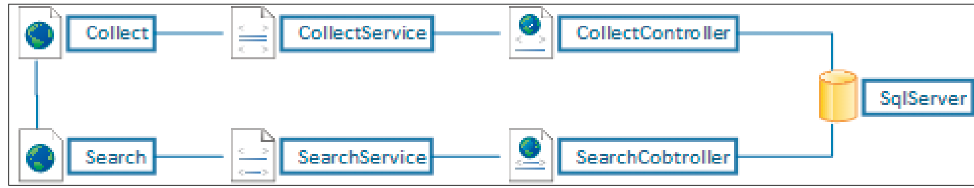
WISSENSDATENBANK MIT ASP.NET CORE UND ANGULAR 2, TEIL 1

Skelett und Haut

Praktisches Beispiel einer Angular-2-App, entwickelt in drei Iterationen.

Angular 2 ist in aller Munde: Die Performance gegenüber der Vorgängerversion ist um einiges besser geworden, und mit dem Kommandozeilen-Interface Angular CLI ist ein praktisches Werkzeug entstanden. Zusammen mit ASP.NET Core lassen sich damit umfangreiche Webapplikationen relativ flott realisieren.



Grobe Übersicht über die Elemente der Applikation (Bild 1)

Als Beispielanwendung, die hier vorgestellt werden soll, ist eine webbasierte Wissensdatenbank geplant, mit der Wissensschnipsel erfasst und in einer Zeitlinie abgelegt werden. Diese Wissensschnipsel können mehrere Arten von Daten enthalten, wie zum Beispiel Freitext, Code, Links, Bilder, Dokumente et cetera. Weiterhin soll eine Volltextsuche das Finden von Wissensschnipseln erleichtern. Die Suche soll sich auch zeitlich eingrenzen oder auf bestimmte Daten beschränken lassen. Die Ausgabe ist immer eine Twitter-ähnliche Zeitleiste, in der die neuesten Einträge ganz oben stehen.

Um möglichst zukunftsfähig zu sein, sollen modernste Technologien eingesetzt werden, mit denen sowohl Backend als auch Frontend auf möglichst vielen Plattformen lauffähig sind. Dabei ist die Entscheidung auf Angular 2 im Frontend und ASP.NET Core 1.0 im Backend gefallen. Mit dem ASP.NET Core Stack werden lediglich die REST-APIs erstellt. Razor Views sind nicht vorgesehen. Für die Views ist ausschließlich Angular 2 zuständig. Der Auftraggeber hat sich für eCollector als Produktname entschieden.

Die Artikelserie, deren ersten Teil Sie gerade lesen, ist in drei Teile aufgeteilt, die sich jeweils unterschiedlichen Aspekten widmen. Es ist also ein Release in drei Iterationen geplant. In der ersten Iteration sind folgende Themen vorgesehen:

- Architektur, Entwurf der Oberfläche, Projekt-Setup.
- Angular-2-App, Basis-UI, Masken für Datenerfassung und -ausgabe.
- Eingabemaske für die Suche und Ausgabe der Ergebnisse.

Die weiteren Iterationen befassen sich dann mit der Serverseite (dem Backend), dem Anbinden des Frontends an das Backend sowie mit dem Deployment der Applikation.

Die Anforderungen

Es soll eine Single-Page-Applikation (kurz: SPA) erstellt werden. Das heißt, im Frontend findet alles auf einer einzigen HTML-Seite statt, die auch nicht neu geladen werden muss. Die Applikation wird im Frontend komplett mit JavaScript umgesetzt. Angular 2 ist dafür bestens geeignet und bietet dank Templating und Routing eine sehr einfache Möglichkeit, Views zu wechseln und Deep-Links in die Applikation zu setzen. Das heißt, einzelne Views sind über einen URL beziehungsweise eine Route direkt ansteuerbar. Außerdem wird TypeScript verwendet, das die Arbeit mit Angular 2 enorm erleichtert.

Die Angular-2-Applikation wird über ein ASP.NET-Core-Web-API per JSON-basierter HTTP-Schnittstelle mit dem Backend kommunizieren. Mehr zu ASP.NET Core finden Sie im Kasten ASP.NET Core.

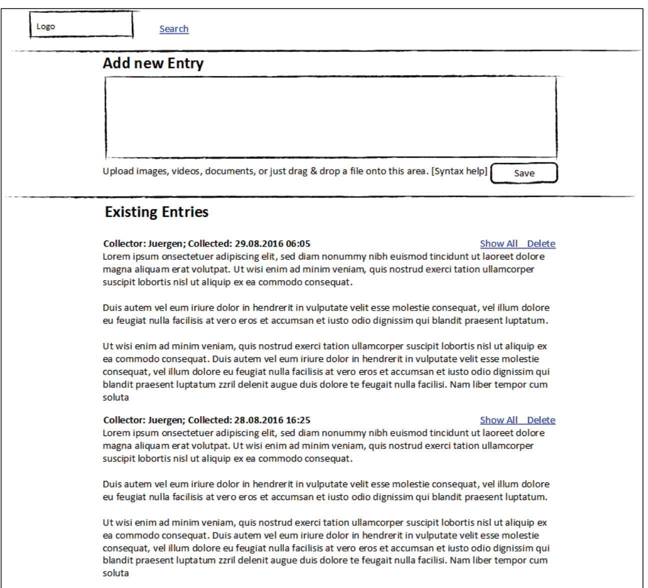
Diese Applikation wäre perfekt dafür geeignet, eine Dokumentendatenbank (zum Beispiel Azure DocumentDB) sowie eine Elastic-Search-Implementation (zum Beispiel den Azure Search Service) zu nutzen.

