KaffeeKlatsch
Das Magazin rund um Software-Entwicklung

ISSN 1865-682X
07/2012

Der Super-Duper-Happy-Pfad

2 1/2 Blicke auf großartige Open-Source-Frameworks für .NET

von Timothée Bourguignon

Der Super-Duper-Happy-Pfad

2 1/2 Blicke auf großartige Open-Source-Frameworks für .NET

von Timothée Bourguignon

Eine Web-Applikation zu schreiben sollte purer Spaß sein. Ein neues Projekt aufzubauen sollte reibungslos funktionieren ohne Gänsehaut beim Gedanken, verschiedene Frameworks und Tools zusammen zu führen. Die *.NET*-Welt außerhalb von Microsoft wuchert nicht so wild wie bei anderen Technologien, aber im Open-Source-Universum erscheinen doch ab und zu beeindruckende Projekte: *Nancy*[1] und *Simple.Data*[2] sind zwei davon. Werden diese zusammen und in Kombination mit *MongoDB*[3] genutzt, entsteht bald der „Super-Duper-Happy-Pfad".

Erstes Date mit Nancy

Nancy gehört zu der Familie der *Micro-Frameworks* und ist inspiriert vom *Ruby*-Framework *Sinatra* [4]. Nancy ist ein leichtgewichtiges Framework mit dem man Web-Seiten entwickeln kann. Und nur das! Anstatt gebündelt zusammen mit zig Tools zu kommen – wie *ASP.NET MVC* oder *Ruby on Rails* – bringt Nancy nur genügend Features mit um sich selber aufzubauen, die Möglichkeit auf *HTTP-Requests* zu reagieren und Daten zu und von *Views* zu bewegen.

Simple.Data ist ein ganz kleiner *Object Relational Mapper* (ORM), der keine Objekte benutzt, auch mit nicht-relationalen Datenbanken funktioniert und kein Mapping braucht. Also ein ORM ohne O und nicht unbedingt R oder M. Er wurde geschrieben um *SQL-Injections* zu vermeiden. Er wurde von *ActiveRecord* und *DataMapper* aus Ruby on Rails inspiriert und basiert auf dem „dynamic" Schlüsselwort unter dem .NET-Framework 4.

MongoDB ist eine dokumentenorientierte *NoSQL*-Datenbank. Anstatt Tabellen zu verwenden, benutzt MongoDB *JSON*-artige Dokumente mit dynamischem Schema, dem *BSON*. Dies hat viele Vorteile, insbesondere ein sehr einfaches Speichern von komplexen Objekten, die in einer relationalen Datenbank mehrere Tabellen brauchen würden und bietet dadurch, da sie Schemalos sind, eine riesige Flexibilität.

Alle drei Tools sind Open-Source und werden durch aktive Communities unterstützt.

Liebe auf den ersten Blick

Nancy wurde auf dem *.NET Client Profile* gebaut. Es braucht also sehr wenig zum Laufen. Es funktioniert auch unter *Mono* [5] und kann auf verschiedenen Web-Servern laufen, inkl. *IIS* und *OWIN* [6], kann sich aber auch selber „hosten".

Um ein Nancy-Projekt anzufangen, baut man am besten ein leeres *ASP.NET*-Projekt in Visual Studio und installiert das *Nancy.Hosting.Aspnet Nuget* [7] Package. Damit referenziert Nancy zwei *DLLs* und fügt ihre *HTTP-Handler* an die *web.config*-Datei hinzu. Das war's auch schon!

Nancys Herz

Nancys zentraler Baustein ist das *Modul*. Es ist ein bisschen wie ein ASP.NET MVC-Kontroller und ist das einzige Teil, um das man nicht herum kommt. Module sind die Stellen, an denen man die unterstützten *Routen* und das Verhalten der Applikation definiert.

Ein Modul muss von der *NancyModule* Basisklasse erben. Neben der Definition von Verhalten, bringen Module Informationen über aktuelle Anfragen, Kontext der Anfragen, *Response Helpers* und viel mehr mit. Endlich werden damit alle Module dynamisch „gefunden"; eine Registrierung ist nicht nötig.

Routen werden im Konstruktor eines Moduls definiert und mit *Lambda-Expressions* vom Typ *Func<dynamic, Response>* geschrieben. Eingabe ist ein dynamisches *Nancy-Dictionary* und Antwort ein *Nancy-Response*-Objekt. Das eingegebene Dictionary enthält alle zusätzlichen Informationen über die Routen, wie zum Beispiel Parameter.

Aber genug, hier nun Nancys „Hallo Welt". Wir schreiben unsere Module in eine neue *C#*-Klasse:

```
using Nancy;
public class GraffitiModule : NancyModule {
  public GraffitiModule() {
    Get["/"] = _ => {
      return "Hello World";
    };
  }
}
```

Dieses Stück Code erklärt Nancy, dass die Antwort zu einem *HTTP Get Request* auf den *Root*-Pfad *"/"* der Text „Hello World" ist. Einfacher geht's nicht.

