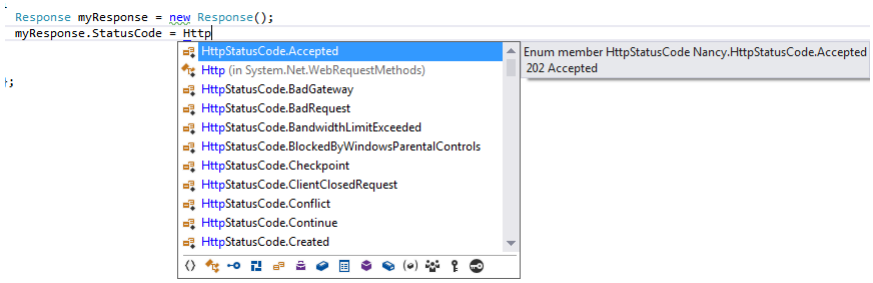


Figure 26: Response HTTP codes shown by IntelliSense

For example, you could do something like the following:

Code Listing 45

```
Get[@"/"] = _ =>
{
  Response myResponse = new Response();
  myResponse.StatusCode = HttpStatusCode.Forbidden;
  myResponse.ReasonPhrase = "Server says NO!!";

  return myResponse;
};
```

This allows you to send your own custom **`403 (Access forbidden)`** result code, which appears in Chrome with the following:

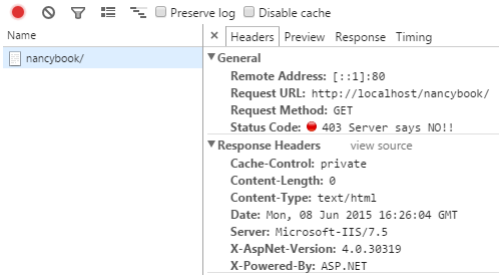


Figure 27: Our custom 403 response as seen by Chrome

Given how smart Nancy is, however, if all you want to do is return a specific status code and not set the message, then you can make things even simpler:

Code Listing 46

```
public BaseRoutes()
{
  Get[@"/"] = _ => 403;
}
```

This will return the same **`403 (Access forbidden)`** code that the previous example did.

Nancy will attempt to return whatever you ask it to in the most appropriate way, so if it can convert your response to an integer, it will make an assumption that the integer given is intended to be a response code.

If you respond with a string, Nancy will return that as a regular OK (Response code **`200`**) response, returning plain text to the client.

All-in-all, Nancy's response mechanism is very smart, and like everything else in Nancy, is easily extendable too. All you need to do is derive a new class from the existing **`Response`** base class, and then you can return that from your modules as easily as anything else.

For an interesting example of where this comes in handy, see this post on Simon Cropp's blog**,** which discusses returning files from outside your server's file tree.

Summary

In this chapter, you got to see how flexible Nancy's response mechanism really is. You learned that Nancy can return almost anything with relative ease from any of its routing modules. You also learned that if you can't return something easily, it's reasonably straightforward to extend the response mechanism.

In the next chapter, we'll take a look at using Nancy's pre-made authentication features to allow you to protect your Nancy URLs and endpoints.

Chapter 10 Authentication

If you're using a web framework to build web services to be used by the general public, then it's likely you'll need some way of controlling who has access to which facilities in said service.

It may be as simple as ensuring that API users provide a key in the headers before they can access your API, or it may that you're building an entire platform that has to support multiple users of different access levels.

Yet again, Nancy has all the bases covered here, and implementing an authentication scheme is very easy. As with many of the other features, everything is modular and installable using NuGet, and in the case of the authentication stuff, nothing is installed by default.

Authentication methods available

Before I give you an example, I'm going to go over the three methods that are available to install from NuGet. You can find them all by searching for "Nancy.Authentication" and installing the package you want.

While it's perfectly fine to install more than one method, there isn't much documentation on running two of them side-by-side. My recommendation is to stick with one method at a time.

Given what I've read however, it should be possible to implement two or more methods at the same time if you really did need to do so.

So what's available? If you look on NuGet, you'll see the following methods listed: Stateless, Forms, and Token. Each of these methods has its strengths and its weaknesses, but more importantly, each is designed to be used in a specific scenario, as you'll see next.

Forms authentication

We'll start with the easiest one first. If you're used to anything in ASP.NET, it should be forms authentication.

Used since the days of the original WebForms implementation, Forms authentication has been (and still continues to be) the underpinning of the authentication on the vast majority of .NET-based web applications built in the last 10 years.

It comes as no surprise, then, that Nancy implements this in exactly the same way as the other implementations you're used to. Later in this chapter I'll run through an example of how to use it, but for now, if you're used to using and configuring this for web sites that are designed to be used by human beings, then this should be your first choice in your Nancy application.

Its primary motivation is interactive web sites and web applications, interacted with by a human operator through a standard web browser-style interface.

Stateless authentication

Stateless authentication is primarily designed for use by API-based applications or applications that need to check and authenticate on each request.

as an example of the intended use of this method, the NancyFX documentation gives an API endpoint with access key provided in the request headers. You could, however, use it for many other service methods.

Its use is very simple; unlike the other methods, there's no configuration other than a simple handler that needs to be performed to get things working.

The handler that you define to use the method will be called each time an endpoint secured using the method is called. The handler should then either return Null if the request should not be allowed, and a Nancy **`IUserIdentity`** if the request is considered valid and should be allowed (we'll learn more about this in a moment).

Token authentication

The Token method is probably the most complex of the methods to understand.

The documentation states that its primary use is for multiple consuming clients such as mobile apps, single-page apps, and others that should allow authentication for future requests, but not have to re-query the backend data store once a user has successfully authenticated the first time.

There is a better description of its intended usage on the NancyFX Wiki, so I won't go into the description in much depth here.

The method works by keeping a dual key store inside the Nancy app, which it checks against for each protected resource.

The initial user authentication is performed by a custom method provided by the application writer. Once this is performed the first time for a given user, subsequent requests then just use the tokens generated by the method, which are valid for any given 24-hour period before timing out.

Using Nancy's authentication services

Irrespective of which method of authentication you choose, or even if you use your own custom provider, there are some concepts that are common to all of them.

The first concept is that of a user. Nancy understands what a user is and how the idea of having a user included as part of the request works.

At any point in your Nancy application where you have access to the **`NancyContext`**, you will also have the ability to read a property called **`CurrentUser`**.

If the **CurrentUser** property is null, then the request is not considered to be an authenticated one. If the property has an object attached to it that derives from the **IUserIdentity** interface implemented in the Nancy core, then the request is considered authenticated.

At minimum, the **CurrentUser** property should implement a string containing the users name and an enumerable list of strings containing a list of user claims.

If these are set, you can then begin to secure your application by attaching a **Before** pipeline filter to your request; in this filter you need only check the current user and act accordingly. You will find out more about these in the next couple of chapters.

Nancy makes adding this filter much simpler by providing some premade extension methods specifically for handling your applications authentication needs.

The simplest of these is **RequiresAuthentication**, which will kick back an HTTP response of **403 (Not Authorized)** if the user property is null.

You can also have **RequiresClaims**, which allows you to match a list of claims against the list of claims provided by the user identity, allowing you to provide different levels of access depending on the user's abilities. Be aware that if you use anything other than **RequiresAuthentication**, most of it will call this first, so even though you can specify multiple, you often will not explicitly have to call **RequiresAuthentication** in most cases.

RequiresAnyClaim is similar to **RequiresClaims**, with the difference being that at least one should match and not all. For example if your user has *Admin*, *Commenter*, *User* and you ask for **RequiresClaims** listing *Admin*, *User*, then both of those claims must be present. **RequestAnyClaim** will validate on either Admin or User (or both) being present.

