Sitecore® Experience Platform™ 7.5 or later

Developer's Guide to Sitecore.Services.Client

Table of Contents

Chapter 1 Introduction and Overview 4
1.1 What is Sitecore.Services.Client? 5
Chapter 2 Security 6
2.1 Overview 7
2.2 Security Policies 8
2.2.1 Exclude Controllers from Security Policies 8
2.3 Authorization Filters 9
2.4 Custom Authorization Filters 10
Chapter 3 The ItemService 11
3.1 Introduction 12
3.1.1 Implementation 12
3.1.2 ItemModel Return Values 12
3.1.3 ItemService Security 13
Request Security Context 13
Authentication 13
Anonymous Access 13
3.1.4 Example ItemService Requests 13
3.2 Using the ItemService from a SPEAK component 15
3.3 Using the ItemService from JavaScript 16
3.3.1 Create Items 16
3.3.2 Fetch Items 16
3.3.3 Fetch the Children of an Item 17
3.3.4 Save Items 17
3.3.5 Destroy Items 17
3.3.6 Search for Items 18
How to Use search() 18
How to Use query() 18
3.3.7 Create a Dirty Item 18
3.3.8 Check if an Item is Dirty 19
3.3.9 Retrieve an Item as Raw JSON 19
3.3.10 Trackable Items 20
3.3.11 Save Items Automatically 20
3.3.12 The hasChanged() method 20
3.3.13 The revertChanges() method 21
3.3.14 The Query Object 21
3.3.15 How Promises Are Handled 22
3.3.16 Working with Middleware 22
3.3.17 Working with Event Emitters 23
3.3.18 ItemService and Validation 23
3.4 Using the RESTful API for the ItemService 25
3.4.1 Authentication 25
Login 25
Logout 25
3.4.2 Retrieve an Item by ID 25
3.4.3 Retrieve an Item by Content Path 27
3.4.4 Retrieve the Children of an Item 28
3.4.5 Create an Item 29
3.4.6 Edit an Item 30
3.4.7 Delete an Item 31
3.4.8 Run a Stored Query 32

3.4.9 Run a Sitecore Search ... 33
3.4.10 Run a Stored Sitecore Search ... 35
Chapter 4 The EntityService ... 38
4.1 Introduction ... 39
4.2 How to Create an EntityService ... 40
4.2.1 Steps to Create an EntityService ... 40
4.2.2 Persisting an EntityService ... 40
How to Store Business Objects in Sitecore ... 40
4.2.3 Adding Custom Methods to the Service ... 41
4.3 EntityService Validation ... 42
4.3.1 Basic Validators ... 42
4.3.2 Extend Validation ... 42
Defining a new Validator type ... 42
4.4 The EntityService and CORS ... 44
4.4.1 How to Configure CORS ... 44
4.4.2 How to Enable CORS for an EntityService Controller ... 44
4.5 EntityService Metadata Exchange ... 45
4.5.1 Action Endpoints ... 47
4.5.2 Entity Metadata ... 47
4.5.3 .NET to Javascript Type Mapping ... 48
4.5.4 Support for Non-Primitive Types ... 49
4.6 Other EntityService features ... 50
4.6.1 Promises ... 50
4.6.2 Middleware ... 50
4.6.3 Event Emitters ... 51
4.7 Using the EntityService from a SPEAK component ... 52
4.8 Using the EntityService from JavaScript ... 53
4.8.1 The EntityService.Entity ... 54
4.8.2 Creating Entities ... 54
4.8.3 Fetching a Single Entity ... 55
4.8.4 Fetching Multiple Entities ... 56
4.8.5 Saving Entities ... 56
4.8.6 Destroying Entities ... 57
4.8.7 Dirty Entities ... 57
4.8.8 isNew ... 57
4.8.9 How to Retrieve Raw JSON ... 58
4.8.10 Trackable Entities ... 58
4.8.11 Automatically Save ... 59
4.8.12 hasChanged() ... 59
4.8.13 revertChanges() ... 60
4.9 Using the RESTful API for the EntityService ... 61
4.9.1 Default Routing ... 61
Mapping HTTP Verbs ... 61
Mapping Type Name to URL ... 61
Custom Controller Action Routing ... 62
4.9.2 Configuration ... 62
Custom Route Configuration ... 62
Service Endpoint Stem ... 62
4.9.3 Override Namespace Routing ... 62

Chapter 1

Introduction and Overview

This chapter gives you an overview and an introduction. It contains the following section:

- What is Sitecore.Services.Client?

1.1 What is Sitecore.Services.Client?

Sitecore.Services.Client provides a service layer on both the server and the client side of Sitecore applications that you use to develop data-driven applications.

Sitecore.Services.Client is configurable and extendable, and the framework and the “scaffolding” it gives you help you create the client-server communication in an application in a consistent way. Sitecore.Services.Client uses the ASP.NET Web API as a foundation

The framework gives you convention over configuration: you do not have to configure your server side controller. You mark the controller with the [ServicesController] attribute and this gives you all the features. The framework also gives you a mechanism that lets clients gain information about service metadata. You can then use this information when you create subsequent calls to the service. In addition, the server side in general responds using standard HTTP response types.

Clients can use the server-side classes in three different ways:

- Use SPEAK components.
- Use client-side JavaScript.
- Use the Restful API directly.

Sitecore.Services.Client provides two services:

- ItemService: this service gives you access to regular Sitecore items.
- EntityService: this service gives you access to business objects that you define.

Later sections in this document describe the two services in more detail.

Chapter 2

Security

This chapter gives you an overview of security and Sitecore.Services.Client. It contains the following section:

- Overview
- Security Policies
- Authorization Filters
- Custom Authorization Filters

2.1 Overview

The Entity and ItemServices use the Web API ActionFilters to determine whether they handle or reject a request. You can read about ActionFilters at MSDN: http://msdn.microsoft.com/en-us/library/system.web.http.filters.actionfilterattribute.aspx.

Sitecore.Services.Client provides two layers of security:

1. A security policy that applies to all Sitecore.Services.Client requests
2. Individual filters which add additional requirements on requests that are to be executed

The ItemService has some additional security settings, see ItemService Security.

2.2 Security Policies

Sitecore.Services.Client ships with three security policies, detailed below:

1. `Sitecore.Services.Infrastructure.Web.Http.Security.ServicesOffPolicy`
   This policy denies access to all Entity and ItemServices.
2. `Sitecore.Services.Infrastructure.Web.Http.Security.ServicesLocalOnlyPolicy`
   This policy denies access to all Entity and ItemServices from requests originating from remote clients.
3. `Sitecore.Services.Infrastructure.Web.Http.Security.ServicesOnPolicy`
   This policy allows access to all Entity and ItemServices.

You configure security policy with the `Sitecore.Services.SecurityPolicy` setting in the `Sitecore.Services.Client.config` file. The default value is `ServicesLocalOnlyPolicy.`

You can create custom security policies by implementing the `Sitecore.Services.Infrastructure.Web.Http.Security.IAuthorizePolicy` interface and specifying the custom class name in the `Sitecore.Services.SecurityPolicy` configuration setting.

2.2.1 Exclude Controllers from Security Policies

You can exclude controllers from the security policy you use. To do this, add an `allowedController` element under `api/services/configuration/allowedControllers` in the `Sitecore.Services.Client.config` file.

2.3 Authorization Filters

The `sitecore/api/configuration/filters` section in the `Sitecore.Services.Client.config` configuration file defines what action filters the Sitecore.Services.Client installs.

It installs the following filters by default:

1. `Sitecore.Services.Infrastructure.Web.Http.Filters.AnonymousUserFilter`
   This filter ensures that the requests to the ItemService respect the configuration settings for unauthenticated users.
2. `Sitecore.Services.Infrastructure.Web.Http.Filters.SecurityPolicyAuthorisationFilter`
   This filter runs the security policy defined in the `Sitecore.Services.SecurityPolicy` configuration setting.
3. `Sitecore.Services.Infrastructure.Web.Http.Filters.LoggingExceptionFilter`
   This filter ensures that uncaught exceptions do not leak out over the Web API requests. It writes details about uncaught exceptions in the Sitecore logs and it sets the response status for the request to Internal Server Error (500).
4. `Sitecore.Services.Infrastructure.Web.Http.Filters.RequireHttpsFilter`
   This filter makes HTTPS mandatory for all Web API requests to the site. It is commented out by default.
5. `Sitecore.Services.Infrastructure.Web.Http.Filters.ServicesRequireHttpsFilter`
   This filter makes HTTPS mandatory for all EntityService and ItemService requests. It is commented out by default.

2.4 Custom Authorization Filters

You create custom authorization filters this way:

1. Derive a filter class from `System.Web.Http.Filters.AuthorizationFilterAttribute` and override the `OnAuthorization(HttpActionContext actionContext)` method.
2. Add the new filter class definition to the `sitecore/api/configuration/filters` section in the `Sitecore.Services.Client.config` configuration file.

Chapter 3

The ItemService

This chapter describes the ItemService. It has this content:

- Introduction
- Using the ItemService from a SPEAK component
- Using the ItemService from JavaScript
- Using the RESTful API for the ItemService

3.1 Introduction

The ItemService provides a single HTTP endpoint and API to interact with Sitecore items over HTTP.

Sitecore.Services.Client ships with routes predefined to interact with the ItemService, and you do not need to do any server-side development to use the ItemService.

You can use the ItemService in three ways:

- You can use it in SPEAK applications, using the StoredQueryDataSource data source. When you use the ItemService this way, it is completely transparent.
- You can use the ItemService from client-side JavaScript (in the browser).
- You can use the RESTful API to access Sitecore items directly.

