7 Go web services

This chapter covers

- Using RESTful web services
- Creating and parsing XML with Go
- Creating and parsing JSON with Go
- Writing Go web services

Web services, as you'll recall from our brief discussion in chapter 1, provide a service to other software programs. This chapter expands on this and shows how you can use Go to write or consume web services. You'll learn how to create and parse XML and JSON first, because these are the most frequently used data formats with web services. We'll also discuss SOAP and RESTful services before going through the steps for creating a simple web service in JSON.

7.1 Introducing web services

One of the more popular uses of Go is in writing web services that provide services and data to other web services or applications. Web services, at a basic level, are software programs that interact with other software programs. In other words, instead of having a human being as the end user, a web service has a software

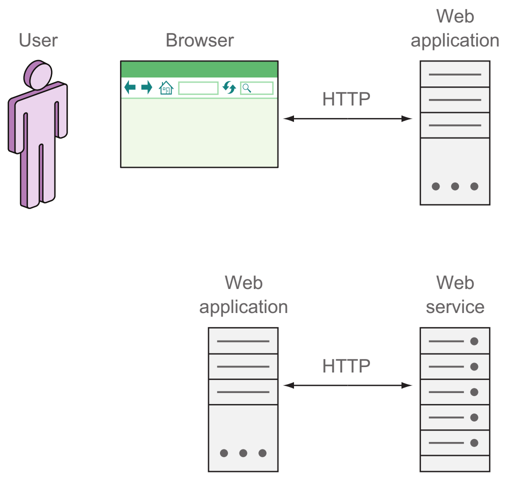

Figure 7.1 Comparing a web application with a web service

program as the end user. Web services, as the name suggests, communicate over HTTP (see figure 7.1).

Interestingly, though web applications are generally not solidly defined, you can find a definition of web services in a Web Services Architecture document by a W3C working group:

> *A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.*
>
> Web Service Architecture, February 11, 2004

From this description it appears as if all web services are SOAP-based. In reality, there are different types of web services, including SOAP-based, REST-based, and XML-RPC-based. The two most popular types are REST-based and SOAP-based. SOAP-based web services are mostly being used in enterprise systems; REST-based web services are more popular in publicly available web services. (We'll discuss them later in this chapter.)

SOAP-based and REST-based web services perform the same function, but each has its own strengths. SOAP-based web services have been around for a while, have been standardized by a W3C working group, and are very well documented. They're well supported by the enterprise and have a large number of extensions (collectively known as the WS-* because they mostly start with WS; for example, WS-Security and WS-Addressing). SOAP-based services are robust, are explicitly described using WSDL (Web Service Definition Language), and have built-in error handling. Together with

UDDI (Universal Description, Discovery, and Integration—a directory service), SOAP-based services can also be discovered.

SOAP is known to be cumbersome and unnecessarily complex. The SOAP XML messages can grow to be verbose and difficult to troubleshoot, and you may often need other tools to manage them. SOAP-based web services also can be heavy to process because of the additional overhead. WSDL, while providing a solid contract between the client and server, can become burdensome because every change in the web service requires the WSDL and therefore the SOAP clients to be changed. This often results in version lock-in as the developers of the web service are wary of making even the smallest changes.

REST-based web services are a lot more flexible. REST isn't an architecture in itself but a design philosophy. It doesn't require XML, and very often REST-based web services use simpler data formats like JSON, resulting in speedier web services. REST-based web services are often much simpler than SOAP-based ones.

Another difference between the two is that SOAP-based web services are function-driven; REST-based web services are data-driven. SOAP-based web services tend to be RPC (Remote Procedure Call) styled; REST-based web services, as described earlier, focus on resources, and HTTP methods are the verbs working on those resources.

ProgrammableWeb is a popular site that tracks APIs that are available publicly over the internet. As of this writing, its database contains 12,987 publicly available APIs, of which 2061 (or 16%) are SOAP-based and 6967 (54%) are REST-based.[1] Unfortunately, enterprises rarely publish the number of internal web services, so that figure is difficult to confirm.

Many developers and companies end up using both SOAP- and REST-based web services at the same time but for different purposes. In these cases, SOAP is used in internal applications for enterprise integration and REST is used for external, third-party developers. The advantage of this strategy is that both the strengths of REST (speed and simplicity) and SOAP (security and robustness) can be used where they're most effective.

7.2 Introducing SOAP-based web services

SOAP is a protocol for exchanging structured data that's defined in XML. SOAP was originally an acronym for Simple Object Access Protocol, terminology that is a misnomer today, as it's no longer considered simple and it doesn't deal with objects either. In the latest version, the SOAP 1.2 specification, the protocol officially became simply SOAP. SOAP is usable across different networking protocols and is independent of programming models.

SOAP is highly structured and heavily defined, and the XML used for the transportation of the data can be complex. Every operation and input or output of the service is clearly defined in the WSDL. The WSDL is the contract between the client and the server, defining what the service provides and how it's provided.

---

[1] Refer to www.programmableweb.com/category/all/apis?data_format=21176 for SOAP-based APIs and www.programmableweb.com/category/all/apis?data_format=21190 for REST-based APIs.

