Dealing with Relationships, Partial Updates, and Other Complexities

In the previous chapter, after a brief introduction to the concepts of authentication and authorization, we added security to the task-management service. We began by applying an authorization filter to secure the AddTask method, and complemented this by implementing a custom message handler supporting Basic authentication. After that, we implemented several scenarios (continuing with the theme of security) to further develop our application's functional capabilities and to demonstrate various ASP.NET Web API features (e.g., global exception handling of custom exceptions, scoping of filter attributes, serialization control, async filters). We wrapped things up by adding support for token-based security.

In this chapter we will continue building out the RESTful API we designed in Chapter 3.We will deal with:

* Relationships - we'll manage task assignees
* Partial updates - we'll update an existing task
* Input validation - we'll validate the request to update an existing task
* Context-sensitive hypermedia - we'll add links to the task service model in the response
* Paging of results - we'll get all tasks, and use a query string to control paging of results

And, naturally, we will highlight several great ASP.NET Web API features along the way. Now let's get started…

# Relationships

We dealt with a simple relationship in the Securing Non-Resource API Operations section of Chapter 6; namely, the relationship between Task and Status. Now we will add support for a more complicated relationship; namely, the relationship between Task and User. Table 7-1, which is excerpted from Table 3-3, summarizes what we will implement in this section.

Table 7-1. A List of Task Operations

|  |  |  |
| --- | --- | --- |
| URI | Verb | Description |
| /api/tasks/123/users | PUT | Replaces all users on the specified task; returns the updated task in the response |
| /api/tasks/123/users | DELETE | Deletes all users from the specified task; returns the updated task in the response |
| /api/tasks/123/users/456 | PUT | Adds the specified user (e.g., 456) as an assignee on the task; returns the updated task in the response |
| /api/tasks/123/users/456 | DELETE | Deletes the specified user from the assignee list; returns the updated task in the response |

We'll stick with our bottom-up approach of adding dependencies first, and the first dependency we'll implement is a query processor (implement as follows):

IUpdateTaskQueryProcessor Interface

using System.Collections.Generic;

using WebApi2Book.Data.Entities;

namespace WebApi2Book.Data.SqlServer.QueryProcessors

{

public interface IUpdateTaskQueryProcessor

{

Task ReplaceTaskUsers(long taskId, IEnumerable<long> userIds);

Task DeleteTaskUsers(long taskId);

Task AddTaskUser(long taskId, long userId);

Task DeleteTaskUser(long taskId, long userId);

}

}

UpdateTaskQueryProcessor Class

using System.Collections.Generic;

using System.Linq;

using NHibernate;

using WebApi2Book.Data.Entities;

using WebApi2Book.Data.Exceptions;

namespace WebApi2Book.Data.SqlServer.QueryProcessors

{

public class UpdateTaskQueryProcessor : IUpdateTaskQueryProcessor

{

private readonly ISession \_session;

public UpdateTaskQueryProcessor(ISession session)

{

\_session = session;

}

public Task ReplaceTaskUsers(long taskId, IEnumerable<long> userIds)

{

var task = GetValidTask(taskId);

UpdateTaskUsers(task, userIds, false);

\_session.SaveOrUpdate(task);

return task;

}

public Task DeleteTaskUsers(long taskId)

{

var task = GetValidTask(taskId);

UpdateTaskUsers(task, null, false);

\_session.SaveOrUpdate(task);

return task;

}

public Task AddTaskUser(long taskId, long userId)

{

var task = GetValidTask(taskId);

UpdateTaskUsers(task, new[] {userId}, true);

\_session.SaveOrUpdate(task);

return task;

}

public Task DeleteTaskUser(long taskId, long userId)

{

var task = GetValidTask(taskId);

var user = task.Users.FirstOrDefault(x => x.UserId == userId);

if (user != null)

{

task.Users.Remove(user);

\_session.SaveOrUpdate(task);

