Client Object Modellen in SharePoint 2010

Startpunt voor succesvolle eerste clientapplicatie

Ton Stegeman

Voor SharePoint ontwikkelaars is er met de komst van de nieuwe versie weer veel nieuws onder de zon. Een van de zaken die de aandacht trekken zijn de nieuwe Client Object modellen. Zoals de naam doet vermoeden, zijn deze APIs te gebruiken om client applicaties voor SharePoint te programmeren.

In SharePoint 2007 is code geschreven om het SharePoint objectmodel alleen op de server te gebruiken. Op de client werden de webservices en de RPC protocollen gebruikt. In SharePoint Foundation 2010 (voorheen WSS) en SharePoint Server 2010 hebben we straks drie nieuwe modellen:

- .NET Managed. Gebruik deze API om managed client applicaties te maken (vanaf .NET 3.5);
- Silverlight. Dit managed object model kan worden gebruikt om in Silverlight applicaties gebruik te maken van SharePoint functionaliteit (vanaf Silverlight 2);
- ECMAScript. API om vanuit javascript object georienteerd SharePoint functionaliteit aan te roepen.

Deze 3 APIs zijn een subset van de server API van SharePoint Foundation 2010. Een groot deel van de objecten om site collections en alle onderliggende structuren te benaderen zijn aanwezig. Zeer krachtig, want zo kunnen straks eenvoudig betere clientapplicaties voor SharePoint gebouwd worden. Bovendien kunnen deze applicaties voor de nieuwe versie van SharePoint Online worden gebruikt.

Het programmeren met deze nieuwe objectmodellen werkt net even anders dan met het server objectmodel. In dit artikel worden van ieder objectmodel voorbeelden getoond. Let op: de codevoorbeelden zijn gemaakt met een vroege beta versie. Het is mogelijk dat de code in de RTM versie afwijkt. Foutafhandeling is niet opgenomen in de codevoorbeelden om ze zo compact mogelijk te houden.

.NET Managed Object Model

Om gebruik te maken van de .NET Managed API, krijgt het project in Visual Studio een referentie naar twee assemblies; Microsoft.SharePoint.Client.dll en Microsoft.SharePoint.Client.Runtime.dll. Deze zijn te vinden in de folder ´C:\Program Files\Common Files\Microsoft Shared\Web Server Extensions\14\ISAPI´ op de SharePoint server. De installatieprocedure van de client applicatie dient deze twee assemblies te installeren. In onderstaande voorbeelden zien we stukken code van een Windows Forms applicatie die vanaf een client een aantal zaken in een SharePoint teamsite uitrolt:

- Een lijst Companies;
- Een lookup site column naar de lijst Companies;
- Een nieuw Contact content type dat gebruik maakt van deze lookup kolom;
- Associeer het nieuwe content type met de lijst met contactpersonen.

De eerste stap is het kijken of de lijst Companies al bestaat. Net als in het server object model, wordt de SharePoint context hiervoor gebruikt. Instantieer hiervoor een ClientContext object, gebruikmakend van de url van de SharePoint-site. De eigenschap Web van dit object is de equivalent van het SPWeb server object en is de referentie naar de site. Dit Web object heeft een eigenschap Lists, een referentie naar de ListCollection met alle lijsten in de site. Tot nu toe lijkt alles hetzelfde te gaan als in de server API. Dat is echter niet het geval. Tot nu toe is er nog geen enkele opdracht richting de SharePoint database uitgevoerd. De client applicatie heeft nog geen contact gehad met SharePoint. De ListCollection is dus nog leeg. De Load methode zorgt ervoor dat de collectie wordt geladen. Dit gebeurt echter pas op het moment dat context.ExecuteQuery wordt uitgevoerd. Dit lijkt omslachtig, maar zorgt ervoor dat het aantal roundtrips naar de SharePoint server beperkt kan blijven. Initialiseer zoveel mogelijk client objecten, voordat de query gestart wordt. De zojuist beschreven code vind je in Codevoorbeeld 1. De applicatie itereert nu door de collectie met lijsten en kijkt op basis van de titel van de lijst of de lijst Companies bestaat.

```
List companiesList = null;
using (ClientContext context = new ClientContext("http://tstmss/
teamsite"))
{
    Web teamWeb = context.Web;
    ListCollection lists = teamWeb.Lists;
    context.Load(lists);
    context.ExecuteQuery();

    foreach (List list in lists)
```

```
    {
        if (list.Title.Equals("Companies"))
        {
            companiesList = list;
            break;
        }
    }
}
```