Der Einfachheit halber – und da es im Moment nicht gefordert ist – werden die Daten im SQL Server und die Binärdaten im Dateisystem abgelegt. Auch die Volltextsuche wird im SQL Server umgesetzt (Bild 1).

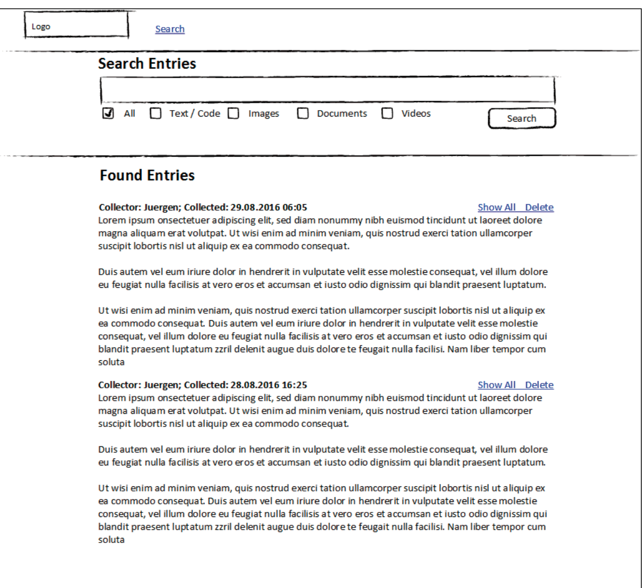
Das User Interface (UI) wird so einfach wie möglich gestaltet. Analog zu Twitter wird es die Eingabemaske zuoberst darstellen, mit weiteren Optionen zum Erfassen von Bildern, Videos und Dokumenten. Text und Code werden per Mark-

ASP.NET Core

ASP.NET Core ist ein Web-Framework, das von Microsoft als Open Source komplett neu geschrieben wurde. ASP.NET Core basiert auf .NET Core, kann aber auch mit dem .NET Framework 4.6 betrieben werden. Zum Zeitpunkt, als der Artikel geschrieben wurde, war ASP.NET Core bereits als RTM in der Version 1.0 verfügbar. Allerdings standen Tooling und SDK vorerst nur als Preview 2 zur Verfügung. Die RTM-Versionen sollen zusammen mit der nächsten Visual-Studio-Version zum Ende des Jahres 2016 erscheinen. Mehr zum Thema ASP.NET Core ist unter http://get.asp.net zu finden.



Mockup für Eingabeformular und Timeline (Bild 2)



Mockup für Suchformular und Ausgabe der Treffer (Bild 3)

down erfasst (siehe Bild 2). Hyperlinks sollen automatisch erkannt und anklickbar ausgegeben werden. Zur Darstellung von Sourcecode wird *prism.js* verwendet. YouTube-Links sollen automatisch als Videos eingebunden werden, Dokumente und Bilder sollen über einen klassischen Dateiauswahldialog und optional auch per Drag-and-drop hinzugefügt werden können.

Die Suche (Bild 3) besteht aus einem einzeiligen Feld mit daneben platziertem Suchknopf. Felder zur zeitlichen Einschränkung sowie zur Beschränkung auf einen oder mehrere spezielle Wissenstypen sollen bei Bedarf genutzt werden können. Die Ausgabe der Suchergebnisse erfolgt auf die gleiche Art wie bei der Timeline.

Sowohl die Timeline als auch die Suchausgabe sollen über ein „unendliches“ Scrolling angeschaut werden können.

Arbeiten mit Angular CLI

Die Web-Frontend-Entwicklung ist – abgesehen vom Editor – sehr konsolenlastig. Für viele ASP.NET-Entwickler ist das sicher eine kleine Herausforderung. Neben Node.js, NPM, Gulp und Co. kommt nun auch noch Angular CLI hinzu. Bei der Arbeit mit diesem Toolset hat sich herausgestellt, dass es sinnvoll ist, mindestens zwei Konsolen geöffnet zu haben. In der einen laufen sowohl der mit *ng serve* gestartete Server als auch ein File-Watcher, der den TypeScript-Compiler auslöst (Bild 4). In der zweiten Konsole wird gearbeitet, sprich: Es werden mit Angular CLI neue Dateien erstellt, per NPM neue Pakete geladen et cetera. Wenn man nun eine Datei ändert, wird bei Bedarf kompiliert und der Browser automatisch aktualisiert. So ist das Arbeiten mit Angular 2 und TypeScript eine bequeme Sache. Sogar bequemer, als es ein ASP.NET-Entwickler mit Visual Studio gewohnt ist.