RequiresValidatedClaims validates against the claims list attached to the user, but instead of comparing existing data, it requires a lambda pointing to a function that will check and verify the claims in real time.

The final extension, **RequiresHttps**, allows the module to be called only if it's being called via a secure HTTP connection, and will refuse access if it's not.

Each of these extensions are called on your Nancy routing module inside the constructor just before you define your routes, and more than one can be called. Everything you need at a base level is defined in **Nancy.Security**.

With this information under your wing, securing your application becomes adding one or more of the extension methods to your module, as the following code shows:

Code Listing 47

```
using Nancy;
using Nancy.Security;

namespace nancybook.modules
{
  public class AuthRoutes : NancyModule
  {
```

```
    public AuthRoutes() : base("/auth")
    {
      this.RequiresAuthentication();

      Get[@"/"] = _ => View["auth/index"];
    }
  }

}
```

In this case, all that is required is that an authenticated user be present; you can add as many of the others as required to control level of security you require in your application.

Once you've secured your module, you then need to install and set up one of the available three authentication methods, or write your own.

For the remainder of this chapter, I will show you how to get forms authentication up and running.

Forms authentication example

Implementing forms authentication is quite straightforward once you understand how everything works. However, you will need to create a custom bootstrap class, which we have not yet covered.

We'll learn more about what exactly a Bootstrap class is in the next chapter; for now, just follow the instructions to create it, and all will be revealed soon.

Open up your NuGet package manager, and search for and install Nancy.Authentication.Forms. Once you have this installed, create a new class in the root of your application called **`CustomBootstrapper.cs`** and make sure it has the following contents (remember to adjust things like the namespace as needed for your project):

Code Listing 48

```
using Nancy;
using Nancy.Authentication.Forms;
using Nancy.Bootstrapper;
using Nancy.TinyIoc;

namespace nancybook
{
  public class CustomBootstrapper : DefaultNancyBootstrapper
  {
    protected override void ConfigureRequestContainer(
      TinyIoCContainer container,
      NancyContext context)
    {
```

```
      base.ConfigureRequestContainer(container, context);
      container.Register<IUserMapper, FakeDatabase>();
    }

    protected override void RequestStartup(
      TinyIoCContainer container,
      IPipelines pipelines,
      NancyContext context)
    {
      base.RequestStartup(container, pipelines, context);

      var formsAuthConfiguration = new FormsAuthenticationConfiguration
      {
        RedirectUrl = "~/account/login",
        UserMapper = container.Resolve<IUserMapper>()
      };

      FormsAuthentication.Enable(pipelines, formsAuthConfiguration);
    }

  }
}
```

Save and close this bootstrapper class; we won't need to do anything else with it for now.

You now need to create a class that provides an **IUserIdentity** as a result for the authentication calls to query. In this case, we have told Nancy that we're going to create a **FakeDatabase** class to provide it with this information. In reality, we would implement a full way of communicating with a backend data store, and finding an appropriate user's ID. For our purposes, we'll just create a static list of users and allow our app to query against that.

Create a new class called **FakeDatabase.cs**, and make sure it has the following code in it:

Code Listing 49

```
using System;
using System.Collections.Generic;
using System.Linq;
using Nancy;
using Nancy.Authentication.Forms;
using Nancy.Security;

namespace nancybook
{
  public class DatabaseUser : IUserMapper
  {
    private List<FakeDatabaseUser> _users = new List<FakeDatabaseUser>
    {
      new FakeDatabaseUser
      {
```

```
        UserId = new Guid(01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11),
        UserName = "admin",
        Claims = new List<string> {"Admin", "Reader", "Writer"},
        Password = "admin"
      },
      new FakeDatabaseUser
      {
        UserId = new Guid(02, 03, 04, 05, 06, 07, 08, 09, 10, 11, 12),
        UserName = "alice",
        Claims = new List<string> {"Reader", "Writer"},
        Password = "letmeout"
      },
      new FakeDatabaseUser
      {
        UserId = new Guid(03, 04, 05, 06, 07, 08, 09, 10, 11, 12, 13),
        UserName = "bob",
        Claims = new List<string> {"Reader"},
        Password = "letmein"
      }
    };

    public IEnumerable<FakeDatabaseUser> Users { get { return _users; }}

    public IUserIdentity GetUserFromIdentifier(Guid identifier,
NancyContext context)
    {
      var user = _users.FirstOrDefault(x => x.UserId == identifier);
      return user == null
        ? null
        : new AuthenticatedUser
        {
          UserName = user.UserName,
          Claims = user.Claims
        };
    }
  }
}
```

We also need classes for **`FakeDatabaseUser`** and **`AuthenticatedUser`**:

FakeDatabaseUser.cs

Code Listing 50

```
using System;
using System.Collections.Generic;

namespace nancybook
{
  public class FakeDatabaseUser
  {
```

```
    public Guid UserId { get; set; }
    public string UserName { get; set; }
    public List<string> Claims { get; set; }
    public string Password { get; set; }
  }

}
```

AuthenticatedUser.cs

Code Listing 51

```
using System.Collections.Generic;
using Nancy.Security;

namespace nancybook
{
  public class AuthenticatedUser : IUserIdentity
  {
    public string UserName { get; set; }
    public IEnumerable<string> Claims { get; set; }
  }
}
```

With these three classes in place, and the custom bootstrapper setting things up, we should now have everything we need in place to add the authentication into our application.

For the purposes of receiving the login information from an HTML view that implements a form for the user to enter login credentials, we need to create a suitable view model. I've defined this to be:

Code Listing 52

```
using System.ComponentModel.DataAnnotations;

namespace nancybook.Models
{
  public class LoginParams
  {
    [Required]
    public string LoginName { get; set; }

    [Required]
    public string Password { get; set; }

  }

}
```

In a file called **LoginParams.cs**, there's no requirement to model this in any specific way. It's up to you what you provide from your actual client-side experience; all you need to remember is that your user identity in the backend must have a GUID as part of its stored data, and that you must be able to retrieve that GUID and pass it to Nancy when you authenticate the login form.

I'll let you create your own views to implement the various screens needed, but you'll need one to provide a login form, and one to display a login failed message. You'll also need a view for your secure part of the application that will be displayed when the user successfully logs in.

Once you have the views in place, the last thing you need are a couple of Nancy routing modules. The first one is your secure area:

Code Listing 53

```
using Nancy;
using Nancy.Security;

namespace nancybook.modules
{
  public class AuthRoutes : NancyModule
  {
    public AuthRoutes() : base("/auth")
    {
      this.RequiresAuthentication();

      Get[@"/"] = _ => View["auth/index"];
    }
  }
}
```

The second module is a route to provide the ability for the user to log in and log out of the system:

Code Listing 54

```
using System.Linq;
using nancybook.Models;
using Nancy;
using Nancy.Authentication.Forms;
using Nancy.ModelBinding;

namespace nancybook.modules
{
  public class AccountRoutes : NancyModule
  {
    public AccountRoutes() : base("/account")
    {
      Get[@"/login"] = _ => View["account/login"];

      Post[@"/login"] = _ =>
      {
```

```
        var loginParams = this.Bind<LoginParams>();
        FakeDatabaseUser user;

        FakeDatabaseUser db = new FakeDatabaseUser();

        user = db.Users.FirstOrDefault(
          x =>
            x.UserName.Equals(loginParams.LoginName) &&
            x.Password.Equals(loginParams.Password));

        if(user== null)
        {
          return View["account/loginerror"];
        }

        return this.Login(user.UserId, fallbackRedirectUrl: "~/auth");
      };

      Get[@"/logout"] = _ => this.LogoutAndRedirect("~/account/login");
    }
  }
}
```

At this point, you should be able to build your project. Browsing to **/auth** should trigger the login page, allowing you to enter one of the user names and passwords in the fake database class, which in turn should allow you to access the routes in the **`Auth`** module.

