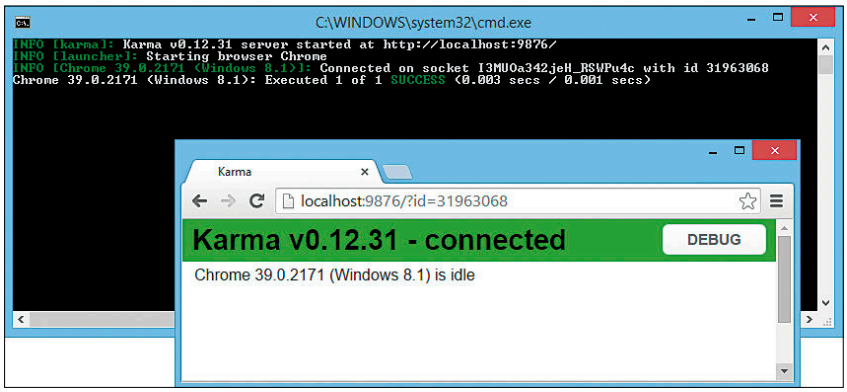
ANGULARJS UND ASP.NET, TEIL 4

Tests in allen Schichten

AngularJS und Microsofts Web-Stack ergänzen sich ideal. Beim Einbinden von AngularJS in .NET-Anwendungen soll auch das Testen nicht zu kurz kommen.

Wenn so verschiedene Komponenten wie der Modul-Loader RequireJS und das AJAX-Framework Breeze.js mit einer .NET-Anwendung zusammenarbeiten sollen, stellt sich auch die Frage, ob sie das fehlerfrei tun. Das lässt sich mit C#- und JavaScript-Unit-Tests bewerkstelligen. Die Vorbereitungen dazu haben die ersten drei Artikel dieser kleinen Serie gelegt, nun stehen die Tests selbst im Mittelpunkt [1, 2, 3].

Kurze Rekapitulation: Im zweiten Teil wurde eine einfache Geschäftslogik mit zwei Entitäten eingeführt. Die Grundlage bildete das Entity Framework 6 mit dem Code-First-Ansatz. Die damit erzeugten Instanzen repräsentieren gleichzeitig die Geschäftsobjekte. Die Geschäftslogik besteht aus der Entität *Kunde*, die eine beliebige Anzahl an Rechnungen enthalten konnte (Listing 1). Ein ASP.NET-Web-API-Controller übernahm es, eine Liste aller Kunden per REST zur Verfügung zu stellen (Listing 2).



Ein erfolgreicher Test mit dem Karma-Testrunner (Bild 1)

Listing 1: Die „Geschäftslogik" 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; }
}
```

Um die Klasse *CustomersController* [2] für (automatische) Unit-Tests zugänglich zu machen, wird ihr nach dem Prinzip der „Inversion of Control" (IoC) eine Instanz von *DataContext* übergeben, siehe Listing 3; das dazu verwendete IoC-Framework war Autofac [4]. Weiterhin kamen für die Tests die Frameworks Machine.Specifications (MSpec) [5], MSpec Test Adapter [6], Fluent Assertions, NSubstitute, Effort, karma-xml-reporter [7], der Karma-Test-Adapter [8] und Jasmine [9] zum Einsatz.

Der Artikel schloss mit einem ersten JavaScript-Unit-Test des AMD-Moduls *helloWorld*; das AMD-Format dieses Tests wird mit dem *define*-Befehl gekennzeichnet (siehe Listing 4), das Ergebnis auf der Kommandozeile ausgegeben. Es öffnet sich ebenso auch ein Browser, der die Entwicklung und die Fehlersuche in einem Test in einer gewohnten Debugging-Umgebung ermöglicht (Bild 1).

Den Vertrag im Client einhalten

Auf dem Server definiert ein Unit-Test, dass die GET-Methode von *CustomerController* entweder mit dem Statuscode 200 oder 404 antwortet. Diese Regel sollte auch der Client berücksichtigen, was idealerweise ein Unit-Test sicherstellt.

AngularJS wird mit der Datei *angular-mocks* ausgeliefert, die das AngularJS-Modul ngMock enthält. Es vereinfacht die Arbeit mit Unit-Tests beträchtlich. Wird ein Test mit einem Test-Framework wie Jasmine ausgeführt, so tauscht es unter anderem den originalen *$httpBackend*-Dienst von AngularJS mit einem Dummy, einem Mock aus. Ebenso können Sie mittels des *module*-Befehls eigene AngularJS-Module für den Unit-Test vorbereiten. Im Beispiel in Listing 5 wird das Modul *example1* vorbereitet und anschließend auf die beiden Testfälle geprüft.