Setup der Entwicklungsumgebung

Die Anforderungen sind nun geklärt. Um mit der Entwicklung starten zu können, muss das Entwicklungsprojekt zunächst aufgesetzt werden. Die Entscheidung fällt auf Visual Studio 2015 als IDE im Backend und Visual Studio Code (VS Code) als IDE für die Frontend-Entwicklung. Jede andere IDE wäre für das Frontend ebenso geeignet. Andererseits wäre es möglich, VS Code auch für das Backend zu verwenden. Allerdings ist die C#-Unterstützung durch Visual Studio etwas besser.

Die Entwicklung von Frontend und Backend wird physikalisch getrennt, weil Visual Studio ein paar Probleme bei der Entwicklung mit Angular 2 hat. Zwar funktioniert die Entwicklung mit TypeScript und JavaScript prinzipiell sehr gut, allerdings gibt es immer wieder drastische Performance-Probleme durch die vielen automatisch generierten Dateien bei aktiver Sourcecode-Verwaltung, sodass es auf Dauer keinen Spaß machen wird, damit zu arbeiten. TypeScript generiert sehr viele JavaScript- und Map-Dateien. Der Node Packet Manager (NPM) holt Unmengen an Daten aus dem Netz. Bundling- und Minifying-Tools legen weitere Dateien dynamisch im Projektordner ab.

Ein weiterer Vorteil der Trennung ist, dass man nicht die komplette .NET-Maschinerie starten muss, um mit dem Client zu arbeiten. So schnell das mit .NET Core auch gehen mag, die Kompilierungszeit ist dennoch spürbar. Ebenso ist es andersherum: Um am HTTP-Dienst zu arbeiten, muss nicht unbedingt das UI starten. Es genügt, das Web-API-Projekt zu starten und das API mit Tools wie Fiddler oder Postman auszuprobieren.

Zur Sourcecode-Verwaltung fiel die Entscheidung zugunsten von Git, das auf GitHub gehostet wird. Unter [1] finden Sie den aktuellen Stand der Sourcen.

Wie zu Beginn geschrieben, wird in dieser ersten Iteration nur das Frontend angelegt. Das Setup des Frontends kann etwas umfangreicher werden, aber auch hier gibt es Werk- ▶

zeuge, die uns die meiste Arbeit abnehmen. Am nützlichsten erscheint dafür Angular CLI [2]. Dabei handelt es sich um ein Command-Line-Toolset, mit dem eine Angular-2-Applikation initial aufgesetzt werden kann, das aber auch während der Entwicklung beim Erstellen von Angular-Komponenten hilft. Auch eine Test-Suite sowie ein einfacher Server werden mitgeliefert. Zudem ist TypeScript bereits integriert und komplett konfiguriert (siehe auch Kasten Arbeiten mit Angular CLI).

Voraussetzung für Angular CLI sind Node.js und NPM auf der Entwicklungsmaschine. Angular CLI selbst wird am besten global per NPM installiert:

```
npm install angular-cli -g
```

De folgende Aufruf erstellt eine neue, fertig konfigurierte Angular-2-Applikation im aktuellen Ordner:

```
ng int
```

Optional kann ein Name angegeben werden. Dieser Vorgang kann eine Weile dauern, da der Paketmanager NPM zuvor alle erforderlichen Abhängigkeiten lädt und konfiguriert.

Sobald das Projekt fertig aufgesetzt ist, kann die App mit dem Kommando *ng serve* (siehe Bild 4) gestartet und das Projekt im Browser getestet werden. Man erhält dafür in der Konsole einen URL ausgegeben. Auf diese Art erfährt man sofort, ob alles funktioniert oder nicht. Somit ist das Client-Projekt aufgesetzt, ohne dass es manuell konfiguriert werden musste.

Das nötige Setup für die clientseitige Applikation ist nun fertig. Wir kennen die Anforderungen; die Architektur und die ersten Entwürfe für das UI liegen vor. Es steht nun nichts mehr im Weg, mit der eigentlichen Entwicklung zu starten.

Setup der Applikation

Der Einfachheit halber soll das User Interface mit Bootstrap gestaltet werden. Die Standardstile können später noch angepasst werden oder mithilfe eines speziellen Bootstrap-Templates ausgetauscht werden. Dafür muss Bootstrap im Client-Projekt noch in den Abschnitt *dependencies* der Datei *package.json* aufgenommen werden:

```
C:\git\dnp-e-collector\client (master) (client@0.0.0)
λ ng serve
** NG Live Development Server is running on http://localhost:4200. **
12075ms building modules
48ms sealing
4ms optimizing
0ms basic module optimization
255ms module optimization
1ms advanced module optimization
12ms basic chunk optimization
0ms chunk optimization
4ms advanced chunk optimization
0ms module and chunk tree optimization
104ms module reviving
16ms module order optimization
4ms module id optimization
8ms chunk reviving
0ms chunk order optimization
20ms chunk id optimization
134ms hashing
2ms module assets processing
260ms chunk assets processing
```

Das Angular-CLI-Kommando ng serve in der Konsole (Bild 4)

Listing 1: index.html

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>dotnetpro eCollector</title>
    <base href="/">
    <meta name="viewport" content=
      "width=device-width, initial-scale=1">
    <link rel="icon" type="image/x-icon"
      href="favicon.ico">
  </head>
  <body>
    <app-root>Loading...</app-root>
  </body>
</html>
```