CODEVOORBEELD 1 ZOEKEN OF EEN LIJST BESTAAT.

Het patroon dat je in bovenstaand voorbeeld ziet, is aanwezig in alle voorbeelden, bij alle client APIs Er zijn steeds drie stappen:

- Objecten instantieren en benodigde eigenschappen opvragen;
- Objecten laden en eventueel een selectie maken;
- De ClientContext verzendt met ExecuteQuery de opdracht(en) richting SharePoint.

Het volgende voorbeeld maakt de voorgaande code robuuster. In plaats van op de titel van de lijst kijkt de applicatie of de lijst bestaat op basis van de naam van de root folder. Als een gebruiker de titel van de lijst in SharePoint wijzigt, blijft de code normaal functioneren. Hiervoor wordt het if statement in de foreach loop aangepast. De list.Title wordt vervangen door list.RootFolder.Name. Na deze aanpassing geeft de client nu een foutmelding van het type PropertyOrFieldNotInitializedException. Dit wordt veroorzaakt door een tweede wezenlijk verschil tussen de server en de client objectmodellen. In de server API zijn na het laden van het object altijd alle properties beschikbaar. In de client APIs daarentegen, worden slechts een zeer beperkt aantal eigenschappen geladen. Alle eigenschappen die gebruikt worden in de code, moeten ook expliciet worden geladen. Het laden van de ListCollection uit Codevoorbeeld 1 wordt aangepast naar de code in Codevoorbeeld 2.

```
ListCollection lists = teamWeb.Lists;
context.Load(lists, listCollection => listCollection.IncludeWithDe
faultProperties(list => list.RootFolder));
context.ExecuteQuery();
```

CODEVOORBEELD 2: SPECIFICEER ALLE BENODIGDE EIGENSCHAPPEN.

In dit voorbeeld wordt gebruik gemaakt van de methode IncludeWithDefaultProperties die op veel collectieobjecten aanwezig is. In de parameters van deze methode, wordt gespecificeerd welke eigenschappen van de List objecten worden geladen. IncludeWithDefaultProperties is niet beschikbaar in de ECMAScript javascript library. Een alternatief voor het gebruik van IncludeWithDefaultProperties is het vervangen van Load door LoadQuery. Deze methode accepteert een LINQ-query als parameter. Codevoorbeeld 3 toont de LINQ-query. Deze query specificeert dat de RootFolder eigenschap wordt geladen en zorgt ervoor dat alleen de lijsten met RootFolder naam ´Companies´ worden geladen. Dit maakt de routine om te kijken of de lijst bestaat veel efficiënter. In plaats van alle lijsten te laden en op de client te evalueren, zorgt de query ervoor dat alleen de gewenste lijst terug wordt geladen.

```
ListCollection lists= teamWeb.Lists;
IEnumerable<List> resultLists = context.LoadQuery(lists.Include(
                    list => list.Title,
                    list => list.Description,
                    list => list.RootFolder,
                    list => list.Id).Where(
                        list => list.RootFolder.Name ==
                        "Companies"
    ));
context.ExecuteQuery();

if (resultLists.Count() == 1)
{
    companiesList = resultLists.First();
}
```

CODEVOORBEELD 3: VRAAG ALLEEN DE BENODIGDE LIJST OP.