}

return task;

}

public virtual Task GetValidTask(long taskId)

{

var task = \_session.Get<Task>(taskId);

if (task == null)

{

throw new RootObjectNotFoundException("Task not found");

}

return task;

}

public virtual User GetValidUser(long userId)

{

var user = \_session.Get<User>(userId);

if (user == null)

{

throw new ChildObjectNotFoundException("User not found");

}

return user;

}

public virtual void UpdateTaskUsers(Task task, IEnumerable<long> userIds, bool appendToExisting)

{

if (!appendToExisting)

{

task.Users.Clear();

}

if (userIds != null)

{

foreach (var user in userIds.Select(GetValidUser))

{

if (!task.Users.Contains(user))

{

task.Users.Add(user);

}

}

}

}

}

}

Dependency Configuration (add to bottom of NinjectConfigurator.AddBindings)

container.Bind<IUpdateTaskQueryProcessor>().To<UpdateTaskQueryProcessor>().InRequestScope();

Note that three of the four interface methods (ReplaceTaskUsers, DeleteTaskUsers, and AddTaskUser) have a similar pattern of implementation. First, a Task is fetched from the database. Then, the Users collection is updated appropriately via the UpdateTaskUsers method. Finally, the updated Task is persisted back to the database with its updated Users associations.

UpdateTaskUsers is a helper method that either replaces the existing Users collection in its entirety or appends to it, as directed by the calling method to produce the desired result. The foreach loop ensures that duplicate users aren't added to the specified task. This protects the idempotence (remember that term from Chapter 2?) of the operations that rely upon this method.

The only other non-trivial method is DeleteTaskUser. This method supports the idempotence of the "remove the user from the task" operation by first ensuring that the specified user is still associated with the task before trying to break the association.

The next dependency to implement is also simple; in fact, simpler than the query processor we just discussed. Implement it as follows:

ITaskUsersMaintenanceProcessor Interface

using System.Collections.Generic;

using WebApi2Book.Web.Api.Models;

namespace WebApi2Book.Web.Api.MaintenanceProcessing

{

public interface ITaskUsersMaintenanceProcessor

{

Task ReplaceTaskUsers(long taskId, IEnumerable<long> userIds);

Task DeleteTaskUsers(long taskId);

Task AddTaskUser(long taskId, long userId);

Task DeleteTaskUser(long taskId, long userId);

}

}

TaskUsersMaintenanceProcessor Class

using System.Collections.Generic;

using WebApi2Book.Common.TypeMapping;

using WebApi2Book.Data.SqlServer.QueryProcessors;

using WebApi2Book.Web.Api.Models;

namespace WebApi2Book.Web.Api.MaintenanceProcessing

{

public class TaskUsersMaintenanceProcessor : ITaskUsersMaintenanceProcessor

{

private readonly IAutoMapper \_autoMapper;

private readonly IUpdateTaskQueryProcessor \_queryProcessor;

public TaskUsersMaintenanceProcessor(IUpdateTaskQueryProcessor queryProcessor, IAutoMapper autoMapper)

{

\_queryProcessor = queryProcessor;

\_autoMapper = autoMapper;

}

public Task ReplaceTaskUsers(long taskId, IEnumerable<long> userIds)

{

var taskEntity = \_queryProcessor.ReplaceTaskUsers(taskId, userIds);

return CreateTaskResponse(taskEntity);

}

public Task DeleteTaskUsers(long taskId)

{

var taskEntity = \_queryProcessor.DeleteTaskUsers(taskId);

return CreateTaskResponse(taskEntity);

}

public Task AddTaskUser(long taskId, long userId)

{

var taskEntity = \_queryProcessor.AddTaskUser(taskId, userId);

return CreateTaskResponse(taskEntity);

}

public Task DeleteTaskUser(long taskId, long userId)

{

var taskEntity = \_queryProcessor.DeleteTaskUser(taskId, userId);

return CreateTaskResponse(taskEntity);