```
"bootstrap": "^3.3.6"
```

Ein Aufruf von *npm install* in der Konsole installiert Bootstrap. Bootstrap benötigt jQuery, allerdings ist jQuery nicht wirklich kompatibel zu Angular 2. Daher werden einige Bootstrap-Komponenten nicht sauber funktionieren. Das liegt daran, dass man jQuery-Elemente initialisiert, wenn das Dokument fertig aufgebaut ist (*onDocumentReady*). Angular 2 fängt dann aber erst an, das UI zu generieren. jQuery kann demnach noch nicht auf das komplette DOM zugreifen. Außerdem sollte man das von Angular 2 generierte DOM nicht manipulieren, damit es mit dem „virtual DOM" übereinstimmt, das Angular 2 nutzt, um schneller zu rendern. Hier wird daher von Bootstrap lediglich CSS verwendet. jQuery wird also nicht eingebunden und genutzt.

Um Bootstrap in die Applikation einzubinden, wird es direkt aus dem Ordner *node_modules* in die bestehende CSS-Datei importiert:

```
@import url(
  "../node_modules/bootstrap/dist/css/bootstrap.css");
```

Die Basis der Angular-2-App bildet die Datei *index.html*, die im Ordner *src* des Client-Projekts liegt, siehe Listing 1.

Schaut man sich die Datei *index.html* genauer an, ist zu sehen, dass hier keine CSS- oder JavaScript-Datei eingebunden wird. Diese werden von Angular CLI, genauer von Webpack, in die Datei *index.html* eingebunden. Das aktuelle Angular CLI basiert auf Webpack. Webpack wiederum ist ein Werkzeug, mit dem verschiedene Arten von Abhängigkeiten zusammengefasst und auf einer Website eingebunden werden können. Die Konfiguration von Webpack ist recht komplex, allerdings ist in Angular CLI alles automatisiert und gekapselt und beschränkt sich mehr oder weniger auf eine Datei *angular-cli.json*, die man in den meisten Fällen aber nicht anfassen muss. Webpack hat einen entscheidenden Vorteil:

Listing 2: app.component.ts

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: 'app.component.html',
  styleUrls: ['app.component.css'],
})
export class AppComponent implements OnInit {
  constructor() {}

  ngOnInit() {
  }
}
```

Listing 3: app.component.html

```
<nav class="navbar navbar-default">
  <div class="container">
    <div class="navbar-header">
      <button type="button" class=
        "navbar-toggle collapsed"
        data-toggle="collapse" data-target=
        "#bs-example-navbar-collapse-1"
        aria-expanded="false">
        <span class="sr-only">Toggle navigation
        </span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <script src="app.js"></script>
      <a class="navbar-brand" [routerLink]=
        "['/collect']">
        dotnet<i>pro</i> eCollector</a>
    </div>

    <div class="collapse navbar-collapse"
      id="bs-example-navbar-collapse-1">
      <ul class="nav navbar-nav">
        <li><a [routerLink]="['/search']">Search
        </a></li>
      </ul>
    </div>
  </div>
</nav>

<div class="container">
  <router-outlet></router-outlet>
</div>
```

Es sind keine Werkzeuge wie Gulp mehr nötig, um die Abhängigkeiten aus *node_modules* von NPM in das Webverzeichnis zu holen. Webpack ermittelt alle Abhängigkeiten rekursiv und packt sie in eine einzige Datei, die im richtigen Verzeichnis abgelegt wird. Am Beispiel von CSS sieht das so aus, dass sogar alle Icons und Schriften ermittelt, gegebenenfalls gepackt und in das Ausgabeverzeichnis gelegt werden.

Es muss also nichts weiter konfiguriert werden. Alles Weitere wird so belassen, wie es ist. Darum kümmert sich Angular CLI. Auch alle weiteren UI-Elemente werden nicht in der *index.html* angelegt, sondern über Templates der Angular-2-Komponenten eingebunden, da sie spezielle Funktionalität beinhalten, und sei es nur die Navigation zwischen Views.

Somit haben wir alle Voraussetzungen geschaffen, um eine Angular-2-App zu hosten und mit der Entwicklung der eigentlichen App zu starten.

Die Entwicklung kann beginnen

Im Ordner */src/app/* liegt schon die erste Angular-2-Komponente. Diese Datei hat üblicherweise den Namen *app.component.ts*. Der Code ist noch frei von Logik und enthält lediglich einen Verweis auf ein Template sowie eine Definition eines Selektors mit dem Namen *app-root* (Listing 2).

Die AppComponent ist üblicherweise die Startkomponente einer App. Da alle Komponenten in Angular 2 hierarchisch aufeinander aufbauen, bildet sie die Root-Komponente, ist mithin der Einstiegspunkt der App.

Die *app.component.ts* wird nicht weiter angepasst. Das zugehörige Template (Listing 3) allerdings schon, es enthält das Grundlayout für diese Applikation.

Im Template ist das Routing zu erkennen, mit dem zwischen verschiedenen Views gewechselt wird. Zudem finden Sie dort eine Direktive mit dem Namen *router-outlet*. Hier werden die Ergebnisse des Routings gerendert. Das ist praktisch ein Platzhalter für alle weiteren Komponenten, die per Routing geladen werden.

Der Router wird in einer weiteren Datei mit dem Namen *app-routing.module.ts* konfiguriert (Listing 4).