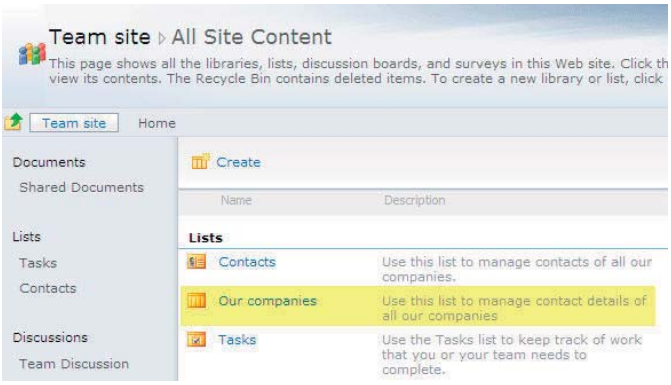
Volgende stap is het aanmaken van de lijst Companies als deze nog niet bestaat, en anders het eventueel aanpassen van titel en beschrijving. Voor het aanmaken van een nieuwe lijst wordt gebruik gemaakt van een ListCreationInformation object. Voor veel SharePoint objecten bestaat zo´n object. Het patroon is hetzelfde als in de voorgaande voorbeelden. Na het opzetten van de ClientContext wordt het nieuwe List object aangemaakt en worden Title en Description ingesteld. Op het moment dat de lijst volledig ingericht is, wordt de Update methode aangeroepen, gevolgd door het uitvoeren van de query met ExecuteQuery. De applicatie heeft geen enkel contact met SharePoint tot het strikt noodzakelijk is, tijdens het aanroepen van ExecuteQuery. De code hiervoor vind je in Codevoorbeeld 4.

```
bool update = false;
if (companiesList == null)
{
    ListCreationInformation creationInfo = new ListCreation-
    Information();
    creationInfo.Title = "Companies";
    creationInfo.TemplateType = (int)ListTemplateType.GenericList;
    companiesList = teamWeb.Lists.Add(creationInfo);
    update = true;
}
String companiesTitle = "Our companies";
if (!companiesList.Title.Equals(companiesTitle))
{
    companiesList.Title = companiesTitle;
    update = true;
}
String companiesDescription = "Use this list to manage contact
details of all our companies";
if (companiesList.Description==null ||
    !companiesList.Description.Equals(companiesDescription))
{
    companiesList.Description = companiesDescription;
    update = true;
}
if (update)
{
    companiesList.Update();
    context.ExecuteQuery();
}
```

CODEVOORBEELD 4: AANMAKEN EN BIJWERKEN COMPANIES LIJST.

De Windows Forms applicatie heeft nu de lijst in de teamsite aangemaakt. Zie Figuur 1.

Volgende stap is het aanmaken van de site column. Dit wordt een lookup veld dat verwijst naar de Companies lijst. In het vorige voor-



beeld checkte de code vooraf of een object bestaat. Als dat niet het geval is, wordt het object aangemaakt en anders aangepast. Deze stappen kunnen ook gedaan worden in één keer, waarbij gebruik wordt gemaakt van foutafhandeling in de .NET Managed API. Dit maakt de code compacter en het scheelt een aanroep naar de SharePoint server. Codevoorbeeld 5 toont de code.

```
String fieldName = "OurCompany";
Field companyField = null;

ExceptionHandlingScope scope = new ExceptionHandlingScope(context);
using (scope.StartScope())
{
    using (scope.StartTry())
    {
        companyField = teamWeb.Fields.GetByInternalNameOrTitle
        (fieldName);
    }
    using (scope.StartCatch())
    {
        String lookupXML = "<Field SourceID=\"http://schemas.
        microsoft.com/sharepoint/v3\" Type=\"Lookup\"
                           Name=\"{1}\" DisplayName=\"Our
                           company\" Required=\"TRUE\" List=\
                           "{0}\" ShowField=\"Title\" />";
        lookupXML = String.Format(lookupXML, companiesList.Id.
        ToString("B"), fieldName);
        companyField = teamWeb.Fields.AddFieldAsXml
        (lookupXML, true, AddFieldOptions.AddFieldCheckDisplayName);
    }
    using (scope.StartFinally())
    {
        companyField = teamWeb.Fields.GetByInternalNameOrTitle
        (fieldName);
    }
}
context.ExecuteQuery();
```

CODEVOORBEELD 5: AANMAKEN SITE COLUMN DOOR GEBRUIK TE MAKEN VAN FOUTAFHANDELING.