}

public virtual Task CreateTaskResponse(Data.Entities.Task taskEntity)

{

var task = \_autoMapper.Map<Task>(taskEntity);

return task;

}

}

}

Dependency Configuration (add to bottom of NinjectConfigurator.AddBindings)

container.Bind<ITaskUsersMaintenanceProcessor>().To<TaskUsersMaintenanceProcessor>().InRequestScope();

TaskUsersMaintenanceProcessor is so trivial that it requires little discussion. Some items to note before moving on:

* The implementation is necessarily dependent upon the WebApi2Book2.Data and WebApi2Book.Web.Api.Models projects. Therefore, it would have been inappropriate to push the logic down into the query processor, which should have no knowledge of WebApi2Book.Web.Api.Models.
* The full implementation, available in our GitHub repository, includes logic in CreateTaskResponse that adds hypermedia links to the Task response. This is the main reason why CreateTaskResponse was broken out into a separate method. In this section we're focusing on relationships, so that detail was omitted here. We'll get to links later in the chapter.

And now to bring it all together, the TaskUsersController class (implement as follows):

using System.Collections.Generic;

using System.Web.Http;

using WebApi2Book.Common;

using WebApi2Book.Web.Api.MaintenanceProcessing;

using WebApi2Book.Web.Api.Models;

using WebApi2Book.Web.Common;

using WebApi2Book.Web.Common.Routing;

namespace WebApi2Book.Web.Api.Controllers.V1

{

[ApiVersion1RoutePrefix("tasks")]

[UnitOfWorkActionFilter]

[Authorize(Roles = Constants.RoleNames.SeniorWorker)]

public class TaskUsersController : ApiController

{

private readonly ITaskUsersMaintenanceProcessor \_taskUsersMaintenanceProcessor;

public TaskUsersController(ITaskUsersMaintenanceProcessor taskUsersMaintenanceProcessor)

{

\_taskUsersMaintenanceProcessor = taskUsersMaintenanceProcessor;

}

[Route("{taskId:long}/users", Name = "ReplaceTaskUsersRoute")]

[HttpPut]

public Task ReplaceTaskUsers(long taskId, [FromBody] IEnumerable<long> userIds)

{

var task = \_taskUsersMaintenanceProcessor.ReplaceTaskUsers(taskId, userIds);

return task;

}

[Route("{taskId:long}/users", Name = "DeleteTaskUsersRoute")]

[HttpDelete]

public Task DeleteTaskUsers(long taskId)

{

var task = \_taskUsersMaintenanceProcessor.DeleteTaskUsers(taskId);

return task;

}

[Route("{taskId:long}/users/{userId:long}", Name = "AddTaskUserRoute")]

[HttpPut]

public Task AddTaskUser(long taskId, long userId)

{

var task = \_taskUsersMaintenanceProcessor.AddTaskUser(taskId, userId);

return task;

}

[Route("{taskId:long}/users/{userId:long}", Name = "DeleteTaskUserRoute")]

[HttpDelete]

public Task DeleteTaskUser(long taskId, long userId)

{

var task = \_taskUsersMaintenanceProcessor.DeleteTaskUser(taskId, userId);

return task;

}

}

}

Again, another very simple class, at least at first glance. However, note the route prefix attribute, the various route attributes, the HttpPut and HttpDelete attributes, the authorization filter, and the unit of work attribute. Through the use of the declarative attributes (which we've discussed in previous chapters), a lot of cross-cutting concerns are taken care of for us so that we don't need to clutter the controller code with them. These ensure that request gets routed to the correct controller and action method, that the request is restricted to users with the required role, and that the request is processed in the context of a unit of work to ensure database updates are handled properly. It turns that there's a lot more going on in here than one would think by a simple line count!

