One of the important characteristics of the popular web APIs in use today is that they are RESTful services, or at least they are not SOAP based.

Just because I have built an HTTP service that handles XML or JSON payloads and respond to HTTP methods such as GET, POST, PUT, and DELETE, I cannot claim that I have built a RESTful service. In this chapter, we see what it takes for an HTTP service to be called RESTful. We then build our first web API, a simple Hello-World API.

What Is a RESTful Service?

Representational State Transfer (REST) is an architectural style. The term REST was introduced and defined by Roy T. Fielding in his doctoral dissertation in the year 2000. A service that conforms to the REST constraints is referred to as being RESTful. To be RESTful, a service has to conform to the following mandatory constraints.

1. **Client-server constraint**, which is based on the separation of concerns, is about separating user interface concerns from data storage concerns. Clients are not concerned with data storage, which is a concern of servers, and servers are not concerned with the user interface or user state, which are concerns of clients.

2. **Stateless constraint** is about each request being an independent self-contained unit with all the necessary information for the server to service the request without looking at anything else for the context.

3. **Cache constraint** is about the server being able to label a response as cacheable or not, so that the client handles the response appropriately from the point of view of later use.

4. **Layered constraint** is about composing the system into layers, with each layer being able to see and interact with only its immediate neighbor. A layer cannot see through its neighbor. Between the client and server, there could be any number of intermediaries—caches, tunnels, proxies, and so on.

5. **Uniform interface constraint** is about providing a uniform interface for identification of resources, manipulation of resources through representations, self-descriptive messages, and hypermedia as the engine of application state.

How can we build a service that satisfies the given constraints using the ASP.NET Web API framework? Client-server constraint is an easy one to satisfy out of the box. ASP.NET Web API is all about responding to the client request with the data, without bothering about client state or how data will be presented to the end user.

Stateless constraint can also be easily satisfied out of the box, unless something horrible is done such as using the ASP.NET session state from the web API.

ASP.NET MVC supports the OutputCache attribute that can be used to control output caching. ASP.NET Web API has no support out of the box, but it is easy to roll out our own action filter attribute. The bottom line is that the
Cache-Control response header is the lever ASP.NET Web API can use to label a response as cacheable or not. By default, Cache-Control is set to no-cache and the response is not cached. Chapter 4 covers the topic of web caching, including ETags.

Layered constraint is more along the infrastructure line—proxies, firewalls, and so on. There is nothing special that needs to be done from ASP.NET Web API to satisfy this constraint.

Uniform interface constraint includes the following four constraints and is a key factor in deciding if an HTTP service is RESTful or not.

1. Identification of resources
2. Manipulation of resources through representations
3. Self-descriptive messages
4. Hypermedia as the engine of application state (HATEOAS)

We now look at uniform interface constraint in detail through each of the four constraints.

### Identification of Resources

A resource is any data that a web API sends to its clients. Examples could be a product that your company sells, a purchase order received from a buyer, a list of employees in your company, or an individual employee in a department. In the real world, a product or an employee could be uniquely identified through an identifier, such as a product ID or an employee ID.

In the case of RESTful web services, a resource is identified by a URI. An employee with an identifier of 12345 will be represented by http://server/employees/12345. In the case of ASP.NET Web API, the URI can be slightly different and it includes api by default in the URI, so it will be more like http://server/api/employees/12345. If you fire up an instance of Internet Explorer, type that URI in the address bar, and press Enter, Internet Explorer does an HTTP GET and you will get the JSON representation of the resource, which is an employee with the ID of 12345 in this case.

From the .NET code point of view (see Listing 2-1), the corresponding class will be EmployeesController, which is a subclass of ApiController and the method that executes to create the resource representation to be sent back to the client in its Get(int) method.

**Listing 2-1. Identification of Resources**

```csharp
public class EmployeesController : ApiController
{
    public Employee Get(int id)
    {
        // return employee
    }

    public IEnumerable<Employee> GetAllEmployees()
    {
        // return all employees
    }
}
```

In Listing 2-1, the resource that is a noun has the URI representation of http://server/api/employees/12345. This resource was accessed through GET HTTP method, which is the verb. Like one single employee, a list of employees is also a resource and its identifier will be http://server/api/employees. The corresponding method is GetAllEmployees(), which returns IEnumerable<Employee>.