Uitgangspunt in bovenstaande code is dat er in StartTry() een fout optreedt als het veld niet bestaat en dat het niet bestaan van het veld de enige reden is. Verder valt in Codevoorbeeld 5 op dat er voor de velden (zowel voor een lijst als een site column) geen CreationInformation objecten bestaan. Net als in het server object model, kunnen velden aangemaakt worden met behulp van een XML string met de velddefinitie.
Volgende stap in het proces is het aanmaken van een nieuw contenttype. De code wordt getoond in Codevoorbeeld 6. Hiervoor wordt een query uitgevoerd op de AvailableContenttypes collectie van het Web. Daarbij zoeken we in één query naar het nieuwe contenttype, om te kijken of dat bestaat, en het parent contenttype voor het contenttype. Wederom een andere aanpak dan bij het programmeren met het server objectmodel. Zorg er in de client objectmodellen voor dat zoveel mogelijk acties op de server samengevoegd worden in één ExecuteQuery statement. Dit voorkomt veel extra communicatie tussen de client en de server.

```
ContentTypeCollection contentTypeCollection = teamWeb.
AvailableContentTypes;
IEnumerable<ContentType> ourContactContentTypes = context.LoadQuery(
            contentTypeCollection.Include(
                ct => ct.FieldLinks).Where(
                ct => ct.Name == "OurContact"));
IEnumerable<ContentType> contactContentTypes = context.LoadQuery(
            contentTypeCollection.Include().Where(
                ct => ct.Name == "Contact"));
context.Load(contentTypeCollection);
context.ExecuteQuery();

ContentType ourContactContentType = ourContactContentTypes.First-
OrDefault();
ContentType parentContentType = contactContentTypes.FirstOr-
Default();

if (ourContactContentType == null && parentContentType!=null)
{
    ContentTypeCreationInformation creationInfo = new ContentType
    CreationInformation();
    creationInfo.Name = "OurContact";
    creationInfo.ParentContentType = parentContentType;
    parentContentType = teamWeb.ContentTypes.Add(creationInfo);
    parentContentType.Update(true);
    context.ExecuteQuery();
}
```

CODEVOORBEELD 6: AANMAKEN NIEUW CONTENT TYPE.

Het aanmaken van het nieuwe contenttype in bovenstaand voorbeeld werkt met een ContenttypeCreationInformation object. Om te kijken of de lookup site column uit Codevoorbeeld 5 bestaat in de FieldLinks collectie van het contenttype, wordt een LINQ-query gebruikt. De FieldLinks collectie van het contenttype wordt bewaard in een variabele. waarop de LINQ-query wordt uitgevoerd. Zie Codevoorbeeld 7. Dit voorbeeld voegt tevens de site column toe, als deze nog niet met het contenttype is geassocieerd. Let er op dat de Update methode van het ContentType moet worden aangeroepen om de wijzigingen daadwerkelijk door te voeren.

```
IEnumerable<FieldLink> contactLinks = ourContactContentType.Field-
    Links;
FieldLink companyLink = contactLinks.Where(ct => ct.Name.
Equals("OurCompany")).FirstOrDefault();
if (companyLink==null)
{
    FieldLinkCreationInformation companyLinkInfo = new FieldLink
    CreationInformation();
    companyLinkInfo.Field = teamWeb.Fields.GetByInternalNameOr-
    Title("OurCompany");
    ourContactContentType.FieldLinks.Add(companyLinkInfo);
    ourContactContentType.Update(true);
    context.ExecuteQuery();
}
```

CODEVOORBEELD 7: FIELDLINK ZOEKEN MET EEN LINQ QUERY.