Let's test it out to ensure it's working properly. We'll follow the order of operations listed in Table 7-1, so let's start by assigning a couple of users to a task. We'll use our favorite task (#17), and we'll use bhogg's credentials because we know he's authorized (he's a senior worker):

Replace Task Users Request (abbreviated)

PUT http://localhost:61589/api/v1/tasks/17/users/ HTTP/1.1

Content-Type: application/json; charset=utf-8

Authorization: Basic YmhvZ2c6aWdub3JlZA==

[2,3]

Note how we only need to specify the user IDs, not entire users, in the request message body to add them as assignees. And now for the response…

Replace Task Users Response (abbreviated)

HTTP/1.1 200 OK

Content-Type: application/json; charset=utf-8

{"TaskId":17,"Subject":"Fix something important","StartDate":"2014-05-13T00:52:34","DueDate":null,"CreatedDate":"2014-05-10T19:02:52","CompletedDate":null,"Status":{"StatusId":2,"Name":"In Progress","Ordinal":1},"Assignees":[{"UserId":2,"Username":"jbob","Firstname":"Jim","Lastname":"Bob","Links":[]},{"UserId":3,"Username":"jdoe","Firstname":"John","Lastname":"Doe","Links":[]}],"Links":[]}

Excellent, Jim Bob and John Doe have been added as assignees to the task! We have finally associated User objects with Task objects. Because the operation is idempotent you should be able to send the request multiple times and get the same response; in fact, this applies to all operations in this section.

Now let's remove all assignees from the task:

Remove Task Users Request (abbreviated)

DELETE http://localhost:61589/api/v1/tasks/17/users HTTP/1.1

Content-Type: application/json; charset=utf-8

Authorization: Basic YmhvZ2c6aWdub3JlZA==

Remove Task Users Response (abbreviated)

HTTP/1.1 200 OK

Content-Type: application/json; charset=utf-8

{"TaskId":17,"Subject":"Fix something important","StartDate":"2014-05-13T00:52:34","DueDate":null,"CreatedDate":"2014-05-10T19:02:52","CompletedDate":null,"Status":{"StatusId":2,"Name":"In Progress","Ordinal":1},"Assignees":[],"Links":[]}

As expected, we're back to the situation where task #17 has no assignees. Next, let's assign a single user to the task. Note the assignee's ID is in the URL instead of the message body this time:

Add Task User Request (abbreviated)

PUT http://localhost:61589/api/v1/tasks/17/users/2 HTTP/1.1

Content-Type: application/json; charset=utf-8

Authorization: Basic YmhvZ2c6aWdub3JlZA==

Add Task User Response (abbreviated)

HTTP/1.1 200 OK

Content-Type: application/json; charset=utf-8

{"TaskId":17,"Subject":"Fix something important","StartDate":"2014-05-13T00:52:34","DueDate":null,"CreatedDate":"2014-05-10T19:02:52","CompletedDate":null,"Status":{"StatusId":2,"Name":"In Progress","Ordinal":1},"Assignees":[{"UserId":2,"Username":"jbob","Firstname":"Jim","Lastname":"Bob","Links":[]}],"Links":[]}

This looks good; user #2 (Jim Bob) is back on the task. However, to complete our testing, go ahead and remove him as follows by specifying his ID in the URL:

Remove Task User Request (abbreviated)

DELETE http://localhost:61589/api/v1/tasks/17/users/2 HTTP/1.1

Content-Type: application/json; charset=utf-8

Authorization: Basic YmhvZ2c6aWdub3JlZA==

Remove Task User Response (abbreviated)

HTTP/1.1 200 OK

Content-Type: application/json; charset=utf-8

{"TaskId":17,"Subject":"Fix something important","StartDate":"2014-05-13T00:52:34","DueDate":null,"CreatedDate":"2014-05-10T19:02:52","CompletedDate":null,"Status":{"StatusId":2,"Name":"In Progress","Ordinal":1},"Assignees":[],"Links":[]}

And it looks like everything is working properly. Jim Bob is no longer assigned to the task.