3.1.1 Implementation

The `Sitecore.Services.Infrastructure.Sitecore.Controllers.ItemServiceController` class implements the service. We have sealed the class itself so that other classes cannot inherit from it. Only one single instance of the ItemService is supposed to run in a Sitecore website.

3.1.2 ItemModel Return Values

Sitecore.Services.Client maps Sitecore items into instances of `Sitecore.Services.Core.Model.ItemModel` when ItemService returns them. The `ItemModel` instances contain the raw field values of the Sitecore item in a `Dictionary<string, object>,` with the following additional keys:

- ItemID
- ItemName
- ItemPath
- ParentID
- TemplateID
- TemplateName
- CloneSource
- ItemLanguage
- ItemVersion
- DisplayName
- HasChildren
- ItemIcon
- ItemMedialUrl
- ItemUrl

3.1.3 ItemService Security

The ItemService has the following additional security features.

Request Security Context

Requests to the ItemService run in the context of the current Sitecore user.

Authentication

The ItemService provides two routes for authentication:

```
auth/login
auth/logout
```

You must make requests to `auth/login` over HTTPS.

When you make requests to this route from JavaScript, you must load the whole page over HTTPS to avoid the request failing because it is Cross-Origin.

Anonymous Access

The default is that the `extranet\Anonymous` user does not have access to the ItemService. You set this behavior with the `Sitecore.Services.AllowAnonymousUser` setting in the `Sitecore.Services.Client` configuration file.

When you set `Sitecore.Services.AllowAnonymousUser` to true, then the ItemService will run anonymous requests in the security context of the user defined in the `Sitecore.Services.AnonymousUser` configuration setting. Such requests will, by default, do user impersonation and run as the `sitecore\ServicesAPI` user.

3.1.4 Example ItemService Requests

The following examples show how to perform read-only [GET] operations with the ItemService:

- Get Item:
  `/sitecore/api/ssc/item/110D559F-DEA5-42EA-9C1C-8A5DF7E70EF9`
- Get Item with Metadata
  `/sitecore/api/ssc/item/110D559F-DEA5-42EA-9C1C-8A5DF7E70EF9?includeMetadata=true`
- Get Item with standard template fields
  `/sitecore/api/ssc/item/110D559F-DEA5-42EA-9C1C-8A5DF7E70EF9?IncludeStandardTemplateFields=true`
- Get Item with field projection
  `/sitecore/api/ssc/item/110D559F-DEA5-42EA-9C1C-8A5DF7E70EF9?fields=Id,Name,TemplateName`
- Search
  `/sitecore/api/ssc/item/search?term=Home`
- Search with paging and field projection
  `/sitecore/api/ssc/item/search?IncludeStandardTemplateFields=False&Fields=ID%2CName&page=1&Term=sitecore`
- Get Item on Media Item
  `/sitecore/api/ssc/item/C19E9164-FF99-4A05-B8C0-E9C931DA111F`

- Get Item by content path
  `/sitecore/api/ssc/item/?path=/sitecore/content/Home`

3.2 Using the ItemService from a SPEAK component

You can use the ItemService from the StoredQueryDataSource SPEAK component. This component is described in the SPEAK Component Reference.

3.3 Using the ItemService from JavaScript

The ItemService is a standalone XHR (XMLHttpRequest) library for creating, fetching, saving and deleting Sitecore Items. It has many built in utilities and helpers to assist you with data transactions between the front end and the back end.

Unit.js
Many of the examples user functionality from Unit.js. If you want to run the examples literally, you must add a reference to Unit.js in your code.

3.3.1 Create Items

You create an item by passing an object and the path in the Sitecore tree where you want the server to create the item, and then calling the execute method:

```
var peopleService = new ItemService( {
  url: "/sitecore/api/ssc/people"
} );

var aNewGuy = {
  name: "David",
  isActive: true,
  gender: "male"
};

peopleService.create( aNewGuy ).path( "/sitecore/content/home" ).execute().then( function ( david
) {

  david.should.be.an.instanceOf( ItemService.Item );
  david.name.should.eql( "David" );
  david.isActive.should.eql( true );
  david.gender.should.eql( "male" );

  done();

} ).fail( done );
```

You create a dirty item by not calling the execute method:

```
var peopleService = new ItemService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.create().then( function () {

  done();

} ).fail( done );
```

3.3.2 Fetch Items

You fetch an item by ItemID with the fetchItem() method:

```
var peopleService = new ItemService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.fetchItem( "d4119c4f-31e9-4fd0-9fc4-6af1d6e36c8e" ).execute().then( function (
melton ) {

  melton.should.be.an.instanceOf( ItemService.Item );
  done();
} ).fail( done );
```

3.3.3 Fetch the Children of an Item

You fetch the children of an item with the fetchChildren() method:

```
var peopleService = new ItemService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.fetchItem( "d4119c4f-31e9-4fd0-9fc4-6af1d6e36c8e" ).execute().then( function (
melton ) {

  melton.should.be.an.instanceOf( ItemService.Item );
  melton.fetchChildren().execute().then( function ( meltonsChildren ) {

    meltonsChildren.should.be.an.Array.and.have.a.lengthOf( 3 );
    meltonsChildren[ 0 ].should.be.an.instanceOf( ItemService.Item );
    meltonsChildren[ 1 ].should.be.an.instanceOf( ItemService.Item );
    meltonsChildren[ 2 ].should.be.an.instanceOf( ItemService.Item );

    done();

  } ).fail( done );

} ).fail( done );
```

3.3.4 Save Items

You must either fetch or create an item before you can save it:

```
var peopleService = new ItemService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.fetchItem( "d4119c4f-31e9-4fd0-9fc4-6af1d6e36c8e" ).execute().then( function (
melton ) {

  melton.should.be.an.instanceOf( ItemService.Item );
  melton.name = "Melton the Magnificent";

  melton.save().execute().then( function ( savedMelton ) {

    savedMelton.name.should.eql( "Melton the Magnificent" );
    done();

  } ).fail( done );

} ).fail( done );
```

3.3.5 Destroy Items

You must either fetch or create an item before you can destroy it:

```
var peopleService = new ItemService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.fetchItem( "d4119c4f-31e9-4fd0-9fc4-6af1d6e36c8e" ).execute().then( function (
melton ) {

  melton.should.be.an.instanceOf( ItemService.Item );

  melton.destroy().execute().then( function () {
    done();

  } ).fail( done );

} ).fail( done );
```

3.3.6 Search for Items

You can search for item in two ways: you can use `search` or you can use `query`.

How to Use search()

You search for an item by passing a search term and then calling the execute method:

```
var peopleService = new ItemService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.search( "melton" ).execute().then( function ( queryResults ) {

  queryResults.should.have.a.property( "Links" ).and.be.an.Array;
  queryResults.should.have.a.property( "Results" ).and.be.an.Array;
  queryResults.should.have.a.property( "TotalCount" ).and.be.a.Number;
  queryResults.should.have.a.property( "TotalPage" ).and.be.a.Number;

  queryResults.Results[ 0 ].should.be.an.instanceOf( ItemService.Item );
  queryResults.Results[ 0 ].name.should.eql( "Banks Melton" );

  done();

} ).fail( done );
```

How to Use query()

You query for an item by passing a query item and then calling the execute method:

```
var peopleService = new ItemService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.query(( "xxxx-xxxx-xxxx-xxxx" ).execute().then( function ( queryResults ) {

  queryResults.should.have.a.property( "Links" ).and.be.an.Array;
  queryResults.should.have.a.property( "Results" ).and.be.an.Array;
  queryResults.should.have.a.property( "TotalCount" ).and.be.a.Number;
  queryResults.should.have.a.property( "TotalPage" ).and.be.a.Number;

  queryResults.Results[ 0 ].should.be.an.instanceOf( ItemService.Item );

  done();

} ).fail( done );
```

“xxxx-xxxx-xxxx-xxxx” is the Sitecore ID of the query item (see Run a Stored Query for details).

3.3.7 Create a Dirty Item

A dirty item is an item that is only in browser memory and not saved on the server. This can be useful if your application has to wait for user input. You create a dirty item like this:

```
var peopleService = new ItemService( {
  url: "/sitecore/api/ssc/people"
} );
```

```
peopleService.create().then( function ( aNewDirtyItem ) {

  /* At this point `aNewDirtyItem` has not been saved */
  aNewDirtyItem.should.be.an.instanceOf( ItemService.Item );
  aNewDirtyItem.name = "Melton";

  /* After modifying the dirty item, now we save it */
  aNewDirtyItem.save().execute().then( function () {

    done();

  } ).fail( done );

} ).fail( done );
```

3.3.8 Check if an Item is Dirty

You can check if an item has not been saved (it is a “dirty item") with the isNew property:

```
var peopleService = new ItemService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.create().then( function ( guy ) {

  guy.isNew.should.be.true;
  done();

} ).fail( done );
```

When item comes from the server, isNew is always false:

```
var peopleService = new ItemService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.create( {
  name: "guy"
} ).execute().then( function ( guy ) {

  guy.isNew.should.be.false;
  done();

} ).fail( done );
```

3.3.9 Retrieve an Item as Raw JSON

You may need to retrieve the data of an item as raw JSON (without all of the additional methods and properties). You can do it this way:

```
var peopleService = new ItemService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.fetchItem( "d4119c4f-31e9-4fd0-9fc4-6af1d6e36c8e" ).execute().then( function (
melton ) {

  var meltonAsJSON = melton.json();

  melton.should.be.an.instanceOf( ItemService.Item );
  meltonAsJSON.should.be.an.instanceOf( Object );
  meltonAsJSON.should.not.be.an.instanceOf( ItemService.Item );

  done();
} ).fail( done );
```

3.3.10 Trackable Items

You can add tracking to an item. This makes it possible to add additional functionality:

- The item can saved automatically when a property changes.
- You can call hasChanged() to see if the item has changed.
- You can call revertChanges() to revert property values.