In de codevoorbeelden ontbreekt nog de stap om het contenttype te associëren met de lijst. Hiervoor worden dezelfde technieken als hierboven beschreven gebruikt. De methode AddExistingContentType van de ContentTypeCollection legt de associatie met de lijst. De methode DeleteObject van het ContentType object verwijdert het standaard Contact content type van de lijst. Om deze wijzigingen door te voeren, hoeft de Update methode van het List object niet worden aangeroepen voordat ExecuteQuery wordt aangeroepen.
Het laatste voorbeeld (Codevoorbeeld 8) in dit hoofdstuk over het .NET Managed Client object model toont hoe een lijstitem toe kan worden gevoegd aan de lijst met Companies. Net als in de server API, wordt eerst een CAML query geconstrueerd. Deze query checkt of het default item al in de lijst aanwezig is.

```
String defaultCompany = "Adventure Works";

CamlQuery query = new CamlQuery();
query.ViewXml = String.Format("<View><Query><Where><Eq><FieldRef-
Name='Title'/><Value Type='Text'>{0}</Value></Eq></Where></
Query></View>", defaultCompany);
ListItemCollection selectCompanies = companiesList.GetItems
(query);
context.Load(selectCompanies);
context.ExecuteQuery();
if (selectCompanies.Count == 0)
{
    ListItemCreationInformation itemCreateInfo = new ListItem-
```

```
    CreationInformation();
    ListItem companyItem = companiesList.AddItem(itemCreateInfo);
    companyItem["Title"] = defaultCompany;
    companyItem.Update();
    context.ExecuteQuery();
}
```

CODEVOORBEELD 8: ITEM TOEVOEGEN AAN DE LIJST MET COMPANIES.

Als het default item niet wordt gevonden, wordt het toegevoegd met behulp van een ListItemCreationInformation object.

Silverlight Object Model

De Silverlight API is eveneens managed en is te gebruiken door in Visual Studio een referentie te maken naar de assemblies Microsoft.SharePoint.Client.Silverlight.dll en Microsoft.SharePoint.Client.Silverlight.Runtime.dll. Deze zijn te vinden op de SharePoint server in de folder 'C:\Program Files\Common Files\Microsoft Shared\Web Server Extensions\14\TEMPLATE\LAYOUTS\ClientBin'.
Veel van de code lijkt zeer sterk op de code uit het vorige hoofdstuk. Het grote verschil met de .NET Managed API is dat het Silverlight Client Object Model asynchroon is.

Om een company logo te tonen wordt gebruik gemaakt van een nieuwe SharePoint 2010 feature. Hiermee kan bij een lookup kolom een of meerdere velden uit de lookup lijst worden geselecteerd, die ook aan de lijst wordt toegevoegd. Dit komt overeen met de BDC kolom in SharePoint 2007. Codevoorbeeld 9 toont de aanroep van de query om de items uit de lijst te lezen. Tot aan de aanroep van ExecuteQuery loopt deze code via dezelfde patronen als in bovenstaande voorbeelden. Eerst de ClientContext opzetten gevolgd door het specificeren van de gegevens die SharePoint terug moet geven aan de client.
De veldwaarden horen bij de waarden die niet standaard mee gestuurd worden, en moeten dus ook opgegeven worden in het Include statement.

```
using (ClientContext context = new ClientContext("http://tstmss/
team"))
{
    Web teamWeb = context.Web;
    _contactsList = teamWeb.Lists.GetByTitle("Contacts");

    CamlQuery camlQuery = new CamlQuery();
    camlQuery.ViewXml = "<View><RowLimit>100</RowLimit></View>";

    _contacts = _contactsList.GetItems(camlQuery);

    context.Load(_contacts,
         items => items.Include(
            item => item.Id,
            item => item["FullName"],
            item => item["Email"],
            item => item["Picture"],
            item => item["WorkCity"],
            item => item["CellPhone"],
            item => item["ContactCompany_x003a_LogoUrl"],
            item => item.Id,
            item => item.DisplayName));

    ClientRequestSucceededEventHandler success = new ClientRequest
    SucceededEventHandler(SuccesHandler);
    ClientRequestFailedEventHandler failure = new ClientRequest-
    FailedEventHandler(FailureHandler);
    context.ExecuteQueryAsync(success, failure);
}
```