Wie aber wird das alles zusammengesteckt? Das AppRoutingModule und die AppComponent müssen eingangs natürlich bekannt gemacht und aktiviert werden.

Den Anfang macht die Datei *main.ts* (Listing 5), die im *src*-Verzeichnis liegt. Das ist der Einstiegspunkt der App. Hier wird das Hauptmodul geladen, in diesem Fall die *app.modules.ts* aus dem *src/app*-Verzeichnis.

Im AppModule wiederum wird alles zusammengesteckt. Alle Komponenten und Services werden dort registriert und dem System bekannt gemacht. Das muss nicht manuell erfolgen. Verwendet man das aktuelle Angular CLI, wird das automatisch erledigt, sofern man neue Komponenten und Services generiert, siehe Listing 6. In diesem Modul sind auch die AppComponent und das AppRoutingModule zu finden.

Die ersten Komponenten

In der *app.routes.ts* werden zwei Routen definiert und auf die CollectComponent und auf die SearchComponent gemappt. Beide wurden zuvor mit Angular CLI angelegt: ▶

Listing 4: app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from
       '@angular/router';

import { CollectComponent } from
       './collect/collect.component';
import { SearchComponent } from
       './search/search.component';

const routes: Routes = [
  { path: '', redirectTo: '/collect',
    pathMatch: 'full' },
  { path: 'collect', component: CollectComponent },
  { path: 'search', component: SearchComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})

export class AppRoutingModule { }
```

Listing 5: main.ts

```
import './polyfills.ts';
import { platformBrowserDynamic } from
       '@angular/platform-browser-dynamic';

import { enableProdMode } from '@angular/core';
import { environment } from
       './environments/environment';
import { AppModule } from './app/';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Listing 6: app.module.ts

```
import { BrowserModule } from
       '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';

import { AppRoutingModule } from
       './app-routing.module';

import { AppComponent } from './app.component';
import { CollectComponent } from
       './collect/collect.component';
import { SearchComponent } from
       './search/search.component';

import { SearchService } from
       './search/shared/search.service';
import { CollectService } from
       './collect/shared/collect.service';
import { UploadService } from
       './collect/shared/upload.service';
@NgModule({
  declarations: [
    AppComponent,
    CollectComponent,
    SearchComponent,
  ],

  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    AppRoutingModule
  ],
  providers: [
    CollectService,
    SearchService,
    UploadService
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Listing 7: Template der CollectComponent

```
<div class="row" style="padding-bottom: 32px;">
  <div class="col-md-6 col-md-offset-3">
    <h3>Add new entry</h3>
    <app-knowledge-collector>
    </app-knowledge-collector>
  </div>
</div>
<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <h3>Existing entries:</h3>
    <app-knowledge-timeline>
    </app-knowledge-timeline>
  </div>
</div>
```

Listing 8: KnowledgeCollectorComponent

```
import { Component, OnInit } from '@angular/core';

import { CollectService } from '../collect.service';
import { ICollectModel } from '../icollect-model';

@Component({
  selector: 'app-knowledge-collector',
  templateUrl: 'knowledge-collector.component.html',
  styleUrls: ['knowledge-collector.component.css'],
})
export class KnowledgeCollectorComponent
  implements OnInit
{
  constructor(private _collectService:
    CollectService) { }
  ngOnInit() { }

  public text: string = '';
  public files: File[];
  public fileSelector: any;

  public save() {
    let model: ICollectModel = {
      text: this.text,
      files: this.files
    }
    this._collectService.Collect(model)
      .then(success => {
        this.text = '';
        this.files = [];
        if (this.fileSelector) {
          this.fileSelector.form.reset();
        }
      });
  }

  public onChange(event) {
    this.fileSelector = event.srcElement
    this.files = this.fileSelector.files;
    console.log(this.files);
  }
}
```

```
> ng g component collect
> ng g component search
```

Diese Kommandos legen im Ordner *app* je einen Unterordner mit dem entsprechenden Namen an, in dem sich dann die Templates, eine CSS-Datei und zwei TypeScript-Dateien befinden. Eine dieser Dateien ist die Angular-2-Komponente und die andere enthält die Unit-Tests für eben diese Komponente. Die neu erstellten Komponenten nutzen je zwei Direktiven, die in weiteren Unterkomponenten definiert sind. Diese werden dann in den jeweiligen Templates genutzt. Die Templates sehen beide ähnlich aus. Sie enthalten zwei Bootstrap-Zeilen mit je sechs Spalten. In jedem Bereich wird dann eine weitere Unterkomponente angezeigt. Das erfolgt über die Selektoren der jeweiligen Unterkomponenten (Listing 7).

Die CollectComponent sieht im Detail wie folgt aus: In der oberen Zeile ist es die KnowledgeCollectorComponent über den Selektor *<app-knowledge-collector>*, und im unteren Bereich wird eine KnowledgeTimelineComponent über den Selektor *<app-knowledge-timeline>* angelegt. Um diese Komponenten und einen zusätzlichen Service zu erzeugen, wird wieder Angular CLI zu Hilfe genommen:

```
> ng g component collect/shared/knowledge-collector
> ng g component collect/shared/knowledge-timeline
> ng g service collect/shared/collect
```

Die KnowledgeCollectorComponent definiert über das Template die Eingabemaske und enthält die Logik, um die Daten entgegenzunehmen, gegebenenfalls zu validieren und über eine Service-Komponente mit dem Namen *CollectService* an den Server zu senden. Dabei wird der CollectService über Dependency Injection in die Komponente eingebunden, vergleiche Listing 8. ▶

Listing 9: GET-Request an den Server

```
private _service: string = 'http://localhost:5000/';
  LoadTimeline(page: ITimelineRequestModel):
  Promise<ITimelineModel> {
    let headers = new Headers({
      'Content-Type': 'application/json',
      'Page-Number': page.pageNumber,
      'Page-Size': page.pageSize
    });
    let options = new RequestOptions(
      { headers: headers });
    return this._http.get(
      '${this._service}api/timeline', options)
      .toPromise()
      .then(this.extractData)
      .catch(this.handleError);
  }
```

Listing 10: knowledge-timeline.component.html

```
<div class="panel-heading">
  <p>
    Collector: <a href="#">{{item.user}}</a>;
    collected: {{item.date | date:longDate}}
    <button class="btn btn-link pull-right">
      <span class="glyphicon glyphicon-trash">
      </span>Delete
    </button>
    <button class="btn btn-link pull-right">
      <span class="glyphicon glyphicon-zoom-in">
      </span>Show all
    </button>
  </p>
</div>
<div class="panel-body">
  {{item.content}}
  <hr *ngIf="item.documents.length > 0">
  <div *ngIf="item.documents.length > 0">
    <div *ngIf="item.documents[0].type==='Video'">
      <video [attr.src]="item.documents[0].src"
        controls></video>
    </div>
    <div *ngIf="item.documents[0].type==='Audio'">
      <audio [attr.src]="item.documents[0].src"
        width="100%" controls></audio>
    </div>
    <div *ngIf="item.documents[0].type==='Image'">
      <div class="fullSize">
        <a href="bigsrc" data-lightbox="">
          <img [attr.src]="item.documents[0].src"
            [attr.alt]="item.documents[0].name" />
        </a>
      </div>
    </div>
    <hr *ngIf="item.documents.length > 1">
    <ul *ngIf="item.documents.length > 1">
      <li *ngFor="let document of item.documents |
        slice:1"><a [href]="document.src">
        {{document.name}}</a></li>
    </ul>
  </div>
</div>
```

Anfragen an den Server erfolgen asynchron. Daher liefert der Service ein sogenanntes *Promise*-Objekt, auf das man in der Methode *then* reagieren muss. Dafür wird ein Lambda-Ausdruck angegeben, der den Erfolgsfall behandelt. Wie die Anfrage an den Service aussieht, sehen Sie in Listing 9 – den gesamten Service finden Sie auf der Heft-CD oder im Git Repository [1].

Für die Timeline wurde mithilfe von Angular CLI bereits eine weitere leere Komponente gebaut, die nun mit Leben gefüllt werden muss. Zuerst wird der CollectService eingebunden. Über diesen Service werden die bisher eingetragenen Wissensschnipsel vom Server geholt.

Da es verschiedene Typen von Wissensschnipseln gibt, ist die View je nach Typ ein wenig anders zu gestalten. Text und Code-Snippets werden immer auf die gleiche Art angezeigt. Werden zusätzliche Bilder, Videos, Audios und Dokumente eingebunden, so werden diese unter dem Text als Liste dargestellt. Das erste Bild, Video- oder Audio-Element wird unterhalb des Textes direkt angezeigt. Die Video- beziehungsweise Audio-Daten werden dann in einen HTML5-Player eingebunden. Je nach Typ werden bestimmte Elemente ein- oder ausgeblendet. Listing 10 zeigt, wie ein Wissenselement mit Angular 2 in die View aufgenommen wird. Ein wirklich praktisches Angular-2-Feature ist **ngIf*, mit dem Elemente ein- und ausgeblendet werden können, wenn bestimmte Bedingungen erfüllt sind. Die folgende Zeile zeigt ein Element nur dann an, wenn sein Typ dem String *video* entspricht:

```
*ngIf="item.documents[0].type ==='video'"
```

In Listing 10 sehen Sie das gesamte Template für einen Eintrag in die Timeline.

Das *infinity*-Scroll für die Timeline wird hinzugefügt, indem ähnlich vorgegangen wird wie beim klassischem Paging. Wir laden beispielsweise nur 15 Einträge und merken uns eine Seitenzahl (Listing 11). Jedes Mal, wenn wir am unteren Seitenrand angekommen sind, wird die Seitenzahl erhöht, die nächste Seite vom Server geladen und an die aktuelle Liste angehängt. Die Implementierung von *onScroll* ist etwas komplexer, da die Scroll-, Document- und Window-Dimensionen für alle Browser ausgelesen werden müssen. Leider liefern hier die Browser unterschiedliche APIs.

Die aktuelle Seitengröße und Seitenzahl werden von der Komponente an den Service übergeben. Der Service gibt die Daten dann per HTTP-Header an den Server weiter (siehe Listing 9). Beim Nachladen der Daten muss man das darauf folgende Nachladen für eine halbe Sekunde blockieren. Die App benötigt etwas länger, um die Daten zu rendern, und die Scroll-Position würde sonst falsch berechnet. Das würde dazu führen, dass unnötig viele Daten auf einmal nachgeladen werden. Deshalb wird hier das weitere Nachladen unterbrochen. Der Browser hat Zeit, die Daten zu rendern, die Seitenhöhe wird neu berechnet und die Bedingung zum Nachladen der nächsten Seite ist nicht mehr erfüllt.

Die Methode *onScroll* wird über ein *window*-Event angestoßen, das im Template registriert werden muss:

```
(window:scroll)="onScroll($event)"
```

Binärdaten senden

Das Senden der Daten ist komplexer, wenn – wie in unserer App – Dateien mitgesendet werden müssen. Hier reicht die Methode *http.post* nicht mehr aus. Es muss auf das vom Browser zur Verfügung gestellte Objekt *XmlHttpRequest* zurückgegriffen werden. Dieses Objekt erlaubt es, Dateien und einfache Daten in Form einer *FormData*-Liste an den Server zu senden. Dafür wurde ein UploadService geschrieben, der im bestehenden CollectService genutzt wird – den Code dafür finden Sie auf der Heft-CD oder im GitHub Repository. Die aktuellen Methoden für *POST* und *PUT* bieten leider keine Möglichkeit, Binärdaten mitzuliefern. ▶

Listing 11: KnowledgeTimelineComponent

```
export class KnowledgeTimelineComponent
  implements OnInit
{
  constructor(private _collectService:
    CollectService) { }

  timeline: ITimelineModel = {
    items: [],
    pageNumber: 0,
    pageSize: 15,
    overallLength: 0
  };

  ngOnInit() {
    this.loadTimeline(false);
    this._collectService.TimelineUpdated
      .subscribe(item => this.loadTimeline(false));
  }

  private loadTimeline(nextPage: boolean) {
    let timelineRequest: ITimelineRequestModel =
    {
      pageNumber: nextPage ?
      this.timeline.pageNumber : 0,
      pageSize: this.timeline.pageSize,
    };

    this._collectService.LoadTimeline(
      timelineRequest).then(
      data => {
        this.timeline.items =
          nextPage ? this.timeline.items.concat(
          data.items) : data.items;
        this.timeline.pageNumber = data.pageNumber + 1;
        this.timeline.overallLength =
          data.overallLength;
      });
  }

  private _loaded: boolean = false;
  onScroll(event) {
    let body = document.body,
      html = document.documentElement;

    let browserHeight = Math.max(
      document.documentElement.clientHeight,
      window.innerHeight || 0)
    var documentHeight = Math.max(body.scrollHeight,
      body.offsetHeight, html.clientHeight,
      html.scrollHeight, html.offsetHeight);

    if (!this._loaded && Math.round(window.scrollY) >=
      (documentHeight - browserHeight - 10))
    {
      this.loadTimeline(true);
      this._loaded = true;
      window.setTimeout(() => {
        this._loaded = false;
      }, 500);
    }
  }
}
```

Die Suche

Anders als in der Collect-Komponente, bei der die beiden Unterkomponenten annähernd unabhängig voneinander funktionieren, müssen die Unterkomponenten der Search-Komponente zusammenarbeiten. Wird im Suchformular auf den Button *Search* geklickt, muss die Suche auf dem Server ausgelöst und die Timeline mit Daten gefüttert werden. Daher ist es sinnvoll, dass diese Unterkomponenten keine wirkliche Logik enthalten, sondern nur Daten an die übergeordnete Search-Komponente liefern beziehungsweise diese Daten an die Timeline-Komponente weitergeben. Die Unterkomponenten und der SearchService werden ebenfalls mit Angular CLI erzeugt.

```
> ng g component search/shared/search-form
> ng g component search/shared/search-timeline
> ng g service search/shared/search
```

Das Suchformular wird in der Komponente SearchFormComponent – wie im Entwurf beschrieben – angelegt und der Such-Button wird an die Methode *search* gebunden. Nun muss noch ein Event definiert werden, an den sich die übergeordnete Komponente binden kann. Das erfolgt per *Event-Emitter* und einer *Output*-Annotation (Listing 12). So kann die SearchComponent das Event *search* abonnieren, erhält als Argument den Suchbegriff und kann die Suche ausführen.

```
<app-search-form (search)="onSearch($event)">
  </app-search-form>
```

In der SearchTimelineComponent wird stattdessen mit einer *Input*-Annotation eine Eigenschaft ausgezeichnet, die von außen einen Wert entgegennehmen kann:

```
@Input() timeline: ITimelineModel;
```

Die Eigenschaft wird ganz normal an die *timeline*-Eigenschaft der übergeordneten SearchComponent gebunden:

```
<app-search-timeline [timeline]="timeline">
  </app-search-timeline>
```

Nun kann die SearchComponent den SearchService aufrufen, die Suche auslösen und die Timeline mit Daten füllen, siehe Listing 13. Das unendliche Scrolling ist analog zur CollectTimeline implementiert.

Im SearchService wird ein Objekt erstellt, das als *body* an den Server gesendet wird (Listing 14). Hier sind die Daten für das Paging beziehungsweise das unendliche Scrolling enthalten, und dazu die relevanten Informationen für die Suche: Suchbegriff, Suchzeitraum und die zu suchenden Typen. Diese Daten werden per *HTTP POST* an den URL *api/search* gesendet. Das Resultat wird dann an die SearchTimelineComponent übergeben und in der Timeline dargestellt.

Listing 12: EventEmitter