You set the “trackable” option to “true” to add tracking to your item:

```
var peopleService = new ItemService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.fetchItem( "d4119c4f-31e9-4fd0-9fc4-6af1d6e36c8e" ).option( "trackable", true
).execute().then( function ( melton ) {

  melton.option( "trackable" ).should.be.true;

  done();

} ).fail( done );
```

3.3.11 Save Items Automatically

You can specify that Sitecore.Services.Client save changes to an item automatically. You set “trackable” to “true” to turn this feature on. You can listen to the save event by using the emitter “on” or “once”:

```
var peopleService = new ItemService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.fetchItem( "d4119c4f-31e9-4fd0-9fc4-6af1d6e36c8e" ).option( "trackable", true
).execute().then( function ( melton ) {

  /* Listening to the `save` event `once`. You can also use `on` here to continuously listen to
the `save` event. */
  melton.once( "save", function ( error ) {

    done();

  } );

  melton.name = "Melton the Magnificent";

} ).fail( done );
```

3.3.12 The hasChanged() method

You can specify that the ItemService add the hasChanged() method by setting “trackable” to “true”. You use the hasChanged() method to check if an item has changed:

```
var peopleService = new ItemService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.fetchItem( "d4119c4f-31e9-4fd0-9fc4-6af1d6e36c8e" ).option( "trackable", true
).execute().then( function ( melton ) {

  melton.hasChanged().should.be.false;
  melton.name = "Melton the Magnificent";
  melton.hasChanged().should.be.true;

  done();

} ).fail( done );
```

The hasChanged method does not return true if a value changes between null, undefined and ‘’ (the empty string):

```
var peopleService = new ItemService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.fetchItem( "d4119c4f-31e9-4fd0-9fc4-6af1d6e36c8e" ).option( "trackable", true
).execute().then( function ( melton ) {

  melton.hasChanged().should.be.false;
  melton.subscribed = "";
  melton.hasChanged().should.be.false;
  melton.subscribed = null;
  melton.hasChanged().should.be.false;
  melton.subscribed = undefined;
  melton.hasChanged().should.be.false;

  done();

} ).fail( done );
```

3.3.13 The revertChanges() method

You can specify that the ItemService add the revertChanges() method by setting “trackable” to “true”. You use the revertChanges() method to revert changes to the item:

```
var peopleService = new ItemService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.fetchItem( "d4119c4f-31e9-4fd0-9fc4-6af1d6e36c8e" ).option( "trackable", true
).execute().then( function ( melton ) {

  melton.name.should.eql( "Banks Melton" );
  melton.name = "Melton the Magnificent";
  melton.name.should.eql( "Melton the Magnificent" );
  melton.hasChanged().should.be.true;
  melton.revertChanges();
  melton.hasChanged().should.be.false;
  melton.name.should.eql( "Banks Melton" );

  done();

} ).fail( done );
```

3.3.14 The Query Object

Each ItemService method returns a query object:

```
var peopleService = new ItemService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.create( {} ).should.be.an.instanceOf( ItemService.Query );
peopleService.fetchItem( "d4119c4f-31e9-4fd0-9fc4-6af1d6e36c8e" ).should.be.an.instanceOf(
ItemService.Query );
peopleService.query( "/sitecore" ).should.be.an.instanceOf( ItemService.Query );
peopleService.search( "sitecore" ).should.be.an.instanceOf( ItemService.Query );
```

The query object has chainable helper methods. These methods add the given value to the query string of the request:

```
var peopleService = new ItemService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.create( {} )
  .database( "master" )
  .facet( "a facet" )
  .fields( "ItemName" )
  .includeMetadata( true )
  .includeStandardTemplateFields( true )
  .language( "en" )
  .page( 2 )
  .path( "/sitecore/content/home" )
  .sort( "aItemName" )
  .take( 5 )
  .version( "latest" )
  .should.be.an.instanceOf( ItemService.Query );
```

You call the execute() method to run the query. The execute() method returns a promise:

```
var peopleService = new ItemService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.fetchItem( "d4119c4f-31e9-4fd0-9fc4-6af1d6e36c8e" ).execute().then( function (
melton ) {

  melton.should.be.an.instanceOf( ItemService.Item );
  done();

} ).fail( done );
```

3.3.15 How Promises Are Handled

Almost all asynchronous calls return a promise. Specifically, and prominently, the execute() method returns a promise. The ItemService uses the q module to handle this.

```
var peopleService = new ItemService( {
  url: "/sitecore/api/ssc/people"
} );

var fetchQuery = peopleService.fetchItem( "d4119c4f-31e9-4fd0-9fc4-6af1d6e36c8e" );
var fetchQueryPromise = fetchQuery.execute();

fetchQueryPromise.should.have.a.property( "then" );
fetchQueryPromise.should.have.a.property( "fail" );
```

3.3.16 Working with Middleware

ItemService provides middleware functionality, using the sc-useify module:

```
ItemService.use( function ( data, next ) {

  data.timestamp = new Date().getTime();
  next( null, data );

} );
```

When you have integrated middleware as in the previous example, then the next time you receive data from the server, this data will have a “timestamp” property:

```
var peopleService = new ItemService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.fetchItem( "d4119c4f-31e9-4fd0-9fc4-6af1d6e36c8e" ).execute().then( function (
melton ) {

  melton.should.have.a.property( "timestamp" );
  done();

} ).fail( done );
```

You can clear all middleware workers (or clear by key):

```
ItemService.useify.clear();
```

Because you cleared all middleware the previous example, the data will not have a timestamp property next time you receive it from the server:

```
var peopleService = new ItemService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.fetchItem( "d4119c4f-31e9-4fd0-9fc4-6af1d6e36c8e" ).execute().then( function (
melton ) {

  melton.should.have.not.a.property( "timestamp" );
  done();

} ).fail( done );
```

3.3.17 Working with Event Emitters

You can extend items with event emitters:

```
var peopleService = new ItemService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.fetchItem( "d4119c4f-31e9-4fd0-9fc4-6af1d6e36c8e" ).execute().then( function (
melton ) {

  melton.on( "save", function ( error ) {

    done();

  } );

  melton.age = 40;
  melton.save().execute();

} ).fail( done );
```

3.3.18 ItemService and Validation

ItemService does not provide any client-side validation. The following example changes the ID of the item to an invalid GUID and then triggers a save. The client will allow the save to execute because there is no validation. However, when the server receives the request with invalid data, it will not resolve and the promise will fail:

```
var peopleService = new ItemService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.fetchItem( "d4119c4f-31e9-4fd0-9fc4-6af1d6e36c8e" ).execute().then( function (
melton ) {

  melton.ItemID = "invalid guid";
  melton.save().execute().then( function ( badMelton ) {

    should( badMelton ).not.exist;

  } ).fail( function ( error ) {

    error.should.be.an.instanceOf( Error );
    done();

  } );

} ).fail( done );
```

3.4 Using the RESTful API for the ItemService

3.4.1 Authentication

Login

You use this method to authenticate users. It sets the .ASPXAUTH cookie. This method only responds over HTTPS.

Verb	POST
URL	/auth/login

JavaScript example:

```
var xhr = new XMLHttpRequest();
xhr.open("POST", "http://<your server>/sitecore/api/ssc/auth/login");
xhr.setRequestHeader("Content-Type", "application/json");
xhr.onreadystatechange = function () {
  if (this.readyState == 4) {
    alert('Status: '+this.status+'\nHeaders:
'+JSON.stringify(this.getAllResponseHeaders())+'\nBody: '+this.responseText);
  }
};
xhr.send("{ \n    \"domain\": \"sitecore\", \n    \"username\": \"admin\", \n    \"password\":
\"b\" \n}");
```

The ItemService sends 200 (OK) when successful and 403 (Forbidden) when the login did not succeed.

Logout

You use this method to log out of Sitecore. It removes the .ASPXAUTH cookie.

Verb	POST
URL	/auth/logout

JavaScript example:

```
var xhr = new XMLHttpRequest();
xhr.open("POST", "http://<your server>/sitecore/api/ssc/auth/logout");
xhr.onreadystatechange = function () {
  if (this.readyState == 4) {
    alert('Status: '+this.status+'\nHeaders:
'+JSON.stringify(this.getAllResponseHeaders())+'\nBody: '+this.responseText);
  }
};
xhr.send(null);
```

The ItemService sends 200 (OK) when successful and 403 (Forbidden) when the logout did not succeed.

3.4.2 Retrieve an Item by ID

You use this to retrieve a single Sitecore item that you specify by its ID.

