Designing the Sample REST API

Thus far, you’ve learned some basic principles of the REST architecture using the HTTP protocol, and you’re now ready to start developing your task-management service. First, you’ll need to take some time to carefully build up tables of resource types, their available HTTP actions, and associated URIs. This design step is important, similar in kind to the importance of patiently and intentionally modeling a database. It pays to think it through and get it right. And, as you walk through the different resource types, you’ll begin examining some code.

You may recall from the previous chapter that a programmer by the name of Leonard Richardson created what has become known as the REST Maturity Model (RMM). This model defines a pathway for turning a more traditional Remote Procedure Call (RPC)-style API into a REST-style API. As you build your sample API, using this maturity model will help you map from something most developers know (i.e., non-REST) into something new and different (i.e., REST). You will need to be on the lookout for the natural tendency of the API to degenerate into an RPC-style API, thus falling back down the maturity model. We’ll try to draw attention to those moments where a wrong choice could send you sliding back down.

Also in this chapter, you will model a small database for storing tasks and their supporting data. You won’t spend much time doing so, as building a RESTful rather than a non-RESTful service doesn’t change your approach to database modeling. Either way, you need to store instances of your resources and their relationships.

Finally, you will walk through what we believe to be good choices for components to use in your ASP.NET Web API service implementation. Since you’re going to build a working service application, not just a trivial "Hello World" type of application, we’ll show you components such as an Object Relational Mapper (ORM), a logger, an Inversion of Control (IoC) container, a type mapper, and so on.

# Task Management Resource Types

Let’s start by thinking about some things you want the callers of the API to be able to do. Since this service is focused on task management, most of the capabilities it offers will be centered on creating, viewing, and updating tasks. Again, a domain that is simple and well understood will allow you to focus on the nondomain concepts we’re concerned about in this book—specifically, REST and the ASP.NET Web API.

First and foremost, the caller should be able to create a new task. And it should be able to do so without being required to provide anything more than a subject. Values such as start date, end date, and so on can be updated later if they’re not known at the time the task is created. When creating a new task, we will have the system create its identifier, as opposed to the caller generating a custom identifier and passing it in.

The system will provide a listing of all tasks to the caller. This listing will support pagination, because the number of tasks in the system can be large. The caller should also be able to find, update, and delete a specific existing task.

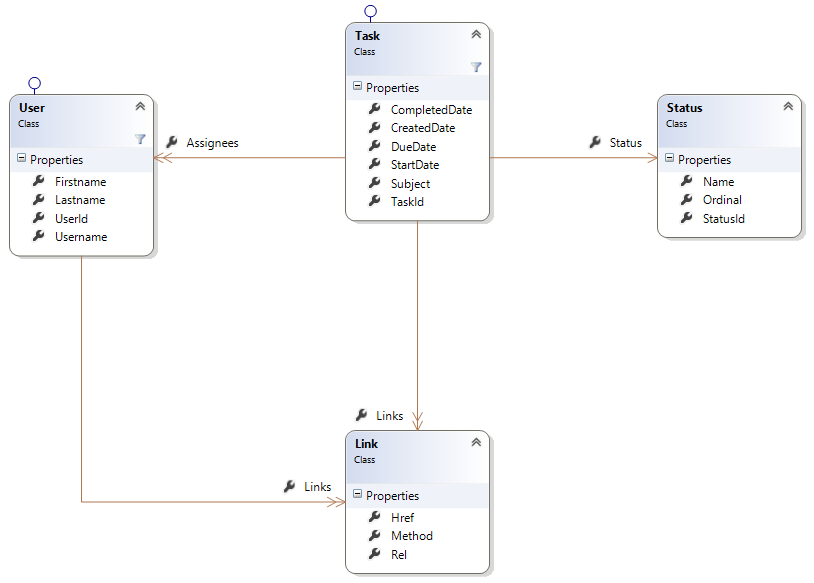
The system will need to support zero or more users as assignees to a task. Most systems dealing with tasks allow only a single user assignment, which can be an inconvenient limitation. Our requirement to support multiple user assignments to a task will make the API a little more interesting.