Summary

In this chapter, you learned how Nancy's authentication features work, and saw that it's very easy to implement authentication in any way you need to.

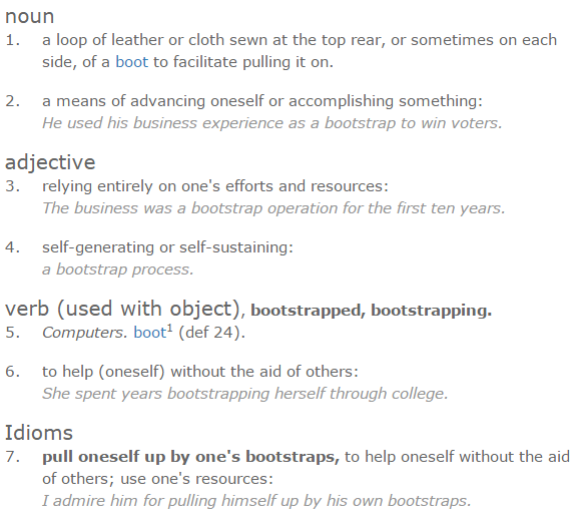
You also learned that there are three common scenarios and authentication methods already defined for implementation, and ready to install using NuGet.

In the next chapter, we'll start to get under the hood of the engine that drives NancyFX. We'll learn what a customized bootstrap class is, and how to use it.

Chapter 11 Bootstrapping

In this chapter, we start getting into the good stuff: we're about to open the bonnet (or the hood, depending on where you live) and start tuning and tinkering around with Nancy's internals.

Just what is a bootstrap? It has nothing to do with that client-side framework everyone uses for making nice, web-based user interfaces. Here is its definition according to the dictionary:



noun

1. a loop of leather or cloth sewn at the top rear, or sometimes on each side, of a boot to facilitate pulling it on.

2. a means of advancing oneself or accomplishing something:
He used his business experience as a bootstrap to win voters.

adjective

3. relying entirely on one's efforts and resources:
The business was a bootstrap operation for the first ten years.

4. self-generating or self-sustaining:
a bootstrap process.

verb (used with object), **bootstrapped, bootstrapping.**

5. *Computers.* boot[1] (def 24).

6. to help (oneself) without the aid of others:
She spent years bootstrapping herself through college.

Idioms

7. **pull oneself up by one's bootstraps,** to help oneself without the aid of others; use one's resources:
I admire him for pulling himself up by his own bootstraps.

Figure 28: Dictionary definition of bootstrap (Courtesy of Dictionary.com)

For our use, bootstrapping (or just booting) is starting up a piece of software or hardware prior to it being in a state that is then useable by an end user.

Where Nancy is concerned, the bootstrap process is the sequence of events that Nancy goes through when an application using it is first run, up to the point where it's ready to start servicing requests sent to it.

A custom bootstrapper, therefore, is a class defined by the application developer and executed at an appropriate place during this startup, allowing the developer to provide customizations and extra startup procedures for use by the Nancy kernel itself.

It's a bit like buying a used car, then tweaking the engine to give it more performance, adding new interiors and a new radio, and providing other improved features to stamp your own identity on the vehicle. By writing and using a custom bootstrapper, you get to make Nancy do all sorts of out-of-the-box extra stuff.

In many cases, configuration of the many extra modules that Nancy has available can only be done in a bootstrapper. Many of the more advanced things you may want to enable can only be done in this way. However, in general, you will find that in some cases, you might not even have to know a bootstrapper exists in order to use a lot of the built-in functionality Nancy has.

Nancy uses the bootstrapper in two ways. It uses it as a method to expose to you the developer—all the hooks, configuration options, and other user-level stuff that can be overridden to the developer working with the framework.

Secondly, Nancy uses the bootstrapper to perform tasks such as configuring the built-in IoC container, TinyIOC, and provides start up points for you to register your own assemblies, in effect wrapping the whole IoC (Inversion of Control) in a domain-specific language (in the words of the NancyFX Wiki).

IoC container

Before we go further, I'd like to introduce another of Nancy's hidden features here: it's built-in IoC container.

For those who are new to the concept, IoC is a method of injecting dependencies into a running code base as required without needing to create new objects. It's an architecture pattern that helps to keep applications composed and easy to manage. It also aids with testing and creating mock objects for functionality that has not been added yet.

What it means for Nancy-based applications is that Nancy will try to satisfy any dependencies required by your code automatically. For example, imagine you had the following route module:

Code Listing 55

```
using System;
using System.Collections.Generic;
using System.IO;
using Nancy;
using Nancy.Responses;

namespace nancybook.modules
{
  public class BaseRoutes : NancyModule
  {
    private FakeDatabase _db;

    public BaseRoutes()
    {
      Get[@"/"] = _ =>
      {
        var _db = new FakeDatabase();
        var myList = db.GetHashCode();
        return View["myview", myList];
      };
    }
  }
}
```

This is a simple example, but an important one.

In this module, your **FakeDatabase** is known as a concrete dependency. That is, it's very hard to manage, and very hard to change. Imagine you had 20 route modules, all with about 10 URLs defined in them, and that each one used that same database library, and had to create a new instance each time it was used.

Now imagine you needed to change that database library from **FakeDatabase** to **RealDatabase**. You suddenly realize just how much work that's going to involve.

An IoC container elevates that considerably. Take the next example:

Code Listing 56

```
using System;
using System.Collections.Generic;
using System.IO;
using Nancy;
using Nancy.Responses;

namespace nancybook.modules
{
  public class BaseRoutes : NancyModule
  {
    private readonly FakeDatabase _db;

    public BaseRoutes(FakeDatabase db)
    {
      _db = db;

      Get[@"/"] = _ =>
      {
        var myList = _db.GetHashCode();
        return View["myview", myList];
      };
    }
  }
}
```

This is exactly the same as the previous example, except now it uses IoC to manage the database library. If I was using this in 10 places in my module, all I'd have to do is change two instances of it: once for the private variable, and once in the constructor parameter.

My IoC library would handle the rest, including making sure I had a new instance connected to **_db** right when I wanted to use it.

It also means that should I wish to, within the Nancy bootstrapper, I can override the automatic settings for this IoC container, and tell Nancy that when something wants to use a **FakeDatabase** object, you should instead give it a **RealDatabase** object. This reduces those two changes in each route module to just one global change in a single class.

Nancy can use several different IoC containers, and has NuGet packages to enable them. If you already use one of these more popular ones, then you might want to consider using it. However, if you haven't used IoC before, but like the idea of how it works, the built-in container more than suffices.

Let's turn our attention back to using the Bootstrapper class.

The default bootstrapper

All this talk of overrides and custom classes might have you thinking that implementing a Nancy bootstrapper is a lot of hard work, and not worth the effort.

This would be very true if you had to implement the entire thing from scratch every time you needed to use one. However, the Nancy team has thought ahead and provided you with a class called **`DefaultNancyBootStrapper`** to abstract from.

