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
* Paging of results - we'll get all tasks, and use a query string to control paging of results
* 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

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 |

Let's begin by implementing the PUT operation that adds a User as an assignee to a Task. 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();

The first thing you may have noticed about our UpdateTaskQueryProcessor class is that it is doing more than supporting the ability to add a user as an assignee to a task; it's supporting all of the operations in Table 7-1 (in terms of persistence). We thought it would be easier to implement all of this simple class' functionality now rather than having to repeatedly return to add to it later.

Note that the 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.

The aforementioned UpdateTaskUsers is a helper method that either replaces the existing Users collection in its entirety or appends to it. 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

# Paging

Get tasks. Use query string for paging control.

# Partial Update of a Task Using PUT/PATCH

Update an existing task. Demo using put and patch.

# Validation Using an Action Filter

Use the attr to validate a task update request

# Hypermedia Links

We'll add the links to the Task

# Summary