Speaking of users, we need to provide a listing of all users to the caller. This listing will support pagination, because the number of users in the system can be large.

Finally, to support classification of the tasks, we will provide support for the task status. We can assume that the available values for status will be configured at the time of deployment.

The task-management example is about managing tasks and highlighting features of the ASP.NET Web API, so we won’t discuss adding, updating, or deleting users or statuses.

Figure 3-1 illustrates what the resource types will look like as a class diagram in Visual Studio 2013.



**Figure 3-1.** A class diagram of resource types

Recall, however, that one of the guiding rules of the REST architecture is to avoid coupling the client to the server by sharing type definitions. So, even though we will be using classes within server code to represent the resources received from and sent to the caller, these definitions are purely internal. This is markedly different from SOAP, where a Web Services Description Language (WSDL) document very explicitly defines all service interfaces, methods and their signatures, and all message types. Indeed this SOAP agreement is a contract, and it couples the client to the definitions on the server. But in REST, you want to avoid this coupling as much as possible and do your best to keep the “contractual” elements of your service limited to those required by the REST architectural style (i.e., HTTP verbs and URIs for accessing and updating resources, and utilizing hypermedia as the engine of application state).

## Hypermedia Links

Speaking of hypermedia, you no doubt noticed the Link class in Figure 3-1, along with the associations with the other classes. Remember that we want to lead the API consumer through our application, similar to the way a user in a web browser is led through a web site with various hyperlinks and web forms. As such, each and every time you send a resource representation back to the caller, you need to give it a list of available actions (i.e., state changes).

Let’s look at the Link class in more detail:

public class Link

{

public string Rel { get; set; }

public string Href { get; set; }

public string Method { get; set; }

}

This should look familiar to you, as it is similar to the link HTML element. Indeed, you’re trying to give the user very similar information to that provided by the link element:

* **Rel:** Specifies the relationship between the resource and the resource identified in the link
* **Href:** Specifies the linked resource’s address
* **Method:** Specifies the HTTP method used to access the resource

As we discussed near the end of Chapter 2, one of the issues with using links like the one just specified is that the REST architecture doesn’t define any specific standard for building hypermedia links in an API. If you search the Internet to find some semblance of a common approach, you will find many different opinions. And as we covered in Chapter 2, Collection+JSON and HAL appear to be the leading options right now.

RMM Lookout

You might be tempted to use a set of more-specific links than just a collection of string-oriented link objects. For example, you could have Link properties for the following:

* Update
* Delete
* Assignees
* NewAssignment

But you need to remember that the RESTful service needs to look, act, and smell like a state machine. That means you must have resources moving through states via predefined state transitions. As defined by REST, your service must specify the allowed transitions for any given resource based on the current state of that resource. In other words, the available links (i.e., state transitions) will change from one call to the next, depending on what state you’re in (e.g., the state of the Task and the permissions of the current user). Therefore, it is imperative that the list of links be dynamic.

There’s another important reason for using a collection of links for the state transitions: the Single Responsibility Principle (SRP). Introduced by Robert C. Martin in 2002, the principle essentially states that a class should have only one reason to change; that is, it should only be responsible for one thing.

If you put those state transitions on your resource types, you violate SRP because now your resource definition will need to change every time you want to change any of the available state transitions. Your definition will also change if you add to or remove any transitions. Instead, the available transitions should be dictated by a separate class, not the resource type class. In other words, the rules that decide what actions the caller is allowed to take on a given resource should be external to that resource. If you keep the available transitions loose (your collection of Link objects), the service code doing the work of returning a resource can be the one to worry about creating appropriate links.