CODEVOORBEELD 9: CONTACTPERSONEN UIT EEN LIJST IN SILVERLIGHT CLIENT OBJECT MODEL.

Vlak voor de aanroep van de query worden er twee event handlers in het leven geroepen. Deze worden als parameters meegegeven aan ExecuteQuery. Dit zijn de handlers voor de events die afgevuurd worden bij afronding van de query. Als deze succesvol is uitgevoerd, wordt de afhandeling van het resultaat uitgevoerd in de ClientRequestSucceededEventHandler. Als er een fout optreedt, eindigt de code in de ClientRequestFailedEventHandler. Codevoorbeeld 10 toont de succes event handler van bovenstaande Silverlight applicatie.

```
private void SuccesHandler(object Sender, ClientRequestSucceeded-
EventArgs e)
{
    List<Contact> contacts = new List<Contact>();
    foreach (ListItem contact in _contacts)
    {
        Contact newContact = new Contact(contact["FullName"].
        ToString());
        newContact.EMail = contact["Email"].ToString();
        FieldUrlValue pictureUrl = contact["Picture"] as Field-
        UrlValue;
        newContact.PictureUrl = pictureUrl.Url;
        newContact.City = contact["WorkCity"].ToString();
        newContact.Phone = contact["CellPhone"].ToString();
        FieldLookupValue companyLogo = contact["ContactCompany_
        x003a_LogoUrl"] as FieldLookupValue;
        newContact.CompanyLogo = companyLogo.LookupValue;
        contacts.Add(newContact);
    }
    Dispatcher.BeginInvoke(() =>
    {
        ContactList.ItemsSource = contacts;
    });
}
```

CODEVOORBEELD 10: SUCCES EVENT HANDLER NA HET LADEN VAN CONTACT PERSONEN.

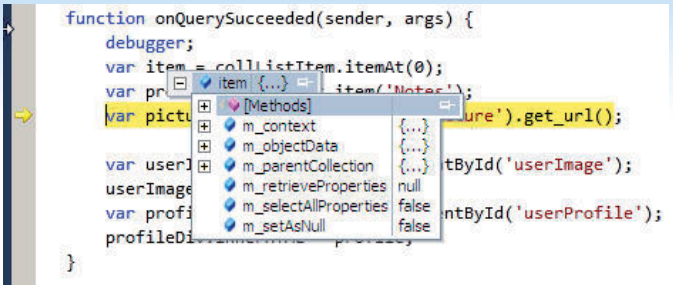
Deze code itereert door alle gevonden contactpersonen in de ListItemCollection. Voor ieder item wordt er een Contact object geinstantieerd. De collectie met deze objecten wordt als datasource aan de XAML-code gekoppeld. Omdat de uitvoer van de query asynchroon is, moet het aanpassen van de UI uitgevoerd worden op de user interface thread. Daarvoor wordt vlak voor het zetten van de ItemsSource Dispatcher.BeginInvoke aangeroepen.

