ANGULARJS UND ASP.NET, TEIL 3

Solides Handwerk

AngularJS und Microsofts Web-Stack ergänzen sich ideal. Bei der Integration von AngularJS in eigene .NET-Anwendung hilft eine Reihe von Entwurfsmustern und Frameworks.

In der ersten Ausgabe dieser Artikelreihe wurde der Modul-Loader RequireJS vorgestellt [1]. Die zweite Folge widmete sich dem OData-Protokoll und dem AJAX-Framework Breeze.js [2]. Doch waren alle vorgestellten Quellcodebeispiele tatsächlich fehlerfrei? Es gilt zu beweisen, dass sowohl der C#- als auch der JavaScript-Code korrekt implementiert wurde. Also müssen Tests die Qualität der Software auf dem Server und im Client sicherstellen können.

Der zweite Teil hat eine Geschäftslogik mit zwei Entitäten eingeführt. Die technische Grundlage bildete das Entity Framework in Version 6 mit dem Code-First-Ansatz. Das Entity Framework ist ein objektrelationaler Mapper (ORM). Es verbindet die objektorientierte .NET-Welt mit einer relationalen Datenbank wie etwa dem SQL Server. Die vom Entity Framework erzeugten Instanzen repräsentierten auch gleichzeitig die Geschäftsobjekte. Die Geschäftslogik bestand aus der Entität *Kunde*, die eine beliebige Anzahl an Rechnungen enthalten konnte, zu sehen in Listing 1. Weiterhin wurde ein ASP.NET-Web-API-Controller verwendet, um eine Liste aller Kunden per REST zur Verfügung zu stellen (siehe Listing 2).

Listing 1: Die Logik der Kundenklasse

```
public class Customer{
  public Customer() {
    Invoices = new List<Invoice>();
  }
  public int Id { get; set; }
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public string Mail { get; set; }
  public DateTime DateOfBirth { get; set; }
  public virtual ICollection<Invoice> Invoices {
    get; set; }
}

public class Invoice {
  public int Id { get; set; }
  public decimal Amount { get; set; }
  public int CustomerId { get; set; }
  public virtual Customer Customer { get; set; }
}

public class DataContext : DbContext {
  public virtual DbSet<Customer> Customers { get; set; }
  public virtual DbSet<Invoice> Invoices { get; set; }
  protected override void OnModelCreating(
      DbModelBuilder modelBuilder) {
    modelBuilder.Configurations.Add(new InvoiceMap());
  }
}
```

Unit-Tests mit dem Entity Framework

CustomersController aus Teil 2 lässt sich so, wie er vorgestellt wurde, nur schwer automatisch testen, denn er ist von der Klasse *DataContext* abhängig. Dadurch lässt sich der Controller nicht mehr losgelöst von allen anderen Komponenten testen. Im konkreten Fall würde das Entity Framework stets versuchen, eine Datenbankverbindung aufzubauen. Solange dies der Fall ist, kann ein Unit-Test nicht implementiert werden. Mittels „Inversion of Control" (IoC) lässt sich der Code schnell korrigieren. Statt eine Instanz von *DataContext* zu erzeugen, wird diese dem Konstruktor übergeben:

```
public class CustomersController :
    ApiController {
  private readonly DataContext db;
  public CustomersController(
    DataContext dataContext) {
      db = dataContext; }
  // ...
}
```

Üblicherweise wird ein vorhandener IoC-Container eingesetzt, der viel Arbeit spart. Der Quelltext auf der Heft-DVD verwendet dazu Autofac [3], das eine komfortable Integration in ASP.NET MVC und ASP.NET Web API bietet (siehe Datei *IocConfig.cs*). Der Controller akzeptiert nun eine beliebige Instanz des Objekts *DataContext*. Weitere Anpassungen sind nicht notwendig, denn erfreulicherweise ist das Entity Framework direkt mit Objekten im Arbeitsspeicher testbar. Für die Version 5 des Entity Frameworks war es noch notwendig, das Objekt mit einem Interface zu maskieren; seit ▶