Verb	GET
URL	/item/{id}?database&language&version&includeStandardTemplateFields&includeMetadata&fields

The URL has these parameters:

Name	Description	Details
id	Specify the `id` of the Sitecore item to retrieve.	guid, required example: `110D559F-DEA5-42EA-9C1C-8A5DF7E70EF9`
database	Specify the database to retrieve the item from.	string, optional example: `core` default: context database for the logged in user
language	Select a language. You can use `all` as a wildcard.	string, optional example: `ja-JP` default: context language for the user that is logged in
version	Select the version of the item to retrieve.	string, optional example: `1` default: latest version
includeStandardTemplateFields	If true, the standard template fields are part of the data that is retrieved.	bool, optional default: `false`
includeMetadata	If true, the metadata is part of the data retrieved.	bool, optional default: `false`
fields	Specify the names of the fields to retrieve in a comma-separated list.	string, optional example: `ItemId,ItemName,TemplateName` default: all fields

JavaScript example:

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "http://<your server>/sitecore/api/ssc/item/110D559F-DEA5-42EA-9C1C-
8A5DF7E70EF9");
xhr.onreadystatechange = function () {
  if (this.readyState == 4) {
    alert('Status: '+this.status+'\nHeaders:
'+JSON.stringify(this.getAllResponseHeaders())+'\nBody: '+this.responseText);
  }
};
xhr.send(null);
```

The response could be:

```
200 (OK)
Accept: application/json
Content-Type: application/json
{
    "ItemID": "110d559f-dea5-42ea-9c1c-8a5df7e70ef9",
    "ItemName": "Home",
    "ItemPath": "/sitecore/content/Home",
    "ParentID": "0de95ae4-41ab-4d01-9eb0-67441b7c2450",
    "TemplateID": "76036f5e-cbce-46d1-af0a-4143f9b557aa",
    "TemplateName": "Sample Item",
    "CloneSource": null,
    "ItemLanguage": "en",
    "ItemVersion": "1",
    "Title": "Sitecore",
    "Text": "\r\n\t\t<p>Welcome to Sitecore</p>\r\n"
}
```

Errors can give one of these responses:

- 400 (Bad Request – this indicates that the request is not accepted by the server. It can be that a parameter is invalid. It also indicates that the request should not repeated.)
- 403 (Forbidden – the request is not permitted for security reasons.)
- 404 (Not Found – the item does not exist or you do not have access to the item.)

3.4.3 Retrieve an Item by Content Path

You use this method to retrieve a single Sitecore item that you specify by its content path.

Verb	GET
URL	/item/?path={path}?database&language&version&includeStandardTemplateFields&includeMetadata&fields

The URL has these parameters:

Name	Description	Details
path	Specify the path to the item in the Sitecore content tree.	string, required example: `/sitecore/content/Home`
database	Specify the database to retrieve the item from.	string, optional example: `core` default: context database for the logged in user
language	Select a language. You can use `all` as a wildcard.	string, optional example: `ja-JP` default: context language for the user that is logged in
version	Select the version of the item to retrieve.	string, optional example: `1` default: latest version
includeStandardTemplateFields	If true, the standard template fields are part of the data that is retrieved.	bool, optional default: `false`
includeMetadata	If true, the metadata is part of the data retrieved.	bool, optional default: `false`

Name	Description	Details
fields	Specify the names of the fields to retrieve in a comma-separated list.	string, optional example: `ItemId,ItemName,TemplateName` default: all fields

JavaScript example:

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "http://<your
server>/sitecore/api/ssc/item/?path=%2Fsitecore%2Fcontent%2FHomedatabase");
xhr.onreadystatechange = function () {
  if (this.readyState == 4) {
    alert('Status: '+this.status+'\nHeaders:
'+JSON.stringify(this.getAllResponseHeaders())+'\nBody: '+this.responseText);
  }
};
xhr.send(null);
```

The response is the same as when you retrieve the item by ID.

3.4.4 Retrieve the Children of an Item

You use this to retrieve the children of a Sitecore item that you specify by its ID.

Verb	GET
URL	/item/{id}/children?database&language&version&includeStandardTemplateFields&includeMetadata&fields

The URL has these parameters:

Name	Description	Details
id	Specify the `id` of the Sitecore items to retrieve.	guid, required example: `110D559F-DEA5-42EA-9C1C-8A5DF7E70EF9`
database	Specify the database to retrieve the items from.	string, optional example: `core` default: context database for the logged in user
language	Select a language. You can use `all` as a wildcard.	string, optional example: `ja-JP` default: context language for the user that is logged in
version	Select the version of the items to retrieve.	string, optional example: `1` default: latest version
includeStandardTemplateFields	If true, the standard template fields are part of	bool, optional default: `false`

Name	Description	Details
	the data that is retrieved.	
includeMetadata	If true, the metadata is part of the data retrieved.	bool, optional default: `false`
fields	Specify the names of the fields to retrieve in a comma-separated list.	string, optional example: `ItemId,ItemName,TemplateName` default: all fields

JavaScript example:

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "http://<your server>/sitecore/api/ssc/item/110D559F-DEA5-42EA-9C1C-
8A5DF7E70EF9/children");
xhr.onreadystatechange = function () {
  if (this.readyState == 4) {
    alert('Status: '+this.status+'\nHeaders:
'+JSON.stringify(this.getAllResponseHeaders())+'\nBody: '+this.responseText);
  }
};
xhr.send(null);
```

The response is similar to the response you get when you retrieve a single item.

3.4.5 Create an Item

You use this method to create a new Sitecore item.

Verb	POST
URL	/item/{path}?database&language

The URL has these parameters:

Name	Description	Details
path	Specify the path to the place in the Sitecore content tree where the item is created.	string, required example: `sitecore/content/home`
database	Specify the database to retrieve the items from.	string, optional example: `core` default: context database for the logged in user
language	Select a language. You can use `all` as a wildcard.	string, optional example: `ja-JP` default: context language for the user that is logged in

JavaScript example:

```
var xhr = new XMLHttpRequest();
```

```
xhr.open("POST", "http://<your server>/sitecore/api/ssc/item/sitecore%2Fcontent%2Fhome ");
xhr.setRequestHeader("Content-Type", "application/json");
xhr.onreadystatechange = function () {
  if (this.readyState == 4) {
    alert('Status: '+this.status+'\nHeaders:
'+JSON.stringify(this.getAllResponseHeaders())+'\nBody: '+this.responseText);
  }
};
xhr.send("{ \n    \"ItemName\": \"Home\", \n    \"TemplateID\": \"76036f5e-cbce-46d1-af0a-
4143f9b557aa\", \n    \"Title\": \"Sitecore\", \n    \"Text\": \"\\r\\n\\t\\t\u003Cp\u003EWelcome
to Sitecore\u003C/p\u003E\\r\\n\" \n}");
```

If the Sitecore creates the item, you get a response similar to this:

```
201 (Created)
Location: /item/0727f965-2338-43cc-bd88-5071ad3f7a12?database=master
```

The ID of the new item is part of the “Location”.

In case of errors, the following responses are possible:

- 400 (Bad Request)
- 403 (Forbidden)
- 404 (Not Found)

3.4.6 Edit an Item

Use this method to edit an item. You can update field values, the item name, and move the item – all in one HTTP request.

Verb	PATCH
URL	/item/{id}?database&language&version

The URL has these parameters:

Name	**Description**	**Details**
id	Specify the `id` of the Sitecore item you want to edit.	guid, required example: `110D559F-DEA5-42EA-9C1C-8A5DF7E70EF9`
database	Specify the database the item is in.	string, optional example: `core` default: context database for the logged in user
language	Specify a language selector. You can use `all` as a wildcard.	string, optional example: `ja-JP` default: context language for the logged in user
version	Specify the version of the item you want to edit.	string, optional example: `1` default: latest version

JavaScript example:

```
var xhr = new XMLHttpRequest();
```

```
xhr.open("PATCH", "http://<your server>/sitecore/api/ssc/item/110D559F-DEA5-42EA-9C1C-
8A5DF7E70EF9");
xhr.setRequestHeader("Content-Type", "application/json");
xhr.onreadystatechange = function () {
  if (this.readyState == 4) {
    alert('Status: '+this.status+'\nHeaders:
'+JSON.stringify(this.getAllResponseHeaders())+'\nBody: '+this.responseText);
  }
};
xhr.send("{ \n    \"ParentID\": \"b974fd33-c72e-4bae-a2da-b94fe44f3d6b\", \n
\"ItemName\":\"Home Renamed\", \n    \"Title\":\"Sitecore Modified\" \n}");
```

You get one of the following responses:

- 204 (No Content – this is the response when the request is successful.)
- 400 (Bad Request – this indicates that the request is not accepted by the server. It can be that a parameter is invalid. It also indicates that the request should not repeated.)
- 403 (Forbidden – the request is not permitted for security reasons.)
- 404 (Not Found – the item does not exist or you do not have access to the item.)

3.4.7 Delete an Item

Use this method to delete an item.

Verb	DELETE
URL	/item/{id}?database&language

The URL has these parameters:

Name	Description	Details
id	Specify the `id` of the Sitecore item you want to delete.	guid, required example: `110D559F-DEA5-42EA-9C1C-8A5DF7E70EF9`
database	Specify the database the item is in.	string, optional example: `core` default: context database for the logged in user
language	Specify a language selector. You can use `all` as a wildcard.	string, optional example: `ja-JP` default: context language for the logged in user

JavaScript example:

```
var xhr = new XMLHttpRequest();
xhr.open("DELETE", "http://<your server>/sitecore/api/ssc/item/110D559F-DEA5-42EA-9C1C-
8A5DF7E70EF9");
xhr.onreadystatechange = function () {
  if (this.readyState == 4) {
    alert('Status: '+this.status+'\nHeaders:
'+JSON.stringify(this.getAllResponseHeaders())+'\nBody: '+this.responseText);
  }
};
xhr.send(null);
```

You get one of the following responses:

- 204 (No Content – this is the response when the request is successful.)
- 400 (Bad Request – this indicates that the request is not accepted by the server. It can be that a parameter is invalid. It also indicates that the request should not repeated.)
- 403 (Forbidden – the request is not permitted for security reasons.)
- 404 (Not Found – the item does not exist or you do not have access to the item.)

3.4.8 Run a Stored Query

You use this method to run ("execute") a query that is stored in a Sitecore item (a "query definition item".)

Verb	GET
URL	/item/{id}/query?pageSize&page&database&includeStandardTemplateFields&fields

The URL has these parameters:

Name	Description	Details
id	Specify the `id` of the Sitecore item that contains the definition of the query to run.	guid, required example: `110D559F-DEA5-42EA-9C1C-8A5DF7E70EF9`
pageSize	Specify the number of results the service returns in the HTTP response.	integer, optional default: `10`
page	Specify the page number in the result set of pages that the service shows.	integer, optional default: `0`
database	Specify the database the item is in.	string, optional example: `core` default: context database for the logged in user
includeStandardTemplateFields	If true, the standard template fields are part of the data that is retrieved.	bool, optional default: `false`
fields	Specify the names of the fields to retrieve in a comma-separated list.	string, optional example: `ItemId,ItemName,TemplateName` default: all fields

JavaScript example:

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "http://<your server>/sitecore/api/ssc/item/110D559F-DEA5-42EA-9C1C-
8A5DF7E70EF9/query");
```

```
xhr.onreadystatechange = function () {
  if (this.readyState == 4) {
    alert('Status: '+this.status+'\nHeaders:
'+JSON.stringify(this.getAllResponseHeaders())+'\nBody: '+this.responseText);
  }
};
xhr.send(null);
```