All you have to do to get a custom bootstrap class (that implements everything Nancy needs, and allows you to use the bits you need to) is the following:

Code Listing 57

```
using System.Text;
using Nancy;
using Nancy.Bootstrapper;

namespace nancybook
{
  public class CustomBootstrapper : DefaultNancyBootstrapper
  {
  }
}
```

Within the body of this class, you can now create overridden methods to make your own custom implementations of any functionality you need. For example, let's imagine that you had some kind of external data repository library, and this library may use some form of generics, allowing you to use different data objects with the same base class. You can easily declare an interface for your base class, wrap that up in a concrete implementation, and then tell your IoC container what to look for:

Code Listing 58

```
using System.Text;
using demodata;
using demodata.entities;
using Nancy;
using Nancy.Authentication.Forms;
using Nancy.Bootstrapper;
using Nancy.Conventions;
using Nancy.Session;
using Nancy.TinyIoc;
```

```
namespace nancybook
{
  public class CustomBootstrapper : DefaultNancyBootstrapper
  {
    protected override void ConfigureApplicationContainer(TinyIoCContainer
container)
    {
      base.ConfigureApplicationContainer(container);

      container.Register<IDataProvider<Genre>>(new GenreDataProvider());
      container.Register<IDataProvider<Album>>(new AlbumDataProvider());
      container.Register<IDataProvider<Track>>(new TrackDataProvider());
      container.Register<IDataProvider<Artist>>(new ArtistDataProvider());
    }
  }
}
```

You can also use the IoC overrides to make singletons and other instances. In my demo application, for example, I present a small but simple way of providing a low-key form of request caching, and the class I use to manage the cache is set up to be instantiated in the service as a singleton. Another scenario where a singleton class is useful is when writing Nancy services that monitor and provide a web interface to hardware.

All of these things and more are configured in the **`ConfigureApplicationContainer`** override. There are many other too; if you start typing “protected override…” in your class, and pause, IntelliSense should list them all for you, along with their signatures:

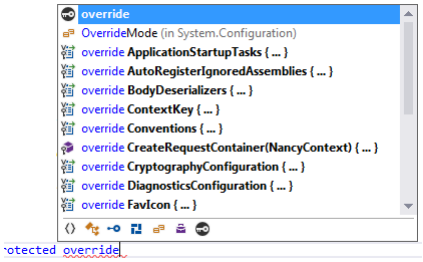


Figure 29: IntelliSense showing the available bootstrapper overrides

In reality, you'll commonly only use a few different overrides. In my example app, for instance, I've used **`RequestStartup`** to set up forms authentication, **`ConfigureRequestContainer`** to configure my **`IUserMapper`**, **`ConfigureApplicationContainer`** to register my external IoC dependencies, **`ConfigureConventions`** to set up static file folders for my web application, and **`ApplicationStartup`** to add an error-handling pipeline.

You'll find that the naming is deliberate, too: for overrides that use “Request” in the name, Nancy will call these hooks on a request basis. These are the usual places where you'd look for things like request authentication, caching, or checking that a given header is available.

Anything that has “Application” in the name is generally specific to the application as a whole, and will be called when that app is started. In the case of a standalone service, that would be when the app is physically started using the service control manager, or when it's run from the Windows desktop by clicking on it. For applications that are hosted using ASP.NET and IIS7, then this will get called whenever the server’s app-pool is recycled.

A bootstrapper example

One of the most useful overrides you can re-implement is the **`ConfigureConventions`** override.

You will remember from previous chapters that Nancy looks for all its content in one place: underneath a folder called **Content** relative to the running assembly.

If you have the defaults in place, then this means that scripts will likely be in **Contents/Scripts/** followed by the name of the JavaScript you want to use.

You can easily change this by overriding the **`ConfigureConventions`** method and adding your custom paths. For my demo application, I did the following:

Code Listing 59

```
using System.Text;
using demodata;
using demodata.entities;
using Nancy;
using Nancy.Authentication.Forms;
using Nancy.Bootstrapper;
using Nancy.Conventions;
using Nancy.Session;
using Nancy.TinyIoc;

namespace nancybook
{
  public class CustomBootstrapper : DefaultNancyBootstrapper
  {
    protected override void ConfigureConventions(NancyConventions
nancyConventions)
    {
      Conventions.StaticContentsConventions
        .Add(StaticContentConventionBuilder.AddDirectory("/scripts",
@"Scripts"));
      Conventions.StaticContentsConventions
        .Add(StaticContentConventionBuilder.AddDirectory("/fonts",
@"fonts"));
      Conventions.StaticContentsConventions
        .Add(StaticContentConventionBuilder.AddDirectory("/images",
@"Images"));
      Conventions.StaticContentsConventions
        .Add(StaticContentConventionBuilder.AddDirectory("/", @"Pages"));
```

```
    }
  }
}
```

I then created **scripts**, **fonts**, **images** and **pages** in my Solution Explorer, and ensured that **`NamespaceProvider`** was set to **`True`** for them, so they would be copied to the same location as my compiled assembly once run.

This allowed me to use the normal NuGet installation procedures for CSS, JavaScript, and other ASP.NET-related web packages, and meant that any files I included would be copied to my application build.

You'll notice that all requests to **`/`** are set to look in a folder called **Pages**; the one thing you cannot do here is point **`/`** to your application root; Nancy recognises the security risk this poses and won't allow it to happen.

What this means in practice is that I can put normal, static HTML files in the pages folder, and then add **/blah.html** to make a request. Nancy will return a static HTML file called **blah.html** from that folder. This allows you to mix traditional classic HTML, Scripts, and CSS files that are served without being processed by Nancy, alongside Nancy's powerful routing engine and API/URL features. I'm employing the very method for the work I've been doing on the Lidnug website, where I'm delivering the user interface directly at high speed using JavaScript for interaction on the client. The JavaScript in this user interface then calls on different endpoints provided by Nancy's web framework to service the AJAX requests made from the client.

If you've used NodeJS at all, you'll recognize almost immediately that this is exactly the same way that it works when used with frameworks such as Express.

The clever part here is not the way it works, but the fact that you can take your application and deploy it freely to a server with or without a web server already in place, which makes it perfect for creating lightweight microservice architecture-based deployments.

Summary

In this chapter, you learned how Nancy's bootstrapper works, and how its built-in IoC container can simplify how you design your Nancy applications.

You also learned how to use the bootstrapper to create static conventions that work on all platforms, allowing you to serve your site up even without using a web server.

In the next chapter, we'll take a look at Nancy's request pipelines, which are usually configured in the bootstrapper.

Chapter 12 Pipeline Interception

All good web frameworks have a request-processing pipeline, and Nancy is up there with the best of them. What does it mean to have a request pipeline? Maybe the following diagram will help clear things up:

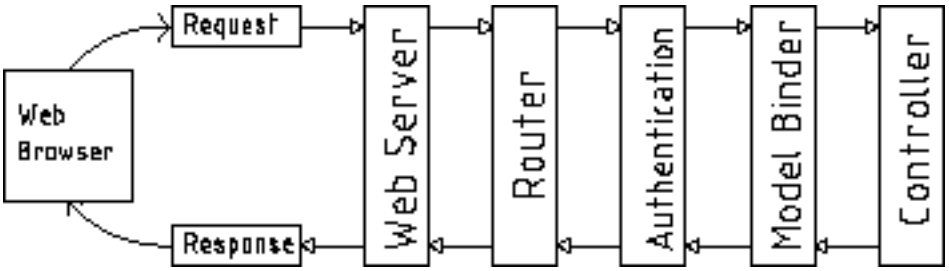


Figure 30: Typical web pipeline diagram

Exactly as the name portrays, a typical web request goes through a number of modules before it reaches its final response, and then it often goes back through all of these modules on its way back to the browser.

How many and what type of these modules the request goes through is highly dependent on the framework used; some have more, some have less. The more there is to process, the slower the response often is.

Nancy, being as lightweight and modular as it is by default, has very few steps in its pipeline, and it gives the developer a LOT of control on how and where these modules interact with the framework as a whole.

When we talk about pipeline interception, we're referring to the three main hook points that Nancy exposes, allowing you to hook your own modules or snippets of code into this processing chain.

You've seen so far that Nancy has a remarkable amount of functionality—authentication is a great example—and you might be thinking that there's a lot of code in the pipeline to support it.