Before we get into modeling our resources against URIs and HTTP verbs, let’s quickly look at the class code for our resource types:

public class Task

{

public long? TaskId { get; set; }

public string Subject { get; set; }

public DateTime? StartDate { get; set; }

public DateTime? DueDate { get; set; }

public DateTime? CreatedDate { get; set;

public DateTime? CompletedDate { get; set; }

public Status Status { get; set; }

public List<Link> Links { get; set; }

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

}

public class Status

{

public long StatusId { get; set; }

public string Name { get; set; }

public int Ordinal { get; set; }

}

public class User

{

public long UserId { get; set; }

public string Username { get; set; }

public string Firstname { get; set; }

public string Lastname { get; set; }

public List<Link> Links { get; set; }

}

There’s nothing particularly remarkable about these types, but note that their identifiers are integers, and those identifying values will be generated by the service, not provided by the caller. Also note that taskId is nullable. The reason for this will become clear when we deal with task updates.

## Modeling the URIs and HTTP Verbs

We now want to model each resource type’s allowed HTTP verbs and associated URIs. The operations (i.e., verbs) available will vary from type to type; there is no requirement for REST-based APIs to support all of the verbs on each resource type or URI.

Let’s start with an easy one: Status. Table 3-1 illustrates that we want to support only one operation.

**Table 3-1.** A List of Status Operations

|  |  |  |
| --- | --- | --- |
| URI | Verb | Description |
| /api/statuses | GET | Gets the full list of all statuses |

We don’t need to allow the caller to modify statuses. The only requirement is to provide a method to get the list of statuses (e.g., to populate drop-down controls).

The URIs and verbs for the User resource type will be similar. The task-management service isn’t going to allow the caller to modify the users in the system. Table 3-2 illustrates the two operations we will allow on User.

**Table 3-2.** A List of User Operations

|  |  |  |
| --- | --- | --- |
| URI | Verb | Description |
| /api/users | GET | Gets the full list of all users; optionally specifies a filter |
| /api/users/123 | GET | Gets the details for a single user |

The main difference between this resource type and the Status type is that we want to allow the caller to supply a filter for limiting the list of users returned. This will be in the form of URL query string arguments. We’ll explore the details of user querying later, when we start building the service code.

OData

The /api/users URI in our task-management service will be providing limited filtering capability in the way of simple query strings. You might be tempted to allow more complex queries by supporting ANDs and ORs, parentheses, TOP, ORDERBY, and so on. However, it is for these capabilities that the Open Data Protocol (OData) exists. This protocol was created by Microsoft and a few other companies to standardize web-based data querying and updating.

Here’s what the www.odata.org web site says:

The Open Data Protocol (OData) enables the creation of REST-based data services, which allow resources, identified using Uniform Resource Identifiers (URIs) and defined in a data model, to be published and edited by Web clients using simple HTTP messages.

In fact, the ASP.NET Web API provides a simple mechanism for supporting OData with your REST service.

The downside to using OData with the Web API is that you must expose your domain model types over the wire. We’ll be taking the approach of mapping domain model types over to resource types before returning the data to the caller. This approach reduces coupling between the client and server, which is one of the salient features of a REST-based API. But we can’t do this if we want to implement an OData query interface using the built-in OData feature in ASP.NET Web API. It is for this reason, and the fact that OData is such a large topic in and of itself, that we trust our readers to explore OData on their own.

Finally, we need to define the URIs and HTTP verbs for the Task resource type. Table 3-3 shows the list of operations available for the Task.

**Table 3-3.** A List of Task Operations

|  |  |  |
| --- | --- | --- |
| URI | Verb | Description |
| /api/tasks | GET | Gets the full list of all tasks; optionally specify a filter |
| /api/tasks/123 | GET | Gets the details for a single task |
| /api/tasks/123/users | GET | Gets the users assigned to the specified task |
| /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 |
| /api/tasks | POST | Creates a new task; returns the new task in the response |
| /api/tasks/123 | PUT | Updates the specified task; returns the updated task in the response |

The relationships with users and statuses make task operations more interesting. For example, here you see something that wasn’t present in the previous resource types: using PUT and DELETE on a collection of related resources. In order to add a new assignee to a task, the caller utilizes the users collection, adding or deleting specific users one at a time. Or, optionally, the caller can use PUT or DELETE against the entire collection. According to the HTTP protocol, this will replace or delete all users associated with the task.