The response could look like this:

```
200 (OK)
Accept: application/json
Content-Type: application/json
{
    "TotalCount": 100,
    "TotalPage": 50,
    "Links": [
        {
            "Href": "http://<your
server>/sitecore/api/ssc/item/query?includeStandardTemplateFields=False&fields=ItemID%2CItemName&
page=1&database=core&query=%2Fsitecore%2F%2F*",
            "Rel": "nextPage",
            "Method": "GET"
        }
    ],
    "Results": [
        {
            "ItemID": "31056d46-2faa-4ea2-8759-be93eae10001",
            "ItemName": "client"
        },
        {
            "ItemID": "e9a53290-8618-43ec-9a4b-7da2af424800",
            "ItemName": "Your Apps"
        }
    ]
}
```

If your request does not succeed, you can get one of these responses:

- 400 (Bad Request – this indicates that the request is not accepted by the server. It can be that a parameter is invalid. It also indicates that the request should not repeated. You get this response when if the query definition item does not exist.)
- 403 (Forbidden – the request is not permitted for security reasons.)

3.4.9 Run a Sitecore Search

You use this method to run a Sitecore search.

Verb	GET
URL	/item/?term&pageSize&page&database&language&includeStandardTemplateFields&fields&sorting&facet

The URL has these parameters:

Name	**Description**	**Details**
term	Specify the text to search for.	string, required example: `Home`
pageSize	Specify the number of results the service returns in the	integer, optional default: `10`

Name	Description	Details
	HTTP response.	
page	Specify the page number in the result set of pages that the service shows.	integer, optional default: `0`
database	Specify the database the item is in.	string, optional example: `core` default: context database for the logged in user
language	Specify a language selector. You use `all` as a wildcarc.	string, optional example: `ja-JP` default: context language for the logged in user
includeStandardTemplateFields	If true, the standard template fields are part of the data that is retrieved.	bool, optional default: `false`
fields	Specify the names of the fields to retrieve in a comma-separated list.	string, optional example: `ItemId,ItemName,TemplateName` default: all fields
sorting	Specify a pipe-separated list of fields to sort on. The first character of each value specifies the sort order: a (ascending) or d (descending)	string, optional example: `aTemplateName|dItemId`
facet	Specify the name and value of a facet you want to use to restrict search results	string, optional example: `_templatename|condition`

JavaScript example:

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "http://<your
server>/sitecore/api/ssc/item/?term&pageSize&page&database&language&includeStandardTemplateFields
&fields&sorting&facet");
xhr.onreadystatechange = function () {
  if (this.readyState == 4) {
    alert('Status: '+this.status+'\nHeaders:
'+JSON.stringify(this.getAllResponseHeaders())+'\nBody: '+this.responseText);
  }
};
xhr.send(null);
```

The response could look like this:

```
200 (OK)
Accept: application/json
```

```
Content-Type: application/json
{
    "Facets": [
        {
            "Name": "_templatename",
            "Values": [
                {
                    "Name": "condition",
                    "AggregateCount": 15,
                    "Link": {
                        "Href": "http://<your
server>/sitecore/api/ssc/item/search?includeStandardTemplateFields=False&fields=ItemName%2CTempla
teName&term=sitecore&facet=_templatename%7Ccondition",
                        "Rel": "_templatename|condition",
                        "Method": "GET"
                    }
                }
            ]
        }
    ],
    "TotalCount": 15,
    "TotalPage": 8,
    "Links": [
        {
            "Href": "http://<your
server>/sitecore/api/ssc/item/search?includeStandardTemplateFields=False&fields=ItemName%2CTempla
teName&page=1&term=sitecore&facet=_templatename%7Ccondition",
            "Rel": "nextPage",
            "Method": "GET"
        }
    ],
    "Results": [
        {
            "ItemName": "Country",
            "TemplateName": "Condition"
        },
        {
            "ItemName": "Page was Visited",
            "TemplateName": "Condition"
        }
    ]
}
```

If your request does not succeed, you can get one of these responses:

- 400 (Bad Request)
- 403 (Forbidden)
- 503 (Service Unavailable)

3.4.10 Run a Stored Sitecore Search

You use this method to run (“execute”) a Sitecore that that is stored in a Sitecore item (“search definition item”.) The search definition item contains default values for things like root item form the search, tenplate type, and so forth. You pass the search term itself in the URL.

Verb	GET
URL	/item/{id}/search?term&pageSize&page&database

The URL has these parameters:

Name	Description	Details
Id	Specify the `id` of the search definition item.	guid, required example: `110D559F-DEA5-42EA-9C1C-8A5DF7E70EF9`
term	Specify the text to search for.	string, required example: `Home`
pageSize	Specify the number of results the service returns in the HTTP response.	integer, optional default: `10`
page	Specify the page number in the result set of pages that the service shows.	integer, optional default: `0`
database	Specify the database the item is in.	string, optional example: `core` default: context database for the logged in user

JavaScript example:

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "<your server>/sitecore/api/ssc/item/110D559F-DEA5-42EA-9C1C-
8A5DF7E70EF9/search?term&pageSize&page&database");
xhr.onreadystatechange = function () {
  if (this.readyState == 4) {
    alert('Status: '+this.status+'\nHeaders:
'+JSON.stringify(this.getAllResponseHeaders())+'\nBody: '+this.responseText);
  }
};
xhr.send(null);
```

The response could look like this:

```
200 (OK)
Accept: application/json
Content-Type: application/json
{
    "Facets": [
        {
            "Name": "_templatename",
            "Values": [
                {
                    "Name": "condition",
                    "AggregateCount": 15,
                    "Link": {
                        "Href": "http://<your
server>/sitecore/api/ssc/item/search?includeStandardTemplateFields=False&fields=ItemName%2CTempla
teName&term=sitecore&facet=_templatename%7Ccondition",
                        "Rel": "_templatename|condition",
                        "Method": "GET"
                    }
                }
            ]
        }
    ],
    "TotalCount": 15,
    "TotalPage": 8,
    "Links": [
        {
            "Href": "http://<your
server>/sitecore/api/ssc/item/search?includeStandardTemplateFields=False&fields=ItemName%2CTempla
teName&page=1&term=sitecore&facet=_templatename%7Ccondition",
            "Rel": "nextPage",
            "Method": "GET"
        }
    ],
    "Results": [
        {
            "ItemName": "Country",
            "TemplateName": "Condition"
        },
        {
            "ItemName": "Page was Visited",
            "TemplateName": "Condition"
        }
    ]
}
```

If your request does not succeed, you can get one of these responses:

- 400 (Bad Request – this indicates that the request is not accepted by the server. It can be that a parameter is invalid. It also indicates that the request should not repeated. You get this response when if the query definition item does not exist.)
- 403 (Forbidden – the request is not permitted for security reasons.)
- 503 (Service Unavailable)

Chapter 4

The EntityService

This chapter describes the EntityService. It has this content:

- Introduction
- How to Create an EntityService
- EntityService Validation
- The EntityService and CORS
- EntityService Metadata Exchange
- Other EntityService features
- Using the EntityService from a SPEAK component
- Using the EntityService from JavaScript
- Using the RESTful API for the EntityService

4.1 Introduction

You can use the EntityService in three ways:

- You can use it through the EntityDataSource in SPEAK applications.
- You can the EntityService from client-side JavaScript.
- You can use the RESTful API to access your business objects directly.

Remember that Sitecore does not provide any business objects. It is up to you to implement the business objects for the EntityService, but the EntityService implementation provides scaffolding that makes it easier for you to implement the business objects you need.

4.2 How to Create an EntityService

This section shows how you create an EntityService that can interact with a custom business object and persist data on the server.

4.2.1 Steps to Create an EntityService

You need to do three things to create an EntityService:

1. Create a class that represents your business object and add validation attributes, as you require. This class must derive from `Sitecore.Services.Model.EntityIdentity`.
2. Create a class that handles the persisting of your business object. This class must implement the `Sitecore.Services.Core.IRepository<T>` interface.
3. Create a controller class derived from Sitecore.Services.Infrastructure.Services.EntityService where T is the business object type that you created in step 1. You must decorate this class with the `[ServicesController]` attribute and it should have a constructor with no parameters.

This example shows a services controller definition:

```
[ServicesController]
[EnableCors(origins: "*", headers: "*", methods: "*")]
public class ExampleController : EntityService<SimpleData>
{
    public ExampleController(IRepository<SimpleData> entityRepository)
        : base(entityRepository)
    {
    }

    public ExampleController()
        : this(new ExampleRepository())
    {
    }
}
```

Note: the EnableCors attribute is optional.

4.2.2 Persisting an EntityService

The EntityService design makes it possible for you to separate concerns between the controller class (that delivers the business object to the server over HTTP) and the repository class (that implements the persistence logic to store the business object).

You do not need to tie the repository class to Sitecore. It is possible to define an EntityService that uses the JavaScript and HTTP plumbing code to move business objects to the server, and then stores these objects in a third-party database or other subsystem.