The fact of the matter is, Nancy has almost no code in its pipeline to make authentication and other similar things happen. Even when you add an authentication method, no code is added to the pipeline. It's only when you add the **`this.RequiresAuthentication();`** line to your routing modules that the pipeline is actually hooked into and intercepted.

Why?

Well, the authentication extensions are actually just thin wrappers around a small piece of in-line code that adds a lambda (an anonymous function for you Java and JavaScript types) to the **`Before`** and **`After`** pipes for that routing module.

In the example shown on Nancy's Authentication page, you can see the following example:

Code Listing 60

```
using Nancy;
using Nancy.Responses;
using Nancy.Security;

namespace nancybook.modules
{
  public class AuthRoutes : NancyModule
  {
    public AuthRoutes() : base("/auth")
    {
      Before += ctx =>
      {
        return (this.Context.CurrentUser == null)
          ? new HtmlResponse(HttpStatusCode.Unauthorized)
          : null;
      };

      Get[@"/"] = _ => View["auth/index"];
    }
  }
}
```

This actually performs the same function as **`RequiresAuthentication`**, but in a shorter and more succinct way.

Within the bit of code used by the authentication system, you can see straightaway that it has two possible outcomes. First off, you can return null, in which case the processing then continues on to run the actual code defined in your route, and has no effect on anything else further down.

The second option is to return a Nancy response object (in this case a **`403 Unauthorized`**). Anything you can return from a route handler, you can return from the **`Before`** pipeline, and if you do, this will be returned straight back to the calling client without ever calling the code in your route handler module.

You should be able see straightaway that this has many uses, not just authentication. However, you could check for other resources, services, or all manner of other conditions and send an error or differing response back to your client before anything else is acted on.

If you wanted to provide some form of front-end caching, for example, you might intercept the **`Before`** pipe and see if that request is already cached, returning the cached version if it is, and only allowing the request to progress through to the route handler if it's not, or if the data needs renewing.

You can also provide an **`After`** handler by doing the following:

Code Listing 61

```
After += ctx =>
{
```

```
  //... Code here ...
}
```

With the **After** pipeline, you'll already have a response object (the one generated by the route that handled the request), and from the Nancy context (available in the **ctx** variable in the example), you're free to change that as you see fit.

For example, you could check some other environment variable, and decide to change the already-generated **200 ok** response to a **403** response before it gets back to the browser.

You would, however, be more likely to use the **After** pipeline to grab the response that's about to be sent back to the requesting client, and add it to your cache service, so that the next request that comes in will find it pre-cached and not need to re-request it from the database.

The possibilities for the uses of the **Before** and **After** pipeline handlers are limited only by your imagination and how fast you can do something.

Remember that these snippets of code are called for every request into the route module to which they're attached. This means if your code takes a long time to execute, then you WILL slow down the speed at which Nancy serves your requests.

Application-wide hooking

You can also wire up before and after pipeline hooks for your entire application, rather than just on a module basis. Using the application-wide hooks means that your hook will fire for EVERY route and module in the application, no matter where it is. Furthermore, if you have an application-wide hook, and then an individual hook on a specific module, both hooks will be called, with the application hook getting called first.

This also means that if you return a response in the application hook, then your module hook will never get called, in exactly the same manner as your route would never get called if you returned a response from your module hook.

You hook up your application pipelines in a custom bootstrapper, either by overriding **ApplicationStartup** or **RequestStartup**. One of the parameters to either of these functions is the **Pipelines** parameter, which includes the **BeforeRequest** and **AfterRequest** properties. You assign to these properties in exactly the same way as you assign to the **Before** and **After** hooks in an individual module.

You'll also find that you have access to an **OnError** property, which can come in handy to implement custom error-handling in your application. For example, in your database access code, you might want to throw a custom **Database Object not found** exception when a requested entity is not present in the database. You could then use your custom bootstrapper to intercept this error and return a **404 File not found** error, something like the following:

Code Listing 62

```
using System.Text;
using Nancy;
```

```
using Nancy.Authentication.Forms;
using Nancy.Bootstrapper;
using Nancy.Conventions;
using Nancy.Session;
using Nancy.TinyIoc;

namespace nancybook
{
  public class CustomBootstrapper : DefaultNancyBootstrapper
  {
    protected override void ApplicationStartup(
      TinyIoCContainer container,
      IPipelines pipelines)
    {
      base.ApplicationStartup(container, pipelines);

      // Add an error handler to catch our entity not found exceptions
      pipelines.OnError += (context, exception) =>
      {
        // If we've raised an EntityNotFound exception in our data layer
        if (exception is EntityNotFoundException)
          return new Response()
          {
            StatusCode = HttpStatusCode.NotFound,
            ContentType = "text/html",
            Contents = (stream) =>
            {
              var errorMessage = Encoding.UTF8.GetBytes("Entity not
found");
              stream.Write(errorMessage, 0, errorMessage.Length);
            }
          };

        // If none of the above handles our exception, then pass it on as a
500
        throw exception;
      };
    }
  }
}
```

In your data access code, you can then just add **`throw new EntityNotFoundException();`** at any appropriate point, and Nancy will convert it to a **`404`** and return it to the client.

Output caching

To give you a good example of how to use the various hooks, I'll close this chapter with an example that provides a simple but useful sliding output cache for your routes.

The example will cache the output from a route, and providing that it's called again within 30 seconds of the last access, it will continue to cache the response, only clearing the cached version if a period longer than 30 seconds has elapsed before another request is made.

The first thing we need is a small class that acts as the cache provider. Create a new class within your application called **`CacheService.cs`**, and add the following code to this class:

Code Listing 63

```
using System;
using System.Collections.Specialized;
using System.Runtime.Caching;
using Nancy;

namespace nancybook
{
  public class CacheService
  {
    private static readonly NameValueCollection _config = new
NameValueCollection();
    private readonly MemoryCache _cache = new MemoryCache("NancyCache",
_config);

    private readonly CacheItemPolicy _standardPolicy = new CacheItemPolicy
    {
      Priority = CacheItemPriority.NotRemovable,
      SlidingExpiration = TimeSpan.FromSeconds(30) // This can be changed
to FromMinutes/FromHours etc it's 30 secs for testing purposes
    };

    public void AddItem(string itemKey, Response itemToAdd)
    {
      _cache.Add(new CacheItem(itemKey, itemToAdd), _standardPolicy);
    }

    public Response GetItem(string itemKey)
    {
      return (Response)_cache.Get(itemKey);
    }

  }
}
```

This class provides two public methods for getting items from and adding them to the cache. You'll notice that these objects are Nancy response objects.

Since we need this class to be registered as a singleton instance, we need to add a configuration to our bootstrapper to configure the IoC container correctly:

Code Listing 64

```
protected override void ConfigureApplicationContainer(TinyIoCContainer
container)
{
  base.ConfigureApplicationContainer(container);
  container.Register<CacheService>().AsSingleton();
}
```

This will ensure that every request that has to deal with the cache is interacting with the same instance, and not a new one in each request.

Next, create a route module that implements a route to display a view, and implements module-level pipelines to check and set the cache for that route as needed:

Code Listing 65

```
using System;
using System.Collections.Generic;
using System.Runtime.InteropServices;
using nancybook.Models;
using Nancy;

namespace nancybook.modules
{
  public class CachingRoutes : NancyModule
  {
    private readonly CacheService _myCache;

    public CachingRoutes(CacheService myCache) : base("/caching")
    {
      _myCache = myCache;
      Get["/"] = x =>
      {
        var cacheData = new CacheDemo() {WhenRequested = DateTime.Now};
        return View["caching/index.html", cacheData];
      };
      Before += ctx =>
      {
        string key = ctx.Request.Path;
        var cacheObject = _myCache.GetItem(key);
        return cacheObject;
      };
      After += ctx =>
      {
        if(ctx.Response.StatusCode != HttpStatusCode.OK)
        {
          return;
        }
        string key = ctx.Request.Path;
        if(_myCache.GetItem(key) == null)
```

```
            _myCache.AddItem(key, ctx.Response);
        };
    }
  }
}
```

Finally, add a view to your **Views** folder and a model to the project, so you have something to demonstrate that the caching is taking place: **caching/index.html**.