We're ready to move on to the topic of partial updates, but before we do, please re-send the "Replace Task Users" request so that the Task data is a little bit more interesting going forward!

# Partial Update of a Task Using PUT/PATCH

Recalling our discussion of HTTP verbs from Chapter 2, we know that, by convention, a PUT operation is used to replace the corresponding resource in its entirety. This is why we designed our operation to update a task as follows in Chapter 3 (excerpted from Table 3-3):

Table 7-2. Update a Task

|  |  |  |
| --- | --- | --- |
| URI | Verb | Description |
| /api/tasks/123 | PUT | Updates the specified task; returns the updated task in the response |

However, often there are often times when it is desirable to apply a partial modification to a resource rather than replace the entire resource. It is for this reason the PATCH method was created, and in this section we will implement the ability to partially update a task.

Let's begin with the query processor, and this one should look familiar; it's the UpdateTaskQueryProcessor. Add the highlighted lines to the interface and class, as shown:

IUpdateTaskQueryProcessor Interface Modifications

using System.Collections.Generic;

using WebApi2Book.Data.Entities;

using PropertyValueMapType = System.Collections.Generic.Dictionary<string, object>;

namespace WebApi2Book.Data.SqlServer.QueryProcessors

{

public interface IUpdateTaskQueryProcessor

{

Task GetUpdatedTask(long taskId, PropertyValueMapType updatedPropertyValueMap);

…

UpdateTaskQueryProcessor Class Modifications

using System.Collections.Generic;

using System.Linq;

using NHibernate;

using WebApi2Book.Data.Entities;

using WebApi2Book.Data.Exceptions;

using PropertyValueMapType = System.Collections.Generic.Dictionary<string, object>;

namespace WebApi2Book.Data.SqlServer.QueryProcessors

{

public class UpdateTaskQueryProcessor : IUpdateTaskQueryProcessor

{

private readonly ISession \_session;

public UpdateTaskQueryProcessor(ISession session)

{

\_session = session;

}

public Task GetUpdatedTask(long taskId, PropertyValueMapType updatedPropertyValueMap)

{

var task = GetValidTask(taskId);

var propertyInfos = typeof(Task).GetProperties();

foreach (var propertyValuePair in updatedPropertyValueMap)

{

propertyInfos.Single(x => x.Name == propertyValuePair.Key)

.SetValue(task, propertyValuePair.Value);

}

\_session.SaveOrUpdate(task);

return task;

}

…

The using directive is used to define an alias to the Dictionary<string,object> type. It is syntactic sugar, and nothing more. Each element in an instance of the PropertyValueMapType is used to map a property name (string) to the corresponding property value (object). Note that the scope of a using directive is limited to the file in which it appears, which is why it appers in both files.

The real work is done in the GetUpdatedTask method. This method accepts two parameters: taskId, which uniquely identifies the Task to update, and updatedPropertyValueMap, which contains one element per property to be modified. Note that in an extreme case, the updatedPropertyValueMap could contain an element for every updateable Task property, and in this case the operation would function more like a PUT than a PATCH. Keep this in mind; we'll revisit it once we get to the controller.

Now let's analyze the GetUpdatedTask logic. First, the Task is fetched from the database. Next, each property on the Task that is represented by an element in the updatedPropertyValueMap is updated using reflection. Finally, the updated Task is persisted.

Gee, that was easy. It seems like the hard part would be computing the updatedPropertyValueMap, so where does that logic appear? Well, we're getting to it next. It requires a pair of dependencies, so add them as follows and then we'll discuss them:

Update an existing task. Demo using put and patch.

For example, think of the wasted network traffic if a fully-serialized Task was required in the request message body to simply update the Subject property of the task.

# Validation Using an Action Filter

Use the attr to validate a task update request

# Hypermedia Links

We'll add the links to the Task

# Paging

Get tasks. Use query string for paging control.

# Summary