How to Store Business Objects in Sitecore

The Sitecore.Services.Contrib project provides a base class you can use when you want to store business objects in the Sitecore content tree. This class gives you a starting point for implementing your own custom repository classes.

Follow these steps:

1. Create your custom repository class, and derive it from `Sitecore.Services.Contrib.Data.SitecoreItemRepository<T>`.

2. Implement the following abstract and virtual methods to provide the code to map between your business object properties and the fields of an item:

   ```
   protected abstract T CreateModelFrom(Item item); protected abstract
   string GetItemName(T entity); protected virtual void UpdateFields(T
   entity, Item item)
   ```

   Other interesting methods are:

   ```
   protected virtual bool IsMatch(T entity, Item item)
   ```

Note*:*
The `Sitecore.Services.Contrib.Data.SitecoreItemRepository<T>` class operates on Sitecore Items in the Master database.

You specify the Sitecore Template type and content root to store the items under as constructor parameters when creating the `Sitecore.Services.Contrib.Data.SitecoreItemRepository<T>` derived type:

```
protected override Category CreateModelFrom(Item item) protected override string GetItemName(T
entity) protected override void UpdateFields(T entity, Item item)

where T : Sitecore.Services.Core.Model.EntityIdentity
Member of
```

The contrib project is here:

https://github.com/Sitecore-Community/Sitecore.Services.Contrib

4.2.3 Adding Custom Methods to the Service

You can add custom action methods to your controller class. They will be selected when requests match the `{namespace}/{controller}/{id}/{action}` route.

4.3 EntityService Validation

The EntityService provides validation features that you use to enforce custom business rules and protect the server from potentially unsafe input.

The validation mechanism employed by the EntityService uses types defined in the System.ComponentModel.DataAnnotations namespace. You define business objects as POCO classes, and optionally apply attributes to the public properties to define the custom validation requirements of the data type.

Validation takes place on the client prior to it submitting data to the server, and it takes place on the server when the server receives the data as well. The server emits metadata in response to a HTTP OPTIONS request to notify the client JavaScript library about the validators defined for the specified business object.

4.3.1 Basic Validators

Sitecore.Services.Client provides the following validators:

- Required
- StringLength
- RegularExpression

The following code snippet shows how you can apply attributes to the public properties of a business object to support data validation:

```
public class SimpleEntity : EntityIdentity
{
  [Required]
  [StringLength(10, ErrorMessage = "Names should be between 1 and 10 characters")]
  public string Name { get; set; }
}
```

4.3.2 Extend Validation

You can extend the validation mechanism, and you can define new validators and apply these to business object types.

In addition to the basic set of validators in Sitecore.Services.Client, there is a contrib project. This project contains an example validator extension that supports checking email addresses. You can find this project at https://github.com/Sitecore-Community/Sitecore.Services.Contrib.

Defining a new Validator type

You can define a new validator by providing a server-side implementation for the type and a client-side validation script.

Server-side Implementation

1. Define a validator by creating a class that derives from `System.ComponentModel.DataAnnotations.ValidationAttribute` and implement your custom validation logic.

   The MSDN article How to: Customize Data Field Validation in the Data Model Using Custom Attributes contains useful information.

2. Define a metadata emitter by creating a class that derives from `Sitecore.Services.Core.MetaData.ValidationMetaDataBase<T>` where `<T>` is your validator type. You can override the virtual method `Describe(ValidationAttribute attribute, string filedName)` to customize the metadata that is emitted in response to an OPTIONS request.

   For example:

```
public class RegularExpressionMetaData :
ValidationMetaDataBase<RegularExpressionAttribute>
{
    public RegularExpressionMetaData()
        : base("regex")
    {
    }

    public override Dictionary<string, object> Describe(ValidationAttribute attribute,
string fieldName)
    {
        var metadata = base.Describe(attribute, fieldName);

        var regularExpressionAttribute = (RegularExpressionAttribute)attribute;
        metadata.Add("param", regularExpressionAttribute.Pattern);
        return metadata;
    }
}
```

Sitecore.Services.Client uses the `AssemblyScannerValidationMetaDataProvider` implementation of `IValidationMetaDataProvider` by default. On application startup, this class scans assemblies in the *bin* folder of your web site for types that provide validation metadata.

If you have specified validation attributes on business objects and no corresponding type that provides metadata for the validation is found, a warning is written in the Sitecore log when the attribute is encountered and the EntityService ignores the attribute.

Client Implementation

The JavaScript library that interacts with the EntityService needs to know how to handle all the validators that are in the site for client-side validation to work. When you add a custom validator, you need to supply a client-side implementation of the validation logic.

You put the client-side validation logic in a JavaScript file in the folder specified by the `Sitecore.Extensions.Validation.FolderPath` configuration setting. The default setting is `\sitecore\shell\client\Services\Assets\lib\extensions\validators`.

When the EntityService JavaScript library first meets a custom validator in the service metadata response, it will make a call to the MetaDataScript controller (sitecore/api/ssc/script/metadata) to retrieve the JavaScript file that contains the client-side logic for that validation attribute.

4.4 The EntityService and CORS

Browser security prevents a web page from making AJAX requests to another domain. This restriction is called "the same-origin policy." However, there are some situations where you need to let other sites call your web API.

Cross Origin Resource Sharing (CORS) is a W3C standard that allows a server to relax the same-origin policy. When you use CORS, a server can explicitly allow some cross-origin requests while rejecting others.

For more information, see *Enabling Cross-Origin Requests in ASP.NET Web API*.

4.4.1 How to Configure CORS

The Sitecore.Services.Client Services package registers support for CORS in `Sitecore.Services.Infrastructure.Web.Http.ServicesHttpConfiguration.ConfigureServices` (the `initialize` pipeline invokes this):

```
config.EnableCors();
```

4.4.2 How to Enable CORS for an EntityService Controller

You enable CORS by adding the EnableCors attribute to a controller class and then specifying the origins, headers and methods parameters as needed.

For example, this TestController sets wildcard values for all of the resource restriction parameters:

```
[ServicesController]
[EnableCors(origins: "*", headers: "*", methods: "*")]
public class TestController : EntityService<SimpleData>
```

In production environments, we recommend that you use a more restrictive definition of what can to access resources.

4.5 EntityService Metadata Exchange

The JavaScript that manages calls to the EntityService on the server needs to have an understanding of the properties of the entity class that it will handle. The JavaScript layer will be able to perform client-side validation of data assigned to the entity when it knows the type of the C# entity class and what validators are present on its properties.

The EntityService handler helps with this metadata exchange by listening for HTTP OPTIONS requests. When it finds an OPTIONS request, it will respond with a JSON-formatted HTTP response that describes the HTTP actions the service provides as well as a detailed description of the entity type the service operates on.

Before it makes any calls to the action endpoints of an EntityService, the JavaScript code makes a single HTTP OPTIONS call to receive the metadata for the service from the server.

This is an example of the metadata:

```
/**
 * Example of the metadata which indicates how to call.
 */
var MetdataAboutHowToCall = {
  "actions": {
    "GET": [
        {
          "FetchEntities": {
            "returnType": "Blog.Model.Blog[]",
            "properties": {}
          }
        },
        {
          "FetchEntity": {
            "returnType": "Blog.Model.Blog",
            "properties":{
              "key":"id",
              "datatype":"Guid"
            }
          }
        }
      ],
      "POST": {
        "CreateEntity": {
          "returnType": "void",
          "properties": {
            "key": "entity",
            "datatype": "Blog.Model.Blog"
          }
        }
      },
      "PUT": {
        "UpdateEntity": {
          "returnType": "void",
          "properties": {
            "key": "entity",
            "datatype": "Blog.Model.Blog"
          }
        }
      },
      "DELETE": {
        "Delete": {
          "returnType": "void",
          "properties": {
            "key": "entity",
            "datatype": "Blog.Model.Blog"
          }
        }
      }
    }
  };
The metadata also contains information about the Entity that can be sent to the server..

var entityMetadata = {
    "entity": {
      "key": "Id",
      "properties": [
        {
          "key": "Name",
          "datatype": "string",
          "validators": []
        },
        {
          "key": "Authors",
          "datatype": [
            [
              {
                "key": "Name",
                "datatype": "string",
                "validators": []
              },
              {
                "key": "Address",
                "datatype": [
                  {
                    "key": "Postcode",
                    "datatype": "string",
                    "validators": []
                  }
                ],
                "validators":[]
              }
            ]
          ],
          "validators":[]
        },
        {
          "key":"Created",
          "datatype":"datetime",
          "validators":[]
        },
        {
          "key":"State",
          "datatype":"number",
          "validators":[]
        },
        {
          "key":"Id",
          "datatype":"string",
          "validators":[]
        },
        {
          "key":"Url",
          "datatype":"string",
          "validators":[]
        }
      ]
    }
  };
```

EntityServiceJS sanitizes your data based on the metadata the server sends.

4.5.1 Action Endpoints

The metadata returned by the OPTIONS request includes an actions object in the JSON response. This object has details about the method and associated method signatures for each of the HTTP verbs that the endpoint exposes.

For example:

```
"actions": {
    "GET": [
        {
            "FetchEntities": {
                "returnType": "entityType[]",
                "properties": {}
            }
        },
        {
            "FetchEntity": {
                "returnType": entityType",
                "properties": {
                    "key": "id",
                    "datatype": "Guid"
                }
            }
        }
    ],
    "POST": {
        "CreateEntity": {
            "returnType": "void",
            "properties": {
                "key": "entity",
                "datatype": "entityType"
            }
        }
    },
    "PUT": {
        "UpdateEntity": {
            "returnType": "void",
            "properties": {
                "key": "entity",
                "datatype": "entityType"
            }
        }
    },
    "DELETE": {
        "Delete": {
            "returnType": "void",
            "properties": {
                "key": "entity",
                "datatype": "entityType"
            }
        }
    }
}
```

where `entityType` is the C# type name of the EntityService.