Code Listing 66

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Nancy Demo | Caching Example</title>
  <link href="~/content/bootstrap.min.css" rel="stylesheet"
type="text/css"/>
</head>

<body>

<div class="container">
  <div class="page-header">
    <h1 style="display: inline-block">Nancy Demo <small>Caching
Example</small></h1>
    <h1 style="display: inline-block" class="pull-right"><small><a
href="~/" title="Click to return to demo home page">home <span
class="glyphicon glyphicon-home"></span></a></small></h1>
  </div>

  <h4>This page was requested at <strong class="text-
success">@Model.WhenRequested</strong></h4>

  <br/><br/>

  <p class="lead">This example uses before and after filters attached
directly to the module servicing this request.</p>
  <p>If you observe the time this page was created when refreshing it,
you'll see the page is handled by an output cache; this cache has a sliding
window of 30 seconds.</p>
  <p>
    As long as you're refreshing the page, you'll reset the timer and the
cache will continue to wait 30 seconds after the last request before
expiring. If you request the page then
    leave it for 30 seconds before re-requesting it, you'll see you get a
new copy.
  </p>
```

```
</div>

<script src="~/scripts/jquery-2.1.3.min.js"></script>
<script src="~/scripts/bootstrap.min.js"></script>
</body>

</html>
```

CacheDemo.cs

Code Listing 67

```
using System;

namespace nancybook.Models
{
  public class CacheDemo
  {
    public DateTime WhenRequested { get; set; }
  }
}
```

Summary

In this chapter, you learned about Nancy's pipeline hooking mechanism, allowing you to freely perform all sorts of pre- and post-request operations on anything within your Nancy-based application.

You learned how to handle exceptions at a global level, and how to implement a very simple frontend output cache.

In the next chapter, we have something to please the hard core TDD purists among you: Nancy's built-in testing framework.

Chapter 13 Testing

One of the real hidden gems when it comes to Nancy is the testing framework provided with it. Nancy was built from the word “go” with the ability to test it in mind. Because of this, the super-duper-happy-path extends not only to developing applications with it, but to testing applications too.

Like everything else you can add to Nancy, you can get the testing framework using NuGet, by installing **Nancy.Testing**.

In order to take a closer look at how to use the testing framework, add a new class library project to your existing solution. You can create the tests as part of your actual application if you wish, but keeping them in a separate assembly is both recommended and sane.

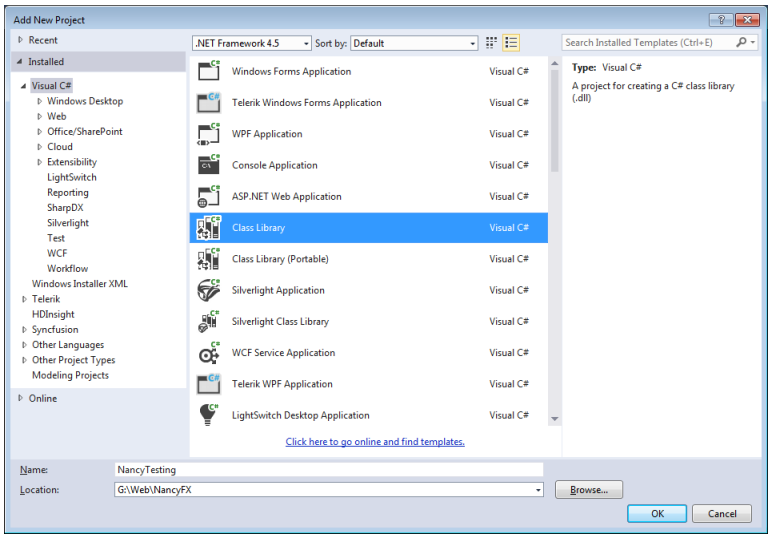


Figure 31: Adding a new class library project to put our tests in

Once you have added your new class library project, pull up your NuGet Package Manager and add **Nancy.Testing** to the new project:

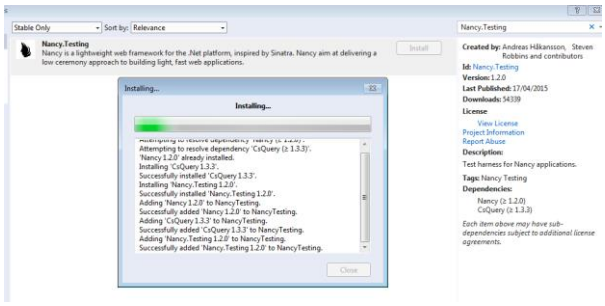


Figure 32: Adding the Nancy testing package

While you're in your Package Manager, add the **NUnit** testing framework (or whichever test framework you prefer to use) to the project. I Use NUnit because I'm a JetBrains ReSharper user, and R# has built-in support for NUnit by default, allowing me to easily manage my tests.

The tests I write in this chapter to demo Nancy's testing abilities will be written using NUnit's syntax. If you use a different test framework, you'll need to alter your tests so that they use the syntax your framework and test runner support.

Add a new class to your test project and add in the following code:

Code Listing 68

```
using nancybook.modules;
using Nancy;
using Nancy.Testing;
using NUnit.Framework;

namespace NancyTesting
{
  class GetTest
  {
    [Test]
    public void SimpleTestOne()
    {
      // Arrange
      var browser = new Browser(with => with.Module(new TestingRoutes()));

      // Act
      var response = browser.Get("/");

      // Assert
      Assert.AreEqual(HttpStatusCode.OK, response.StatusCode);

    }
  }
}
```

You'll see that **`TestingRoutes`** is in red, and if you run your test at this point, it will fail to build.

Add a project reference to your main Nancy-based application to the testing project, and then in your main Nancy application, add an empty route module called **`TestingRoutes`**.cs as follows:

Code Listing 69

```
using System;
using System.Collections.Generic;
using System.IO;
using Nancy;
using Nancy.Responses;

namespace nancybook.modules
```

```
{
  public class TestingRoutes : NancyModule
  {
  }
}
```

Update your test to include the namespace where your new module lives. In my case, I added **`using nancybook.modules;`** to the **`using`** clause of my test assembly.

At this point, the **`TestingRoutes`** reference should change from red to your default class name color, indicating that it can be referenced.

If you try running your test now, you should find that the test compiles and runs. It still fails, however, as we've not yet implemented the route we're testing for.

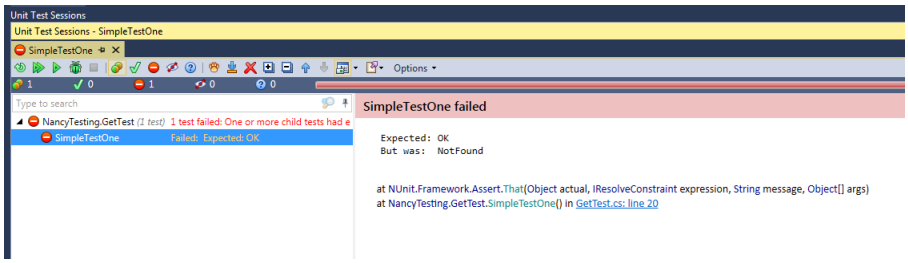


Figure 33: Our first test failing

Go back to your main Nancy application, and update the testing routes module so that it looks like this:

```
using Nancy;

namespace nancybook.modules
{
  public class TestingRoutes : NancyModule
  {
    public TestingRoutes()
    {
      Get[@"/"] = _ => "Hello World";
    }
  }
}
```

Run your test again, and this time it should pass.

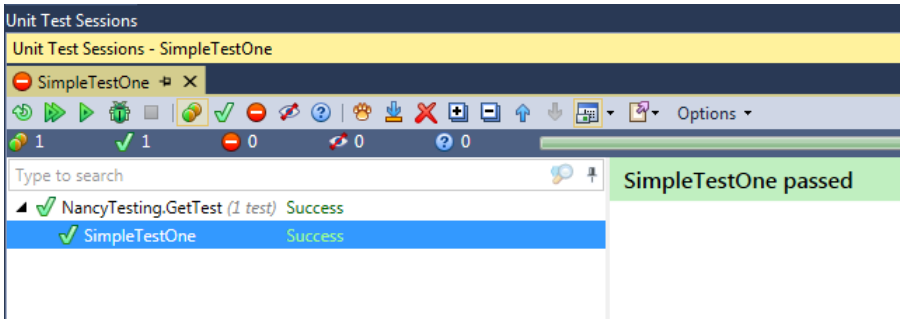


Figure 34: This time our test passes

Nancy provides you with a **`Browser`** object. This object is not an abstraction of the browser in your system; its a custom, self-running headless browser instance that's able to perform many of the things your browser can do, only in isolation.

For example, you can test your post acceptors very easily by asking Nancy to send form values to your routes and examine the response returned. Let's imagine we have a route that allows us to store a name and email address in the database, and we want that route to provide a status code of **`200`** if the record was saved, **`409`** if it already existed, or **`500`** for any other error.

To do this, we'd most likely create three tests as follows:

Code Listing 70

```
    [Test]
    public void PostTest_Should_Return_200()
    {
      // Arrange
      var browser = new Browser(with => with.Module(new TestingRoutes()));
      // Act
      var response = browser.Post("/save/", (with) =>
      {
        with.HttpRequest();
        with.FormValue("Name", "Joe Smith");
        with.FormValue("Email", "joe@anemail.com");
      });
      // Assert
      Assert.AreEqual(HttpStatusCode.OK, response.StatusCode)
    }
    [Test]
    public void PostTest_Should_Return_409()
    {
      // Arrange
      var browser = new Browser(with => with.Module(new TestingRoutes()));
      // Act
      var response = browser.Post("/save/", (with) =>
      {
        with.HttpRequest();
        with.FormValue("Name", "Existing Person");
        with.FormValue("Email", "existing@anemail.com");
```

```
      });
      // Assert
      Assert.AreEqual(HttpStatusCode.Conflict, response.StatusCode);
    }
    [Test]
    public void PostTest_Should_Return_500()
    {
      // Arrange
      var browser = new Browser(with => with.Module(new TestingRoutes()));
      // Act
      var response = browser.Post("/save/", (with) =>
      {
        with.HttpRequest();
        with.FormValue("NOTName", "Existing Person");
        with.FormValue("NOTEmail", "existing@anemail.com");
      });
      // Assert
      Assert.AreEqual(HttpStatusCode.InternalServerError,
response.StatusCode);
    }
```

If you run these immediately, you'll see that they fail, as expected.

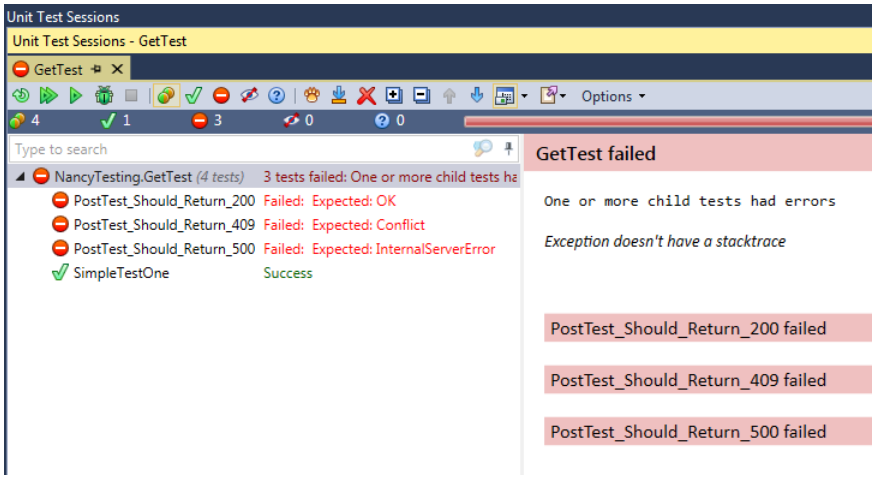


Figure 35: All our post tests fail

Let's switch back to our testing routes module in the main application and check for our parameters; if they don't exist, we'll return a **500** error.

Code Listing 71

```
using Nancy;

namespace nancybook.modules
{
  public class TestingRoutes : NancyModule
  {
```

```
    public TestingRoutes()
    {
      Get[@"/"] = _ => "Hello World";

      Post[@"/save"] = _ =>
      {
        if (!Request.Form.Name.HasValue || !Request.Form.Email.HasValue)
        {
          return 500;
        }

        return null;
      };

    }
  }
}
```

Rerun your tests, and all being well, the **500** one should pass, but the other two should still fail. You might actually find that the **200** test passes too; even though we returned a null, Nancy will still see that as a successful test and pass a **200 ok** by default. If it does, don't worry—for our test purposes, it's perfectly fine.

Let's make our test actually pass the other two tests as expected. Change your route module so that the code now looks like this:

Code Listing 72

```
using Nancy;

namespace nancybook.modules
{
  public class TestingRoutes : NancyModule
  {
    public TestingRoutes()
    {
      Get[@"/"] = _ => "Hello World";

      Post[@"/save"] = _ =>
      {
        if (!Request.Form.Name.HasValue || !Request.Form.Email.HasValue)
        {
          return HttpStatusCode.InternalServerError;
        }

        if (Request.Form.Name == "Existing Person" && Request.Form.Email ==
"existing@anemail.com")
        {
          return HttpStatusCode.Conflict;
        }
```

```
        return HttpStatusCode.OK;
      };

    }
  }
}
```

This time, all three tests should pass.

You can also check for other results, such as redirects. Let's imagine that instead of returning a **500**, our route module returned a redirect to an error page. We can test for this by changing the test as follows:

Code Listing 73

```
    [Test]
    public void PostTest_Should_Return_500()
    {
      // Arrange
      var browser = new Browser(with => with.Module(new TestingRoutes()));

      // Act
      var response = browser.Post("/save/", (with) =>
      {
        with.HttpRequest();
        with.FormValue("NOTName", "Existing Person");
        with.FormValue("NOTEmail", "existing@anemail.com");
      });

      // Assert
      response.ShouldHaveRedirectedTo("/posterror");
    }
```

Running it without changing the route should once again fail. However, we can change the route so that it performs a redirect as follows:

Code Listing 74

```
using Nancy;

namespace nancybook.modules
{
  public class TestingRoutes : NancyModule
  {
    public TestingRoutes()
    {
      Get[@"/"] = _ => "Hello World";

      Post[@"/save"] = _ =>
      {
        if (!Request.Form.Name.HasValue || !Request.Form.Email.HasValue)
```

```
        {
          return Response.AsRedirect("/posterror");
        }

        if (Request.Form.Name == "Existing Person" && Request.Form.Email ==
"existing@anemail.com")
        {
          return HttpStatusCode.Conflict;
        }

        return HttpStatusCode.OK;
      };