Nancy pudert sich die Nase

Einen "Hello World"-String zu liefern ist schön, aber manchmal will man „komplexere" Daten darstellen; dafür braucht man einen View. Nancy kommt mit einer *Super-Simple-View-Engine*, die ähnlich wie *Razor*, die *Standard-ASP.NET-MVC3-View-Engine* von Microsoft, funktioniert.

Für das jetzige Beispiel legt man einfach eine *index.cshtml*-Datei mit dem folgenden Inhalt an.

```
<!DOCTYPE html>
<html>
<head>
<title></title>
</head>
<body>
<h1>Graffiti</h1>
</body>
</html>
```

Zurück im Modul, halten sich die Änderungen in Grenzen:

```
public GraffitiModule() {
  Get["/"] = _ => {
    return View["index"];
  };
}
```

Nancy, Zeit zum Ausgehen

Weiß man, wie Nancy grob funktioniert, ist es höchste Zeit Simple.Data und MongoDB zu erkunden und unsere Applikation mit einer echten Datenbank zu verbinden. Aber erstmal müssen wir ein paar zusätzliche *Nuget Packages* installieren:

```
Install-Package Simple.Data.MongoDB
Install-Package Nancy.Viewengines.Razor
```

Hinweis: das *Razor Package* braucht nur, wer Razor verwenden will. Man könnte mit der Super-Simple-View-Engine weiterarbeiten, oder auch mit *Spark* [8], *NDjango* [9] oder *dotLiquid* [10].

Es sei noch darauf hingewiesen, dass das Simple.Data-Package hier mit dem MongoDB-Adapter installiert wird. Es könnte auch separat via *Install-Package Simple.Data.Core* installiert werden.

Falls ein anderer Datenbank-Typ verwendet werden soll, bedarf es natürlich eines anderen Adapters.

Zurück im Modul, werden ein paar *private* Felder angelegt, um die Datenbankverbindung aufzubauen:

```
usingSimple.Data;
usingSimple.Data.MongoDB;

public class GraffitiModule : NancyModule {
  private const string ConnectionString =
  @"mongodb://localhost:27017/graffiti";
  private readonly dynamic db =
  Database.Opener.OpenMongo(ConnectionString);
  public GraffitiModule() {
    [...]
  }
}
```

Die „Graffiti"-Datenbank existiert noch nicht. Sie wird aber automatisch angelegt, sobald ein Objekt geschrieben wird. Das war's! Simple.Data ist nun bereit mit einer Datenbank zu sprechen.

Wenn noch nicht vorhanden, muss noch das letzte MongoDB-Package von der offiziellen Web-Seite geladen und installiert werden. Danach kann der Datenbank-*Service* in einem *Command*-Fenster gestartet werden:

```
mongod.exe --dbpath <path to the db>
(z. B. mongod.exe --dbpath C:\data\db)
```

Und in einem anderen *Command*-Fenster startet man die *MongoDB Interactive Shell*:

```
mongo.exe localhost/<dbname>
(z. B. mongo.exe localhost/graffiti)
```

Bedarf es einiger Testdaten, so fügt man ein „Graffiti"-Objekt mit einer einfachen *Text-Property* hinzu. In der MongoDB Interactive Shell tippt man dazu Folgendes.

```
db.Graffiti.save({Text: "Comin' from the DB"});
```

Um zu prüfen ob die Daten richtig gespeichert wurden, kann man die Datenbank *Collection* Statistik abfragen:

```
db.printCollectionStats()
```

Und falls alles gut ging, bekommt man folgendes Ergebnis.

```
system.indexes
{
"ns" : "graffiti.system.indexes",
"count" : 1,
"size" : 72,
"avgObjSize" : 72,
"storageSize" : 4096,
"numExtents" : 1,
"nindexes" : 0,
"lastExtentSize" : 4096,
"paddingFactor" : 1,
"flags" : 0,
"totalIndexSize" : 0,
"indexSizes" : {
},
"ok" : 1
}
```

Dabei ist die Zeile *"count" : 1* am wichtigsten. Alles hat geklappt!

Am Rande sei erwähnt, dass die Hilfe der MongoDB Interactive Shell via *.help*, wie z. B. *db.help* oder *db.graffiti.help*, aufgerufen werden kann.

Es wird ernst

Jetzt, wo das *Setup* vollständig ist, ist man in der Lage, die Daten der Datenbank anzuzeigen. Dafür ändert man die *Get["/"]*-Expression, um die Daten zu holen und dem View ein Model zu reichen.

```
Get["/"] = _ => {
  var scrawls = db.Graffiti.All();
  return View["index", scrawls];
};
```

Ein Simple.Data-Aufruf funktioniert nach folgendem Muster: *Datenbank.Objekt.Methode().Optionen()*. In diesen Fall wird die Datenbank nach allen "Graffiti"-Objekten gefragt, ohne spezifische zusätzliche Informationen (z. B. *OrderBy*). Diese Information wird zunächst an den View gereicht.

