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 more 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 (or fully, as it turns out) 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 a 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 fairly easy. It seems like the hard part would be computing the updatedPropertyValueMap, so where does that logic appear? Well, we'll get to that in a little while. First, we're going to deviate from our usual bottom-up approach and implement the controller. The code will be in a non-compilable state for a while because we will be referencing an undefined dependency, but this approach will make things easier to explain in the long run. Go ahead and add the following method to TasksController…

[Route("{id:long}", Name = "UpdateTaskRoute")]

[HttpPut]

[HttpPatch]

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

public Task UpdateTask(long id, [FromBody] object updatedTask)

{

var task = \_updateTaskMaintenanceProcessor.UpdateTask(id, updatedTask);

return task;

}

… and then modify the constructor to accept a new (currently undefined) dependency, as shown in the highlighted code:

…

private readonly ITaskByIdInquiryProcessor \_taskByIdInquiryProcessor;

private readonly IUpdateTaskMaintenanceProcessor \_updateTaskMaintenanceProcessor;

public TasksController(IAddTaskMaintenanceProcessor addTaskMaintenanceProcessor,

ITaskByIdInquiryProcessor taskByIdInquiryProcessor,

IUpdateTaskMaintenanceProcessor updateTaskMaintenanceProcessor)

{

\_addTaskMaintenanceProcessor = addTaskMaintenanceProcessor;

\_taskByIdInquiryProcessor = taskByIdInquiryProcessor;

\_updateTaskMaintenanceProcessor = updateTaskMaintenanceProcessor;

}

…

The UpdateTask method appears deceptively simple. But, as we discussed in the previous section, a lot of things are happening behind the scenes. One of those things is the routing; the framework will route PUT and PATCH requests to this method. This is appropriate, as we recently mentioned, because sometimes a user will actually want to replace the targeted resource (i.e., the Task identified by the id parameter) with an entirely new representation. Also, some callers are not even able to send a request containing the PATCH verb (e.g., Flash-based callers have this restriction). So, even though we're blurring the lines somewhat between PUT and PATCH, this is one of those cases where a bit of pragmatism, rather than stubbon adherence to every detail of the HTTP specification, will make things more usable for the callers.

Another thing to note is that the updatedTask parameter is of type object. With the model binding capabilities of ASP.NET Web API available, why would we want the method to accept the parsed request body as an object? For one thing, this makes partial updates possible. If the method accepted a Task containing data that the user wanted to partially update, it would also have to accept a list of property names to update. Otherwise, for example, how would the application know how to interpret a null Subject property? Would null indicate that the user desires to clear the subject, or would it indicate that he simply doesn't want to modify it? So, rather than requiring the caller to also provide a list of property names, the implementation accepts a potentially sparse representation of the Task.

Another reason for the updatedTask parameter's object type is that the framework will parse the task data from the message body and deliver it to UpdateTask in the form of JSON or XML, as determined by the request message's Content-Type header. The common type between these two representations is object; hence, the current signature.

At this point we're ready to move on to implementing the IUpdateTaskMaintenanceProcessor and one of its dependencies, IUpdateablePropertyDetector. This pair of dependencies works together to compute the updatedPropertyValueMap that we discussed earlier. IUpdateablePropertyDetector determines which properties to update, and IUpdateTaskMaintenanceProcessor uses this information to populate the updatedPropertyValueMap. We'll also need to modify the Task service model class so that it can be inspected by the IUpdateablePropertyDetector. So much to do! Let's start by adding the IUpdateablePropertyDetector as follows:

IUpdateablePropertyDetector Interface

using System.Collections.Generic;

namespace WebApi2Book.Web.Common

{

public interface IUpdateablePropertyDetector

{

IEnumerable<string> GetNamesOfPropertiesToUpdate<TTargetType>(object objectContainingUpdatedData);

}

}

JObjectUpdateablePropertyDetector Class

using System;

using System.Collections.Generic;

using System.ComponentModel.DataAnnotations;

using System.Linq;

using System.Reflection;

using Newtonsoft.Json.Linq;

namespace WebApi2Book.Web.Common

{

public class JObjectUpdateablePropertyDetector : IUpdateablePropertyDetector

{

public IEnumerable<string> GetNamesOfPropertiesToUpdate<TTargetType>(object objectContainingUpdatedData)

{

var objectDataAsJObject = (JObject)objectContainingUpdatedData;

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

var modifiablePropertyInfos = propertyInfos

.Where(x =>

{

var editableAttribute =

x.GetCustomAttributes(typeof(EditableAttribute)).FirstOrDefault() as EditableAttribute;

return editableAttribute != null && editableAttribute.AllowEdit;

}

);

var namesOfSuppliedProperties =

objectDataAsJObject.Properties().Select(x => x.Name);

return

modifiablePropertyInfos.Select(x => x.Name)