    }
  }
}
```

This will once again cause the test to pass.

Nancy's testing features can also check the returned HTML content for given elements, classes, and IDs; in fact, if you can create a CSS3 style selector for it, then you can test for it.

The test classes use CsQuery under the hood for this, and speaking from experience, I've yet to find a selector it's not been able to handle.

Here's an example from the NancyFX Wiki: if you wanted to submit an incorrect login to a login form, then check that the output contained an HTML element called **errorBox**, with a certain CSS class, you could use the following test to do this easily:

Code Listing 75

```
    [Test]
    public void Should_display_error_message_when_error_passed()
    {
      // Given
      var bootstrapper = new DefaultNancyBootstrapper();
      var browser = new Browser(bootstrapper);

      // When
      var response = browser.Get("/login", (with) =>
      {
        with.HttpRequest();
        with.Query("error", "true");
      });

      // Then
      response.Body["#errorBox"]
              .ShouldExistOnce()
              .And.ShouldBeOfClass("floatingError")
              .And.ShouldContain(
                  "invalid",
```

```
                    StringComparison.InvariantCultureIgnoreCase);
    }
```

You'll notice in that example that we created the browser using a default Nancy bootstrapper. You can modify this bootstrapper in exactly the same way as you modify your custom bootstrappers in the main application. You can also use it to configure the IoC container to further aid you in your testing efforts.

Summary

In this chapter, you saw much more of the super-duper-happy-path. You saw how easy Nancy makes it for you to test your Nancy applications, and how it solves one of the biggest headaches that ALL web developers face: testing the content of the HTML that makes up user interfaces.

Over the last 100 pages, you've discovered (I hope) that Nancy once again brings joy to developing web-based services. When I was discussing this with some of my peers, getting ideas for what to include in the book, one of them said something to me that's stuck with me since I started writing it: "NancyFX? Isn't that the toolkit that tries to be NodeJS for C#?"

NancyFX is much more, in my opinion, than NodeJS can ever be (although Node does come a very close second). NancyFX represents a framework that has been built by developers who have gone through all the pain that the rest of us have, and have come out of the other side bearing the scars.

The next time you're browsing around on Twitter, remember to ping a message to **@TheCodeJunkie** and **@GrumpyDev** thanking them for making your life and web development efforts so much easier. If you'd like to participate in the project and possibly become a Nancy MVM (Most Valued Minion) then hop over to NancyFX.org and read the docs.

There's so more you can do with the framework—we've reached the end of our journey in this book, but your journey with Nancy has just begun.

Figure 36: Enjoy the "Super-Duper-Happy-Path" to web application development.

Appendix: NuGet Packages

Due to its modular nature, if you search for "Nancy" using NuGet, you'll likely find that there's more than just a few NuGet packages marked as being part of the project.

In this appendix, I'll list the more common ones and provide a brief description of what they provide and their intended use.

All the packages listed in this chapter and any versions mentioned are current as of the date of writing this chapter, but by the time you get this book, you may find that some versions have advanced and have newer functionality.

Nancy.Authentication.Forms

This package provides standard ASP.NET/IIS-based forms authentication services to a Nancy-based application.

When authentication is enabled using this module, the intention is that it behaves and works the same way that standard ASP.NET forms authentication works in a non-Nancy enabled application.

Nancy.Authentication.Stateless

This package provides a stateless way of authenticating users to a Nancy-enabled application.

It's designed to support scenarios such as "API Key" authentication, or endpoint authentication, where privilege is re-assessed on every single request made to the service.

Nancy.Authentication.Basic

This package adds basic (Base64) authorization header facilities to Nancy.

It's designed to add simple, browser password-based authentication to a Nancy-enabled application. This type of authentication is often used for internal websites and is not designed for external sites due to its unsecure nature.

Nancy.Authentication.Token

This package adds token-based (similar to OAuth) authentication services to Nancy.

The primary reasoning for this authentication method is to authenticate to an API once and then each time present an issued token. Its primary intended use is in things like SPAs, and iOS and Android mobile apps where cookies may not always be available, and where user details are never retrieved and sent across the wire.

Nancy.Hosting.Aspnet

This package provides hosting platform binding for ASP.NET running under IIS web server.

If you're looking to host a Nancy-based application alongside Webforms, MVC, or just a standard IIS-based install, this package is used to enable that scenario.

Nancy.Hosting.Self

This package provides self-hosting platform bindings for standalone, Nancy-enabled applications.

The primary use of this package is to allow console mode, Windows service, Windows desktop, and possibly mobile device applications to expose an HTTP-compatible endpoint without any other HTTP-based server software being installed.

Nancy.Hosting.Wcf

This package provides hosting platform bindings for Windows communications foundation.

Where you have WCF-based services providing endpoints for your web applications, this package may be used to Nancy enable those services or to host and introduce Nancy-based services alongside them without changing the underlying infrastructure.

Nancy.BootStrappers.Structuremap

This package provides dependency injection facilities to Nancy using the StructureMap DI library.

If StructureMap is your DI container of choice, this package enables its use with Nancy.

Nancy.BootStrappers.Mef

This package provides dependency injection facilities to Nancy using the Microsoft MEF (Managed Extensibility Framework) to provide DI facilities.

If MEF is your DI container of choice, this package enables its use with Nancy.

Nancy.BootStrappersAutofac

This package provides dependency injection facilities to Nancy using the Autofac DI library.

If Autofac is your DI container of choice, this package enables its use with Nancy.

Nancy.BootStrappers.Ninject

This package provides dependency injection facilities to Nancy using the Ninject DI library.

If Ninject is your DI container of choice, this package enables its use with Nancy.

Nancy.BootStrappers.Unity

This package provides dependency injection facilities to Nancy using the Unity DI library.

If Unity is your DI container of choice, this package enables its use with Nancy.

Nancy.BootStrappers.Windsor

This package provides dependency injection facilities to Nancy using the Castle Windsor DI library.

If Castle Windsor is your DI container of choice, this package enables its use with Nancy.

Nancy.ViewEngines.NHaml

This package allows Nancy-enabled applications to use views composed using the NHaml templating engine.

If your chosen template/view composition language is based on NHaml, this package will allow you to use that technology to compose views for your routes.

Nancy.ViewEngines.DotLiquid

This package allows Nancy-enabled applications to use views composed using the DotLiquid view engine.

If your chosen template/view composition language is based on DotLiquid, this package will allow you to use that technology to compose views for your routes.

Nancy.ViewEngines.MarkDown

This package allows Nancy-enabled applications to use views composed using the MarkDown content engine.

If your chosen template/view composition language is based on MarkDown, this package will allow you to use that technology to compose views for your routes.

Nancy.ViewEngines.Nustache

This package allows Nancy-enabled applications to use views composed using the Nustache templating engine.

If your chosen template/view composition language is based on Nustache (the .NET version of Mustache), this package will allow you to use that technology to compose views for your routes.

Nancy.ViewEngines.Razor

This package allows Nancy-enabled applications to use views composed using the Microsoft Razor view engine.

If your chosen template/view composition language is based on the standard ASP.NET MVC Razor template language, this package will allow you to use that technology to compose views for your routes.

Nancy.ViewEngines.Spark

This package allows Nancy-enabled applications to use views composed using the Spark view engine.

If your chosen template/view composition language is based on the Spark template language, this package will allow you to use that technology to compose views for your routes.

Nancy.Validation.FluentValidation

This package allows Nancy-enabled applications to use the validation services provided by the FluentValidation .NET toolkit written by Jeremy Skinner.

If you already use fluent validations or would like a large amount of control over how your models are validated, this package will allow you to do just that.

Nancy.Validation.DataAnnotations

This package enables Nancy applications to use the standard aspect-driven method of model validation.

If you're used to using the ASP.NET MVC method of validating models, then using this package will enable that functionality.