Tasks are also related to statuses. In this example, let’s imagine that there is a rules-based workflow that controls task status updates (i.e., updating a task’s status isn’t just a typical update operation in the "CRUD" sense of things). Instead, a series of processing steps must be executed within our service in order to change a task’s status, possibly even sending an email—thereby making this update non-idempotent. How should we handle this from a REST-based API perspective?

We’ll start by thinking about the required operations in the abstract, and then create conceptual resources to represent them. For example, we need to support the ability to begin, or “activate,” a task. And, unless the system’s prospective users are total slackers, we need to support the ability to eventually complete a task. Last, we probably should also support the ability to reopen, or “re-activate,” a task that had been marked as completed.

Did you come up with a list of conceptual resources based on that last paragraph? We came up with Task Activations, Task Completions, and Task Re-activations. Table 3-4 summarizes this.

**Table 3-4.** A List of Task Status Operations

|  |  |  |
| --- | --- | --- |
| URI | Verb | Description |
| /api/tasks/123/activations | POST | Starts, or “activates,” a task; returns the updated task in the response |
| /api/tasks/123/completions | POST | Completes a task; returns the updated task in the response |
| /api/tasks/123/reactivations | POST | Reopens, or “re-activates,” a task; returns the updated task in the response |

The situation of needing to support non–resource API operations using REST is fairly common. Having been through this little exercise with task status, you are now better prepared to deal with it on the job. It’s okay if the list of “resources” you thought of didn’t exactly match those in Table 3-4. The point is to keep thinking in terms of resources so that you don’t degenerate into an RPC API.

And that wraps up this chapter’s exploration of designing the resource types. Next, you will learn how to perform a quick modeling of the database.

# The Task-Management Data Model

In this section, we’re going to create the model for storing the task-management service data.

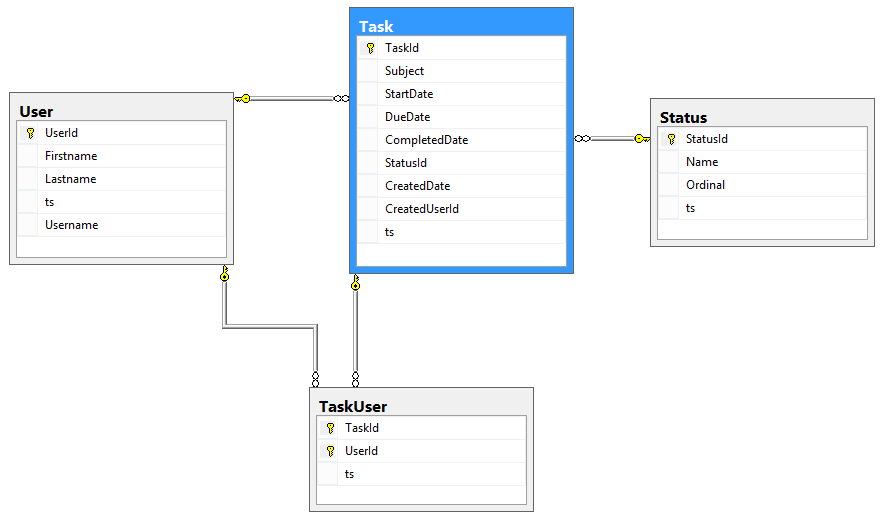
Logically, we have three categories of data to store:

* Reference data
* Tasks
* Users

Statuses are reference data. The Status table will include an identifier, a name, and an ordinal. The ordinal value can be used for sorting the data for display in drop-down or other list controls.

The task and user data is also straightforward. Note that we will use a many-to-many table to link tasks to users because a task can be associated with zero or more users, and a user can be associated with zero or more tasks.

Most of the model in Figure 3-2 looks similar to the resource types you designed earlier in this chapter. However, this model includes a column called ts for each table. As a matter of practice, it is a good idea to include a versioning column to be used for concurrency checking (i.e., checking for dirty data on update). We chose ts for the column name for a few reasons: it stands for *timestamp*, it’s short, and it typically doesn’t conflict with other column names. Later on, as we build the code, you’ll see exactly how the ts column is used to ensure proper concurrency checking.