.Where(x => namesOfSuppliedProperties.Contains(x, StringComparer.InvariantCultureIgnoreCase));

}

}

}

Dependency Configuration (add to bottom of NinjectConfigurator.AddBindings)

container.Bind<IUpdateablePropertyDetector>().To<JObjectUpdateablePropertyDetector>().InSingletonScope();

The first thing you may have noted, from the class name if nothing else, is that some of this implementation is JSON-specific. We could have provided an implementation to handle both JSON and XML (hint: by examining the Content-Type header of the request message and delegating to the appropriate code), but we figured it would be a better use of our time together to try to keep the focus on ASP.NET Web API and avoid getting bogged down in those sorts of details. Besides, the basic algorithms are the same regardless of content type.

As you can see, the parsed task object enters the GetNamesOfPropertiesToUpdate method and is immediately cast to a JObject, one of the many useful types in the powerful Json.NET package - which, by the way, was automatically added when you created the project back in Chapter 4. We'll return to it in a minute.

The next thing that happens is the property metadata is calculated based on the generic parameter type (which, in this scenario TTargetType is going to arrive as Task). Then, inspecting this property metadata one property at a time, we build a list of PropertyInfo instances. Each element in the list corresponds to a property in the target class decorated with an EditableAttribute and having an AllowEdit value of true. This is important, because we only want the user to be able to update editable properties (this as a guard against overposting).

Next, we return to the JObject, from which we extract the names of the properties represented in the task fragment. We then filter this list (namesOfSuppliedProperties) against the list of modifiable properties (modifiablePropertyInfos) and return the result as a list of names of the properties to update. In this way, we have provided a flexible mechanism that can support full or partial updates in a way that is not vulnerable to overposting attacks.

Now let's decorate Task with attributes to indicate which properties are modifiable. The class should appear as follows:

using System;

using System.Collections.Generic;

using System.ComponentModel.DataAnnotations;

namespace WebApi2Book.Web.Api.Models

{

public class Task : ILinkContaining

{

private List<Link> \_links;

private bool \_shouldSerializeAssignees;

[Key]

public long? TaskId { get; set; }

[Editable(true)]

public string Subject { get; set; }

[Editable(true)]

public DateTime? StartDate { get; set; }

[Editable(true)]

public DateTime? DueDate { get; set; }

[Editable(false)]

public DateTime? CreatedDate { get; set; }

[Editable(false)]

public DateTime? CompletedDate { get; set; }

[Editable(false)]

public Status Status { get; set; }

[Editable(false)]

public List<User> Assignees { get; set; }

[Editable(false)]

public List<Link> Links

{

get { return \_links ?? (\_links = new List<Link>()); }

set { \_links = value; }

}

public void AddLink(Link link)

{

Links.Add(link);

}

public void SetShouldSerializeAssignees(bool shouldSerialize)

{

\_shouldSerializeAssignees = shouldSerialize;

}

public bool ShouldSerializeAssignees()

{

return \_shouldSerializeAssignees;

}

}

}

Note the Key attribute, which identifies the property that uniquely identifies each instance. Also, note that certain properies are not user-editable. For example, CreatedDate is set by the application; it is not user-editable. Also, Assignees is not user-editable because we want users to modify a task's assignees via the relationships API we implemented in the previous section.

Now we are ready to add the missing piece that sits between the controller and the quer processor: the IUpdateTaskMaintenanceProcessor. Implement as follows:

IUpdateTaskMaintenanceProcessor Interface

using WebApi2Book.Web.Api.Models;

namespace WebApi2Book.Web.Api.MaintenanceProcessing

{

public interface IUpdateTaskMaintenanceProcessor

{

Task UpdateTask(long taskId, object taskFragment);

}

}

UpdateTaskMaintenanceProcessor Class

using System.Linq;

using Newtonsoft.Json.Linq;

using WebApi2Book.Common.TypeMapping;

using WebApi2Book.Data.SqlServer.QueryProcessors;

using WebApi2Book.Web.Api.Models;

using WebApi2Book.Web.Common;

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

namespace WebApi2Book.Web.Api.MaintenanceProcessing

{

/// <summary>

/// Updates the specified Task.

/// </summary>

/// <remarks>

/// This implementation only supports Json. Support for other content types is

/// left as an exercise for the reader.