Im *cshtml*-View wird der *Body* geändert, um über die Objekt-Kollektion zu iterieren:

```
<!DOCTYPE html>
<html>
<head>
<title></title>
</head>
<body>
<h1>Graffiti</h1>
@foreach(var scrawl in @Model)
{
<p>@scrawl.Text</p>
}
</body>
</html>
```

Damit werden die „Graffiti"-Objekte angezeigt.

Obwohl hier Razor-Syntax verwendet wird, ist die Model-Definition einfacher geworden; leider verliert man damit aber die *Intellisense-Auto-Complete*-Hilfe in den Views.

Gemeinsame Zukunft?

Um Daten hinzuzufügen braucht man noch ein Formular:

```
<!DOCTYPE html>
<html>
<head>
<title></title>
</head>
<body>
<h1>Graffiti</h1>
<form method="POST">
Write something: <input type="text" name="Text"/>
<input type="submit" value="Scrawl!" />
</form>
@foreach(var scrawl in @Model)
{
<p>@scrawl.Text</p>
}
</body>
</html>
```

Dies ist ein Standard-HTML-Formular mit einem *Submit*-Knopf, der eine *POST*-Aktion auslösen wird.

Zurück in unserem Modul müssen wir uns um diese *POST*-Aktion auf *"/"* kümmern:

```
usingNancy.Responses;

Post["/"] = _ =>
{
db.Graffiti.Insert(Text: Request.Form.Text.Value);
return new RedirectResponse("/");
};
```

Dazu sind noch vier Bemerkungen zu machen!

- Das Einfügen von Daten via Simple.Data ist sehr einfach und folgt dem Muster *Datenbank.Objekt.Insert()*.
- Nach Ausführung der *HTTP-POST*-Aktion wird der Benutzer weiter zur *GET["/"]*-Aktion via einem neuen *RedirectResponse("/")*-Objekt geleitet.
- Das Präfix *Text*: wird verwendet, um Simple.Data zu sagen, dass ein *on the fly*-Objekt benutzt werden soll. Ansonsten müsste ein "myGraffiti"-Objekt mit einer Text-Property definiert, die Property ausgefüllt und *db.Graffiti.Insert(myGraffiti)* aufgerufen werden.
- Durch *Request.Form.Text.Value* wird das View-Model repräsentiert.

Damit ist die Applikation vollständig und wir können Text hinzufügen und wieder lesen.

Der Abschiedskuss

Es wurde gezeigt, wie man Nancy, Simple.Data und MongoDB kombinieren, konfigurieren und miteinander verbinden kann. Dabei konnte man sehen, wie Nancy auf eine bestimmte Route reagiert, einen View anzeigt, View-Daten von und zum View schickt. Ebenso wurde gezeigt, wie man Simple.Data verwendet, um Daten zu schreiben bzw. von einer Datenbank zu lesen. Dazu wurde MongoDB gestartet und demonstriert, wie über die *Interactive Shell* Objekte hinzugefügt und gelesen werden können. Das Ganze innerhalb zehn *C#*-Linien und kaum mehr *cshtml*-Linien. Das ist ein erster Blick in den Super-Duper-Happy-Pfad.

Natürlich gibt es viel mehr zu erfahren, aber dafür braucht es einen weiteren Artikel.

Referenzen

[1] NancyFx *Leightweight Web Framework for .NET*, http://nancyfx.org

[2] github *Simple.Data*, https://github.com/markrendle/Simple.Data/wiki

[3] MongoDB *Agile and Scalable* http://www.mongodb.org

[4] Sinatra http://www.sinatrarb.com

[5] Mono http://www.mono-project.com

[6] OWIN *Open Web Interface for .NET* http://owin.org

[7] Nuget gallary *home* http://nuget.org

[8] Spark View Engine *Html friendly. Less is more.* http://sparkviewengine.com

[9] Ndjango http://ndjango.org

[10] DotLiquid *A safe templating system for .NET* http://dotliquidmarkup.org

Weiterführende Literatur

- Håkansson, Andreas *Persönliches Blog* http://thecodejunkie.com/
- Rendle, Mark *Persönliches Blog* http://blog.markrendle.net
- Rendle, Mark *Introduction to Nancy and Simple.Data* http://skillsmatter.com/podcast/open-source-dot-net/introduction-to-nancy-and-simple-data

Kurzbiographie

Timothée Bourguignon ist als Senior Developer für die MATHEMA Software GmbH tätig. Sein Spezialgebiet ist Desktop- und Web-Programmierung mit dem .NET-Framework. Hierbei setzt er auf die Philosophie des Software Craftsmanships. Daneben beschäftigt er sich mit den Neuerungen der .NET-Welt und deren Communities. Als Certified Scrum Master liegt ein weiterer Schwerpunkt auf agilen Methoden.