```
@Output() search: EventEmitter<string> =
  new EventEmitter<string>();
public searchTerm: string = '';

doSearch(){
  console.log('search clicked')
  this.search.next(this.searchTerm);
}
```

Listing 13: Suche in der search.component.ts

```
onSearch(event: ISearchModel) {
  this.timeline = {
    items: [],
    pageNumber: 0,
    pageSize: 15,
    overallLength: 0
  };
  console.log('search for:', event);
  this.searchModel = event;
  if (this.searchModel) {
    this.loadTimeline(false);
  }
}

private loadTimeline(nextPage: boolean) {
  let searchRequest: ISearchModel = {
    text: this.searchModel.text,
    dayRange: this.searchModel.dayRange,
    types: this.searchModel.types,
    pageNumber: nextPage ?
    this.timeline.pageNumber: 0,
    pageSize: this.timeline.pageSize,
  };

  this._searchService.search(searchRequest)
    .then(data => {
      this.timeline.items = nextPage ?
        this.timeline.items.concat(
        data.items) : data.items;
      this.timeline.pageNumber = data.pageNumber + 1;
      this.timeline.overallLength =
        data.overallLength;
    });
}
```

Listing 14: Suchanfrage an den Server senden

```
search(searchModel: ISearchModel):
Promise<ITimelineModel> {
  let headers = new Headers({
    'Content-Type': 'application/json'
  });
  let body = {
    'SearchTerm': searchModel.text,
    'Types': searchModel.types.join(),
    'DayRange': searchModel.dayRange,
    'PageNumber': searchModel.pageNumber,
    'PageSize': searchModel.pageSize
  }
  let options = new RequestOptions(
    { headers: headers });

  return this._http.post(
    '${this._service}api/search', body, options)
    .toPromise()
    .then(this.extractData)
    .catch(this.handleError);
}
```

Ausblick

Die Clientseite ist nun funktionsfähig. Da die Serverseite erst in der nächsten Iteration fertiggestellt wird, wurde in den Services mit Dummy-Daten gearbeitet, die in JSON-Dateien hinterlegt waren. Dazu wurden ganz einfach die URLs in den Serviceklassen abgeändert.

In der nächsten Iteration wird es um die Serverseite mit ASP.NET Core 1.0 gehen, um die HTTP-Dienste bereitzustellen, welche die Daten entgegennehmen und Ergebnisse zurückgeben. In der ersten Iteration wurde definiert, wie die Daten auszusehen haben. Der Server muss nun genau diese Daten in den geforderten Strukturen liefern. In der nächsten Folge dieser Serie erfahren Sie, wie die Serverseite aufgesetzt und entwickelt wird. ■

[1] *Projektseite der Wissensdatenbank auf GitHub, www.dotnetpro.de/SL1701ASPNETCore1*

[2] *Kommandozeilen-Interface Angular CLI, https://cli.angular.io*

Jürgen Gutsch
ist Software Developer, Berater, Trainer und freier Autor. Er betreibt einen Blog auf http://asp.net-hacker.rocks, engagiert sich in der .NET-Community und wurde mehrfach als MVP ausgezeichnet. Sie erreichen ihn unter **juergen@gutsch-online.de**.

dnpCode A1701ASPNETCore

dotnetpro Newsletter

Top-Informationen für den .NET-Entwickler.

Klicken. Lesen. Mitreden.

Sehr geehrter Herr Börner,

Now that we're doomed: Seit der Quellcode des .NET Framework Open Source ist, durchforsten ihn Entwickler aus unterschiedlichsten Gründen. Manche finden in den tausenden Zeilen Code skurrile Dinge, die sie dann zum Besten geben.

mehr ...

Sie haben Performance-Probleme mit .NET-Code? Wir wissen nicht, was ein x-beliebiger Berater vorschlagen würde. Wir raten Ihnen zum ANTS Performance Profiler.

mehr ...

Tilman Börner
Chefredakteur dotnetpro

Teilen Sie den Newsletter mit anderen

Die Performance von .NET Anwendungen messen
Der ANTS Performance Profiler analysiert .NET-Anwendungen und informiert über die Code-Bereiche, die besonders langsam abgearbeitet werden.

Docker für Windows kompilieren
Das Open-Source-Projekt Docker will das Verteilen von Software einfacher machen. Entwickelt wurde es mit Linux, jetzt gibt es eine erste Portierung auf Windows.

Geballtes Wissen zu .NET und zur Entwicklungspraxis
Nur noch wenige Tage. Dann startet die .NET Developer Conference kompakt 2014. Lernen Sie dort: Web API2, arc42, Code Generierung, Tasks, Native mit .NET, Rx, Roslyn, Xamarin, Verteilte Teamumgebung, Universal Windows Apps, Azure Websites, Softwarezellen. Am 2. und 3. Dezember 2014 in Köln.

Jetzt kostenlos anmelden:

dotnetpro.de

facebook.de/dotnetpro

twitter.com/dotnetpro_mag

gplus.to/dotnetpro