/// </remarks>

public class UpdateTaskMaintenanceProcessor : IUpdateTaskMaintenanceProcessor

{

private readonly IAutoMapper \_autoMapper;

private readonly IUpdateTaskQueryProcessor \_queryProcessor;

private readonly IUpdateablePropertyDetector \_updateablePropertyDetector;

public UpdateTaskMaintenanceProcessor(IUpdateTaskQueryProcessor queryProcessor, IAutoMapper autoMapper,

IUpdateablePropertyDetector updateablePropertyDetector)

{

\_queryProcessor = queryProcessor;

\_autoMapper = autoMapper;

\_updateablePropertyDetector = updateablePropertyDetector;

}

public Task UpdateTask(long taskId, object taskFragment)

{

var taskFragmentAsJObject = (JObject) taskFragment;

var taskContainingUpdateData = taskFragmentAsJObject.ToObject<Task>();

var updatedPropertyValueMap = GetPropertyValueMap(taskFragmentAsJObject, taskContainingUpdateData);

var updatedTaskEntity = \_queryProcessor.GetUpdatedTask(taskId, updatedPropertyValueMap);

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

return task;

}

public virtual PropertyValueMapType GetPropertyValueMap(JObject taskFragment, Task taskContainingUpdateData)

{

var namesOfModifiedProperties = \_updateablePropertyDetector.GetNamesOfPropertiesToUpdate<Task>(taskFragment).ToList();

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

var updatedPropertyValueMap = new PropertyValueMapType();

foreach (var propertyName in namesOfModifiedProperties)

{

var propertyValue = propertyInfos.Single(x => x.Name == propertyName).GetValue(taskContainingUpdateData);

updatedPropertyValueMap.Add(propertyName, propertyValue);

}

return updatedPropertyValueMap;

}

}

}

Dependency Configuration (add to bottom of NinjectConfigurator.AddBindings)

container.Bind<IUpdateTaskMaintenanceProcessor>().To<UpdateTaskMaintenanceProcessor>().InRequestScope();

As with JObjectUpdateablePropertyDetector, you can see that parts of this implementation are JSON-specific. As with JObjectUpdateablePropertyDetector, we could have also supported XML; however, for similar reasons as stated previously, we chose to leave that as en exercise for the motivated reader.

Let's analyze the current implementation, starting with UpdateTask. UpdateTask begins by using JObject to parse the task fragment into an actual Task instance. Next, it invokes GetPropertyValueMap, which uses the task fragment and the parsed Task instance to compute the updatedPropertyValueMap that gets passed to the query processor. As we discussed earlier, the updates actually get applied in the query processor. Finally, the automapper maps the task entity returned by the query processor into a service model object, which is then returned to the method caller (which is, in this case, the controller).

Diving down into GetPropertyValueMap, we see it first uses the IUpdateablePropertyDetector to determine the names of properties that need to be updated. Then, for each of those properties, it gets the corresponding value from the Task instance and adds the property name and value pair to the map. Finally, it returns the map to the method caller.

Now that we've completed the implementation, and answered the question about the origin of the updatedPropertyValueMap (which was asked when we were discussing UpdateTaskQueryProcessor), it's time to prove that this actually works! We'll start with a PUT, using bhogg's credentials (he's a senior worker, so he's authorized) and our favorite task (#17):

Update Task Request - PUT (abbreviated)

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

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

Authorization: Basic YmhvZ2c6aWdub3JlZA==

{"Subject":"Get a new HDMI cable",

"CreatedDate":"2011-01-01"

}

Update Task Response - PUT (abbreviated)

HTTP/1.1 200 OK

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

{"TaskId":17,"Subject":"Get a new HDMI cable","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":[]}

Very nice! Notice how the Subject was updated but the CreatedDate was not. It looks like our overpost-proof update functionality is working correctly. We'll conclude by doing something similar, only this time with a PATCH:

Update Task Request - PATCH (abbreviated)

PATCH http://localhost:61589/api/v1/tasks/17 HTTP/1.1

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

Authorization: Basic YmhvZ2c6aWdub3JlZA==

{"Subject":"Fix the compile error that broke the build",

"CompletedDate":"2011-01-01"

}

Update Task Response - PATCH (abbreviated)

HTTP/1.1 200 OK

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

{"TaskId":17,"Subject":"Fix the compile error that broke the build","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":[]}

Great, it's working perfectly; Subject was updated but CompletedDate was not. But how do we guard against callers passing in garbage via the message body? As a matter of fact, that's our next topic!

# Validation Using an Action Filter

We've made great progress, but we're still lacking input validation. What if the contents of updatedTask parameter in the TasksController class' UpdateTask method were gibberish, having nothing to do with a task? We'd be wasting valuable server resources by beginning to process such a request, only to have it fail with an exception. What we need is a way to validate the request before it even gets to the action method. Fortunately, ASP.NET Web API provides support for such validation, through the use of custom action filters.