ECMAScript Object Model

Het derde nieuwe client objectmodel is een javascript API. Net als de Silverlight variant werkt dit objectmodel asynchroon. De volgende codevoorbeelden zijn onderdeel van een LAYOUTS pagina. Deze pagina toont alle gebruikers die direct rechten toegewezen hebben gekregen op de site. Door op de naam van een gebruiker te klikken wordt extra informatie over deze gebruiker getoond. De pagina toont tevens een lijst van beschikbare SharePoint groepen. Door op een groepnaam te klikken, wordt de geselecteerde gebruiker aan de groep toegevoegd en worden zijn/haar directe permissies van de site verwijderd.
Het laden van gebruikers en groepen gebeurt met het server objectmodel in de pagina, de rest met de ECMAScript API. Door gebruik te maken van de javascript library wordt voorkomen dat de pagina voor iedere actie opnieuw geladen moet worden. Om gebruik te maken van de SP namespace uit deze library, wordt eerst een verwijzing naar de bijbehorende script file 'SP.js' aan de pagina toegevoegd:

```
<SharePoint:ScriptLink Name="sp.js" LoadAfterUI="true"
Localizable="false" runat="server" />
```

In feite bestaat de API uit een aantal JS files met de bijbehorende debug equivalenten. Zie de SharePoint Foundation SDK voor de



FIGUUR 2: ECMA SCRIPT DEBUGGEN.

volledige lijst. Door gebruik te maken van het ScriptLink control wordt er automatisch voor gezorgd dat alle benodigde JS bestanden geladen wordt. Als in de web.config het debug attribuut van het <Compilation /> element op true staat, zorgt het ScriptLink control ervoor dat automatisch de debug files worden geladen. In plaats van SP.js wordt dan SP.debug.js geladen. Dit is een ongecomprimeerde JS file.
De tweede gelinkte javascript file bevat de scripts voor de pagina. Codevoorbeeld 11 toont de code voor het ophalen van de gegevens van een gebruiker. In de javascript functie getUserInfo wordt, net als in de eerder behandelde objectmodellen eerst de ClientContext opgebouwd. Het script wordt aangeroepen vanuit een ASPX-pagina in de LAYOUTS folder, en is daardoor al op de hoogte van de SharePoint context. Deze kan eenvoudig geinitialiseerd worden door gebruik te maken van de Current context. Vervolgens wordt de User Information List opgezocht. Er wordt een CAML query gedefinieerd die de gegevens van de gebruiker ophaalt, waarna de query uitgevoerd wordt en de collectie met lijst items wordt geladen.

```
function getUserInfo(userID) {
    var clientContext = new SP.ClientContext.get_current();
    var web = clientContext.get_web();
    var userInfoList = web.get_siteUserInfoList();
    var camlQuery = new SP.CamlQuery();
    camlQuery.set_viewXml('<View><RowLimit>10</RowLimit></View>');
    camlQuery.set_viewXml('<View><Query><Where><Eq><FieldRef
    Name=\'ID\'/><Value Type=\'Number\'>' + userID + '</Value></
    Eq></Where></Query><RowLimit>1</RowLimit></View>');
    this.collListItem = userInfoList.getItems(camlQuery);
    clientContext.load(collListItem);
    clientContext.executeQuery(Function.createDelegate(this, this.
    onQuerySucceeded), Function.createDelegate(this, this.onQuery
    Failed));
}
```

CODEVOORBEELD 11: OPHALEN GEGEVENS GESELECTEERDE GEBRUIKER.

Conclusie

De nieuwe objectmodellen zijn een belangrijke ontwikkeling voor SharePoint-ontwikkelaars. Ze maken het ontwikkelen van clientapplicaties voor SharePoint veel eenvoudiger. In dit artikel is in een aantal codevoorbeelden elk van de drie nieuwe objectmodellen kort voorgesteld. Hierbij is aangegeven op welke punten client ontwikkeling afwijkt van server ontwikkeling. Dit artikel is bedoeld als startpunt voor een succesvolle eerste clientapplicatie.

Ton Stegeman, is als SharePoint architect werkzaam bij PGGM. Op zijn weblog (www.tonstegeman.com/blog) is veel SharePoint development gerelateerde content te vinden. Daarnaast publiceert hij op CodePlex (http://sharepointobjects.codeplex.com) een aantal open source SharePoint projecten. Hij is MVP en bereikbaar via email op ton@tonstegeman.com.