4.5.2 Entity Metadata

The OPTIONS request returns metadata that includes an `entity` object in the JSON response. This `entity` object provides details of the mapped C# entity type that the EntityService handles.

For example:

```
"entity": {
    "key": "Id",
    "properties": [
        {
            "key": "Name",
            "datatype": "string",
            "validators": []
        },
        {
            "key": "Authors",
            "datatype": [
                [
                    {
                        "key": "Name",
                        "datatype": "string",
                        "validators": [
                            {
                                "validatorName": "required",
                                "errorMessage": "Value is required"
                            },
                            {
                                "validatorName": "string",
                                "errorMessage": "Must be less than 50 characters in length",
                                "param": [
                                    0,
                                    50
                                ]
                            }
                        ]
                    },
                    {
                        "key": "Address",
                        "datatype": [
                            {
                                "key": "Postcode",
                                "datatype": "string",
                                "validators": []
                            }
                        ],
                        "validators": []
                    }
                ]
            ],
            "validators": []
        },
        {
            "key": "Id",
            "datatype": "string",
            "validators": []
        },
        {
            "key": "Url",
            "datatype": "string",
            "validators": []
        }
    ]
}
```

Note
You must derive all C# entity types that the EntityService handles from `Sitecore.Services.Core.Model.EntityIdentity`.

4.5.3 .NET to Javascript Type Mapping

Sitecore.Services.Client maps .NET types to JavaScript data types. This is the mapping translation for simple types:

```
.NET Type          Javascript Data Type
--------------------------------------
string             string
bool               boolean
int                number
float              number
double             number
long               number
DateTime           datetime
Guid               guid
Enum               number
```

Note: `guid` is not a JavaScript data type but the EntityService JavaScript layer understand that it is an identity type and it is treats it accordingly.

4.5.4 Support for Non-Primitive Types

The metadata for the entity description supports these non-primitive types:

- Arrays
- List< T > Generic collections
- IEnumerable< T >
- Classes and structs

Note that an entity cannot contain a property of a type that is derived from `Sitecore.Services.Core.Model.EntityIdentity`.

4.6 Other EntityService features

4.6.1 Promises

Almost all asynchronous calls (typically `execute()`) return a promise. EntityService uses the “q” module to handle promises:

```
var peopleService = new EntityService( {
  url: "/sitecore/api/ssc/people"
} );

var fetchQuery = peopleService.fetchEntity( "951c3e2e-02e8-4bbc-bbc8-e69ada95e670" );
var fetchQueryPromise = fetchQuery.execute();

fetchQueryPromise.should.have.a.property( "then" );
fetchQueryPromise.should.have.a.property( "fail" );
```

4.6.2 Middleware

You can use middleware with the EntityService with the sc-useify module. You can only inject middleware after a request has resolved. This gives you the opportunity to modify it just before the promise is resolved.

```
EntityService.use( function ( data, next ) {

  data.timestamp = new Date().getTime();
  next( null, data );

} );
```

Because you have integrated middleware in the previous example, the next time data is received from the server, a timestamp property is added to the data:

```
var peopleService = new EntityService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.fetchEntity( "951c3e2e-02e8-4bbc-bbc8-e69ada95e670" ).execute().then( function (
melton ) {

  melton.should.have.a.property( "timestamp" );
  done();

} ).fail( done );
```

You can clear all middleware workers, or you can clear by key:

```
EntityService.useify.clear();
```

Because you cleared all middleware in the previous example, the next time data is received from the server there will not be a timestamp property on the data.

```
var peopleService = new EntityService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.fetchEntity( "951c3e2e-02e8-4bbc-bbc8-e69ada95e670" ).execute().then( function (
melton ) {

  melton.should.have.not.a.property( "timestamp" );
  done();

} ).fail( done );
```

4.6.3 Event Emitters

All entity objects have the ability to trigger and register for events. Sitecore.Services.Client uses a component called Event Emitters for this functionality. You can find more information here: https://github.com/component/emitter.

4.7 Using the EntityService from a SPEAK component

You can use the EntityService from the EntityDataSource SPEAK component. This component is described in the SPEAK Component Reference.

4.8 Using the EntityService from JavaScript

The EntityService is a standalone XHR (XMLHttpRequest) library for creating, fetching, saving and deleting Sitecore Entities with many built in utilities and helpers to assist in the data transaction between the front end and back end.

```
var peopleService = new EntityService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.createEntity.should.be.a.type( "function" );
peopleService.fetchEntity.should.be.a.type( "function" );
peopleService.fetchEntities.should.be.a.type( "function" );
peopleService.loadMetadata.should.be.a.type( "function" );
```

The structure of an Entity object is based on metadata (a “schema”). The EntityService will request this metadata only once from the server, using the OPTIONS request.

Example:

1. An EntityService is instantiated.
2. When the EntityService is asked to execute any server related request (createEntity, fetchEntity etc), the supplied URL is hit with an OPTIONS method. All subsequent requests are queued until the server responds with a valid metadata object.
3. After the metadata object is returned, it is attached to context EntityService and all entities can be validated and sanitized based on this metadata.

The following code examples are based on this metadata:

```
var metadata = {
  "entity": {
    "key": "ItemID",
    "properties": [ {
      "key": "id",
      "datatype": "number"
    }, {
      "key": "ItemID",
      "datatype": "guid"
    }, {
      "key": "isActive",
      "datatype": "boolean"
    }, {
      "key": "balance",
      "datatype": "string"
    }, {
      "key": "picture",
      "datatype": "string"
    }, {
      "key": "age",
      "datatype": "number"
    }, {
      "key": "name",
      "datatype": "string"
    }, {
      "key": "gender",
      "datatype": "string"
    }, {
      "key": "company",
      "datatype": "string"
    }, {
      "key": "email",
      "datatype": "email"
    }, {
      "key": "phone",
      "datatype": "string"
    }, {
      "key": "address",
      "datatype": "string"
    }, {
      "key": "about",
      "datatype": "string"
    }, {
      "key": "registered",
      "datatype": "date"
    }, {
      "key": "latitude",
      "datatype": "number"
    }, {
      "key": "longitude",
      "datatype": "number"
    }, {
      "key": "subscribed",
      "datatype": "string"
    }, {
      "key": "children",
      "datatype": [ {
        "properties": [ {
          "key": "name",
          "datatype": "string"
        } ]
      } ]
    }, {
      "key": "tags",
      "datatype": [ "string" ]
    } ]
  }
};
```

4.8.1 The EntityService.Entity

`Entity` represents an Entity client side

Constructor

`EntityService.Entity ( sanitizedData, entityServiceSchema, options )`

Parameters:

- `sanitizedData`**:** Object. An object that meets has been validated by the schema
- `entityServiceSchema`**:** Object. The entity schema
- `options`**:** Object. The entity options

Returns:

`Entity`: An Entity

4.8.2 Creating Entities

You create an entity by passing an object, setting the path in the Sitecore content tree where you want the item to be created, and then you call the execute method:

```
var peopleService = new EntityService( {
  url: "/sitecore/api/ssc/people"
} );

var aNewGuy = {
  name: "David",
  isActive: "true",
  gender: "male"
};

peopleService.createEntity( aNewGuy ).then( function ( david ) {

  david.should.be.an.instanceOf( EntityService.Entity )};
  david.isNew.should.be.false;
  david.ItemID.should.not.be.empty;
  david.name.should.eql( "David" );
  david.isActive.should.eql( true );
  david.gender.should.eql( "male" );

  /**
   * ... and because not all of the key/values for this `Entity` were given based on the
   * `peopleService` end point metadata, they will still be added to the `Entity`. Their default
   * values are also based on the metadata.
   */

  david.should.have.a.property( "id", null );
  david.should.have.a.property( "balance", null );
  david.should.have.a.property( "picture", null );
  david.should.have.a.property( "age", null );
  david.should.have.a.property( "company", null );
  david.should.have.a.property( "email", null );
  david.should.have.a.property( "phone", null );
  david.should.have.a.property( "address", null );
  david.should.have.a.property( "about", null );
  david.should.have.a.property( "registered" );
  david.should.have.a.property( "latitude", null );
  david.should.have.a.property( "longitude", null );
  david.should.have.a.property( "subscribed", null );
  david.should.have.a.property( "children", null );
  david.should.have.a.property( "tags", null );

  done();

} ).fail( done );
```

You can create a dirty entity without having to make a call to the server. You do this by not giving an object to the createEntity method. All dirty entities are considered new. You can check this with the `isNew` property.

```
var peopleService = new EntityService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.createEntity().then( function ( david ) {

  david.isNew.should.be.true;
  done();

} ).fail( done );
```

4.8.3 Fetching a Single Entity

You fetch a single Entity with the fetchEntity method, giving the entity id/guid. fetchEntity returns a Query so you need to call execute.

```
var peopleService = new EntityService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.fetchEntity( "951c3e2e-02e8-4bbc-bbc8-e69ada95e670" ).execute().then( function (
cooley ) {

  cooley.name.should.eql( "Queen Cooley" );

  done();

} ).fail( done );
```

4.8.4 Fetching Multiple Entities

You fetch all entities based on the given URL with the fetchEntities method.

```
var peopleService = new EntityService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.fetchEntities().execute().then( function ( people ) {

  people.should.be.an.Array.with.a.lengthOf( 3 );

  people[ 0 ].should.be.an.instanceOf( EntityService.Entity );
  people[ 1 ].should.be.an.instanceOf( EntityService.Entity );
  people[ 2 ].should.be.an.instanceOf( EntityService.Entity );

  people[ 0 ].isValid().should.be.ok;
  people[ 1 ].isValid().should.be.ok;

  /*
   * The data of people[ 2 ] has been intentionally made invalid
   */
  people[ 2 ].isValid().should.not.be.ok;

  done();

} ).fail( done );
```