We've implemented action filters before (e.g., the UnitOfWorkActionFilterAttribute and UserAuditAttribute classes), so some of this should look familiar. Let's begin by implementing the attribute (add it as follows), and then we'll analyze it and apply it to the UpdateTask method:

using System.Net;

using System.Net.Http;

using System.Web.Http;

using System.Web.Http.Controllers;

using System.Web.Http.Filters;

using log4net;

using Newtonsoft.Json;

using Newtonsoft.Json.Linq;

using WebApi2Book.Common.Logging;

using WebApi2Book.Web.Api.Models;

using WebApi2Book.Web.Common;

namespace WebApi2Book.Web.Api.MaintenanceProcessing

{

public class ValidateTaskUpdateRequestAttribute : ActionFilterAttribute

{

private readonly ILog \_log;

public ValidateTaskUpdateRequestAttribute()

: this(

WebContainerManager.Get<ILogManager>())

{

}

public ValidateTaskUpdateRequestAttribute(ILogManager logManager)

{

\_log = logManager.GetLog(typeof (ValidateTaskUpdateRequestAttribute));

}

public override bool AllowMultiple

{

get { return false; }

}

public override void OnActionExecuting(HttpActionContext actionContext)

{

var taskId = (long?) actionContext.ActionArguments[ActionParameterNames.TaskId];

if (!taskId.HasValue)

{

const string errorMessage = "Invalid taskId.";

\_log.Debug(errorMessage);

throw new HttpResponseException(new HttpResponseMessage(HttpStatusCode.BadRequest)

{

Content = new StringContent(errorMessage)

});

}

var taskFragment =

(JObject) actionContext.ActionArguments[ActionParameterNames.TaskFragment];

\_log.DebugFormat("{0} = {1}", ActionParameterNames.TaskFragment, taskFragment);

if (taskFragment == null)

{

const string errorMessage = "Malformed or null request.";

\_log.Debug(errorMessage);

throw new HttpResponseException(new HttpResponseMessage(HttpStatusCode.BadRequest)

{

Content = new StringContent(errorMessage)

});

}

try

{

var task = taskFragment.ToObject<Task>();

if (task.TaskId.HasValue && task.TaskId != taskId)

{

const string errorMessage = "Task ids do not match.";

\_log.Debug(errorMessage);

throw new HttpResponseException(new HttpResponseMessage(HttpStatusCode.BadRequest)

{

Content = new StringContent(errorMessage)

});

}

}

catch (JsonException ex)

{

\_log.Debug(ex.Message);

throw new HttpResponseException(new HttpResponseMessage(HttpStatusCode.BadRequest)

{

Content = new StringContent("Incorrect data in request.")

});

}

}

public static class ActionParameterNames

{

public const string TaskFragment = "updatedTask";

public const string TaskId = "id";

}

}

}

Only the OnActionExecuting method contains logic that we haven't explained previously when analyzing other action filters, so we'll focus on it. It begins by ensuring that an argument named id and of type long? is available as a parameter to be bound to the action method. (If you look at UpdateTask, you see that there is a parameter named id; this is the parameter to be matched.) If no such argument is available, the method throws an HttpResponseException to prevent the processing from continuing on to the controller action method. It can't stop processing by creating a response message - note the void return type in the method signature; it must throw an exception.

Next, the data to be bound to the updatedTask parameter (i.e., taskFragment) is examined. If no JSON-compatible data is available, then, again, an HttpResponseException is thrown to prevent processing from reaching the controller action method.

After that, taskFragment, a JObject instance, is parsed into an actual Task. If this parsing fails, then, again, an HttpResponseException is thrown to prevent processing from reaching the controller method.

Finally, the taskIds from the URL and the message body are compared. If they differ, then again - you guessed it - an HttpResponseException is thrown to prevent processing from reaching the controller method. The request is considered valid if no exceptions are thrown by OnActionExecuting.

Note that this is a JSON-specific implementation. This helps ensure that the JSON-specific processing described in the previous Partial Update of a Task Using PUT/PATCH section will be successful. As mentioned in that section, support for XML could have been provided (hint: by examining the (actionContext.Request.Content).Headers.ContentType); however, for reasons stated previously, we decided to forgo that exercise.

At this point we've implemented the action filter, but for this to be effective, we need to apply the attribute to the controller method. Therefore, update the UpdateTask method so it appears as follows:

[Route("{id:long}", Name = "UpdateTaskRoute")]

[HttpPut]

[HttpPatch]

[ValidateTaskUpdateRequest]

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

public Task UpdateTask(long id, [FromBody] object updatedTask)

{

var task = \_updateTaskMaintenanceProcessor.UpdateTask(id, updatedTask);

return task;

}

Now let's see this in action. First send a valid request to ensure we didn't break anything:

Next, send an invalid request to ensure we're getting the proper response:

Mention gaps, like the ability to null the Subject. But this gives you the idea… focus is on ASP.NET Web API.

Also, ValidateModelAttribute for AddTask

# Hypermedia Links

We'll add the links to the Task

# Paging

Get tasks. Use query string for paging control.

# Summary