**Figure 3-2.** Task and Reference Data Tables

You also may have noticed the CreatedUserId column in the Task table, which we’ve included to help illustrate that the model objects exposed via the API can differ from the model objects, or “entities,” used to persist the data.

At this point, we’ve designed all of the resource types, and we’ve laid out the URIs and HTTP verbs for those types. We’ve also just briefly modeled the underlying database to store our resources. Before closing out this chapter, let’s spend a bit of time choosing the various architecture components we’ll need to build our service application.

# Choosing Architecture Components

The purpose of this book is to take you from a near-zero level of experience in writing .NET services, teach you about REST and the Web API, and have you end up with a reasonably robust, simple, yet fully functional, REST-style service. As such, we feel it prudent to introduce you to some components and tools that can greatly assist you in the implementation. So, now that we’ve done most of the high-level design of the system, let’s explore some these.

## Data Access

There are quite a few options available in .NET when it comes to data access and object persistence on SQL Server. Most of these options fall into one of two categories: using the various SqlClient objects (e.g., SqlConnection, SqlDataAdapter, and SqlCommand) with stored procedures or embedded SQL, or using an Object Relational Mapper (ORM). Sometimes the two approaches are used together, but more often developers choose one or the other.

We will be using the NHibernate ORM. The fact that you can do virtually all of your data access in C#, and the natural support for the Unit of Work pattern (via the ISession) are significant benefits. This is especially appropriate for web or service applications, where you want a given call to execute within the context of a single database session and transaction.

Unit of Work and Repository Patterns

Martin Fowler introduces some extremely valuable enterprise patterns in his book, *Patterns of Enterprise Application Architecture* (Addison-Wesley, 2002). If you aren’t familiar with the definition and use-cases of Unit of Work as they apply to data access, we strongly encourage you to read up on them in Martin’s book. For a free and quick summary of the patterns, you can also visit www.martinfowler.com, where he offers some brief descriptions and diagrams of some of the patterns found in the book. Possessing a solid understanding of such data access–related patterns is key to properly managing database connections and transactions, in-memory object state, and data cache. It is also critical to maintaining testability.

## Type Mapper

A good type mapper liberates you from the tedium of manually mapping data between objects as processing flows up and down the stack. We will be using AutoMapper to map data between resources and their persistent representations, or entities. In other words, we can lean on AutoMapper to copy values from the domain or data model object to and from the corresponding—yet, slightly different—REST resource type. This allows us to easily account for differences in property names, data types, and even differences in the actual number of properties.

## IoC Container

These days, whether you’re working in .NET or in Java, not using an IoC container of some sort can almost be considered foolish. There are certainly special circumstances that might require you to manage dependencies yourself, but generally speaking, using one of the available frameworks is pretty much a no-brainer. The ASP.NET Web API provides the IDependencyResolver interface for the very purpose of resolving dependencies, and we will implement it with the Ninject container.

## Logger

If you ask 10 people for their opinion on the best logger, you will likely get 11 different answers. We’ll spare you the suspense and tell you now that we’ll be using log4net. The log4net logging framework is simple to use, provides a logger interface that can be used with IoC containers, comes with numerous options for routing and filtering, and has been used all around the world in thousands of .NET applications for many years.

## Testing Framework

The two most prominent testing frameworks for .NET are MSTest and NUnit. Both work very well, and both have their pros and cons. We tend to lean towards NUnit for its simplicity, full-featured Assert class, and available fluent interface, though MSTest also works just fine. We will be using NUnit.

## Mocking Framework

We will use Moq for the test mocking framework. It is simple, powerful, and popular.

# Summary

That wraps up the bulk of our exploration of the API design, including the SQL Server database and a selection of the most important components in your architecture.

At this point, using the modeling technique we introduced in this chapter, you should be able to properly design just about any RESTful service, complete with resource types, URIs, and HTTP verbs. You should also be aware of various tool and framework choices that can be leveraged in building a services application with ASP.NET Web API.