4.8.5 Saving Entities

You must fetch or create an Entity before you can save it.

```
var peopleService = new EntityService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.fetchEntity( "951c3e2e-02e8-4bbc-bbc8-e69ada95e670" ).execute().then( function (
cooley ) {

  cooley.should.be.an.instanceOf( EntityService.Entity );
  cooley.name = "Mrs Queen Cooley";

  cooley.save().then( function ( savedCooley ) {

    savedCooley.name.should.eql( "Mrs Queen Cooley" );
    done();

  } ).fail( done );

} ).fail( done );
```

Each time you save an Entity, it will trigger a ‘save’ event that you can listen for: `( entity.on(’save’, callback) ).`

4.8.6 Destroying Entities

You must fetch or create an Entity before you can destroy it.

```
var peopleService = new EntityService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.fetchEntity( "951c3e2e-02e8-4bbc-bbc8-e69ada95e670" ).execute().then( function (
cooley ) {

  cooley.should.be.an.instanceOf( EntityService.Entity );

  cooley.destroy().then( function () {

    done();

  } ).fail( done );

} ).fail( done );
```

4.8.7 Dirty Entities

A dirty entity is an entity you create in browser memory without having to save it first. This is useful, for example, if you are waiting for user input.

```
var peopleService = new EntityService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.createEntity().then( function ( aNewDirtyEntity ) {

  /* At this point `aNewDirtyEntity` has not been saved */
  aNewDirtyEntity.should.be.an.instanceOf( EntityService.Entity );
  aNewDirtyEntity.name = "Queen";

  /* After modifying the dirty item, now we save it */
  aNewDirtyEntity.save().then( function () {

    done();

  } ).fail( done );

} );
```

4.8.8 isNew

You check if an Entity has not been saved (it is “dirty”) with the isNew property:

```
var peopleService = new EntityService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.createEntity().then( function ( guy ) {

  guy.isNew.should.be.true;
  done();

} ).fail( done );
```

When the origin of an Entityis is the server, isNew is always false.

```
var peopleService = new EntityService( {
  url: "/sitecore/api/ssc/people"
} );
```

```
peopleService.createEntity( {
  name: "guy"
} ).then( function ( guy ) {

  /*
   * `guy` is technically from the server because we are creating a new `Entity` by giving an
   * object to the `createEntity` method.
   */
  guy.isNew.should.be.false;
  done();

} ).fail( done );
```

4.8.9 How to Retrieve Raw JSON

You can have situations where you need to retrieve the data stored in an Entity as raw JSON without any of the additional methods and properties. The following method is used internally to retrieve the data that is sent to the server:

```
var peopleService = new EntityService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.fetchEntity( "951c3e2e-02e8-4bbc-bbc8-e69ada95e670" ).execute().then( function (
queen ) {

  var queenAsJson = queen.json();

  queen.should.be.an.instanceOf( EntityService.Entity );
  queenAsJson.should.be.an.instanceOf( Object );
  queenAsJson.should.not.be.an.instanceOf( EntityService.Entity );

  done();

} ).fail( done );
```

4.8.10 Trackable Entities

You can extend an Item with tracking to add additional functionality, for example:

- Automatically save when a property changes.
- Call hasChanged() to see if the Entity has changed.
- Call revertChanges() to revert property values.

To add tracking to your Entity, set the option trackable to true.

```
var peopleService = new EntityService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.fetchEntity( "d4119c4f-31e9-4fd0-9fc4-6af1d6e36c8e" ).option( "trackable", true
).execute().then( function ( melton ) {

  melton.option( "trackable" ).should.be.true;

  done();

} ).fail( done );
```

4.8.11 Automatically Save

When you turn trackable on, changes are automatically saved. You can listen to the save event using the “emitter on” or “once”.

```
var peopleService = new EntityService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.fetchEntity( "d4119c4f-31e9-4fd0-9fc4-6af1d6e36c8e" ).option( "trackable", true
).execute().then( function ( melton ) {

  /* Listening to the `save` event `once`. You can also use `on` here to continuously listen to
the `save` event. */
  melton.once( "save", function ( error ) {

    done();

  } );

  melton.name = "Melton the Magnificent";

} ).fail( done );
```

4.8.12 hasChanged()

When you turn trackable on, the method hasChanged is added so you can check if an Entity has changed:

```
var peopleService = new EntityService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.fetchEntity( "d4119c4f-31e9-4fd0-9fc4-6af1d6e36c8e" ).option( "trackable", true
).execute().then( function ( melton ) {

  melton.hasChanged().should.be.false;
  melton.name = "Melton the Magnificent";
  melton.hasChanged().should.be.true;

  done();

} ).fail( done );
```

The hasChanged method does not return true if a value changes between null, undefined and ``.

```
var peopleService = new EntityService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.fetchEntity( "d4119c4f-31e9-4fd0-9fc4-6af1d6e36c8e" ).option( "trackable", true
).execute().then( function ( melton ) {

  melton.hasChanged().should.be.false;
  melton.subscribed = "";
  melton.hasChanged().should.be.false;
  melton.subscribed = null;
  melton.hasChanged().should.be.false;
  melton.subscribed = undefined;
  melton.hasChanged().should.be.false;

  done();

} ).fail( done );
```

4.8.13 revertChanges()

When you turn trackable on, the method revertChanges is added so you can revert your changes.

```
var peopleService = new EntityService( {
  url: "/sitecore/api/ssc/people"
} );

peopleService.fetchEntity( "d4119c4f-31e9-4fd0-9fc4-6af1d6e36c8e" ).option( "trackable", true
).execute().then( function ( melton ) {

  melton.name.should.eql( "Banks Melton" );
  melton.name = "Melton the Magnificent";
  melton.name.should.eql( "Melton the Magnificent" );
  melton.hasChanged().should.be.true;
  melton.revertChanges();
  melton.hasChanged().should.be.false;
  melton.name.should.eql( "Banks Melton" );

  done();

} ).fail( done );
```

4.9 Using the RESTful API for the EntityService

You use the EntityService to create custom controller classes on the server. These classes are accessible over HTTP.

`Sitecore.Services.Infrastructure.Web.Http.DefaultRouteMapper` provides a default route to access these custom controller classes by. This route and a custom controller selector for the Web API are configured during the application startup of the web site.

4.9.1 Default Routing

A custom controller class that acts as an EntityService has to be addressable through a unique URL. You fulfill this requirement by deriving the URL from the fully qualified type name of the custom controller class in question.

The route definition that applies to EntityService classes is:

`{namespace}/{controller}/{id}/{action}`

where:

- `id` is an optional parameter
- `action` is the name of the method you want to invoke on the controller

Mapping HTTP Verbs

All EntityService endpoints respond to the following HTTP verbs:

- **GET** - maps to the `FetchEntity` (when `id` parameter is supplied) or to the `FetchEntities` (when `id` parameter not supplied) method of the EntityService
- **POST** - maps to the `CreateEntity` method of the EntityService
- **PUT** - maps to the `UpdateEntity` method of the EntityService
- **DELETE** - maps to the `Delete` method of the EntityService
- **OPTIONS** - maps to the `MetaData` method of the EntityService

Mapping Type Name to URL

Sitecore.Services.Client uses the full type name for the controller class to map incoming HTTP requests to EntityService endpoints by default.

The two route parameters that are the key to this process are `{namespace}` and `{controller}`. The non-optional part of the route definition is:

`{namespace}/{controller}`

Where:

- `{namespace}` is derived from the namespace of the class name, with "." characters converted to "-".
- `{controller}` is the class name without the 'Controller' suffix. This follows the MVC convention for controller naming.

As an example:

```
Qualified Type Name                     Url
-------------------                     ------------------

My.Namespace.ProductController    ->    my-namespace/product
```

Custom Controller Action Routing

You can add custom action methods to an EntityService controller class. Sitecore.Services.Client will automatically route requests to the appropriate action method if you supply `{id}` and `{action}` parameters in the URL of the request.

This uses the `{namespace}/{controller}/{id}/{action}` route.

The default routing configuration does not support a request that does not supply an `{id}` parameter (the `{namespace}/{controller}/{action}` route).

4.9.2 Configuration

The `Sitecore.Services.Client.config` include file contains the following configuration options:

Custom Route Configuration

The `Sitecore.Services.RouteMapper` setting specifies the `Sitecore.Services.Infrastructure.Web.Http.IMapRoutes` derived type that configures routes for Sitecore.Services.Client.

The default value for this setting is `Sitecore.Services.Infrastructure.Web.Http.DefaultRouteMapper, Sitecore.Services.Infrastructure`.

Service Endpoint Stem

All Entity and ItemService controllers are located under a single URL: `Sitecore/api/ssc/`.

4.9.3 Override Namespace Routing

The `ServicesControllerAttribute` class provides a constructor that takes a `UniqueName` parameter.

You use the `UniqueName` parameter to specify the `{namespace}` and `{controller}` route data values for a controller explicitly. You can override the default “type name to URL mapping mechanism” described earlier this way.

The following examples show how the `UniqueName` parameter value maps to a URL that will access a controller:

```
Unique Name                     URL
-------------                   ------------------
product                  ->     product
company.product          ->     company.product
company/product          ->     company.product
long.company.product     ->     long-company/product
long/company/product     ->     long-company/product
long.company/product     ->     long-company/product
```

A unique name that does not provide both the `{namespace}` and the `{controller}` parts will not be serviced by the `{namespace}/{controller}/{action}` route, and can generate HTTP 404 responses.

For a high-level overview of routing, see Routing in ASP.NET Web API.