In this chapter we'll focus more on REST-based web services, but you should understand how SOAP-based web services work for comparison purposes.

SOAP places its message content into an envelope, like a shipping container, and it's independent of the actual means of transporting the data from one place to another. In this book, we're only looking at SOAP web services, so we're referring to SOAP messages being moved around using HTTP.

Here's a simplified example of a SOAP request message:

```
POST /GetComment HTTP/1.1
Host: www.chitchat.com
Content-Type: application/soap+xml; charset=utf-8

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
<soap:Body xmlns:m="http://www.chitchat.com/forum">
  <m:GetCommentRequest>
    <m:CommentId>123</m:CommentId>
  </m:GetCommentRequest >
</soap:Body>
</soap:Envelope>
```

The HTTP headers should be familiar by now. Note `Content-Type` is set to `application/soap+xml`. The request body is the SOAP message. The SOAP body contains the request message. In the example, this is a request for a comment with the ID `123`.

```
<m:GetCommentRequest>
  <m:CommentId>123</m:CommentId>
</m:GetCommentRequest >
```

This example is simplified—the actual SOAP requests are often a lot more complex. Here's a simplified example of a SOAP response message:

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetCommentResponse>
    <m:Text>Hello World!</m:Text>
  </m:GetCommentResponse>
</soap:Body>
</soap:Envelope>
```

As before, the response message is within the SOAP body and is a response with the text "Hello World!"

```
<m:GetCommentResponse>
  <m:Text>Hello World!</m:Text>
</m:GetCommentResponse>
```

As you may realize by now, all the data about the message is contained in the envelope. For SOAP-based web services, this means that the information sent through HTTP is almost entirely in the SOAP envelope. Also, SOAP mostly uses the HTTP POST method, although SOAP 1.2 allows HTTP GET as well.

Here's what a simple WSDL message looks like. You might notice that WSDL messages can be detailed and the message can get long even for a simple service. That's part of the reason why SOAP-based web services aren't as popular as REST-based web services —in more complex web services, the WSDL messages can be complicated.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions  name ="ChitChat"
  targetNamespace="http://www.chitchat.com/forum.wsdl"
  xmlns:tns="http://www.chitchat.com/forum.wsdl"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
  <message name="GetCommentRequest">
    <part name="CommentId" type="xsd:string"/>
  </message>
  <message name="GetCommentResponse">
    <part name="Text" type="xsd:string"/>
  </message>
  <portType name="GetCommentPortType">
    <operation name="GetComment">
      <input message="tns:GetCommentRequest"/>
      <output message="tns:GetCommentResponse"/>
    </operation>
  </portType>
  <binding name="GetCommentBinding" type="tns:GetCommentPortType">
    <soap:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="GetComment">
      <soap:operation soapAction="getComment"/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="GetCommentService" >
    <documentation>
      Returns a comment
    </documentation>
    <port name="GetCommentPortType" binding="tns:GetCommentBinding">
      <soap:address location="http://localhost:8080/GetComment"/>
    </port>
  </service>
</definitions>
```

The WSDL message defines a service named `GetCommentService`, with a port named `GetCommentPortType` that's bound to the binding `GetCommentsBinding`. The service is defined at the location http://localhost:8080/GetComment.

```
<service name="GetCommentService" >
  <documentation>
    Returns a comment
  </documentation>
  <port name="GetCommentPortType" binding="tns:GetCommentBinding">
    <soap:address location="http://localhost:8080/GetComment"/>
  </port>
</service>
```

The rest of the message gets into the details of service. The port `GetCommentPortType` is defined with a single operation called `GetComment` that has an input message, `GetCommentRequest`, and an output message, `GetCommentResponse`.

```
<portType name="GetCommentPortType">
  <operation name="GetComment">
    <input message="tns:GetCommentRequest"/>
    <output message="tns:GetCommentResponse"/>
  </operation>
</portType>
```

This is followed by a definition of the messages themselves. The definition names the message and the parts of the message and their types.

```
<message name="GetCommentRequest">
  <part name="CommentId" type="xsd:string"/>
</message>
<message name="GetCommentResponse">
  <part name="Text" type="xsd:string"/>
</message>
```

In practice, SOAP request messages are often generated by a SOAP client that's generated from the WSDL. Similarly, SOAP response messages are often generated by a SOAP server that's also generated from the WSDL. What often happens is a language-specific client (for example, a Go SOAP client) is generated from the WSDL, and this client is used by the rest of the code to interact with the server. As a result, as long as the WSDL is well defined, the SOAP client is usually robust. The drawback is that each time we change the service, even for a small matter like changing the type of the return value, the client needs to be regenerated. This can get tedious and explains why you won't see too many SOAP web service revisions (revisions can be a nightmare if it's a large web service).

I won't discuss SOAP-based web services in further detail in the rest of this chapter, although I'll show you how Go can be used to create or parse XML.

7.3 Introducing REST-based web services

REST (Representational State Transfer) is a design philosophy used in designing programs that talk to each other by manipulating resources using a standard few actions (or verbs, as many REST people like to call them).

In most programming paradigms, you often get work done by defining functions that are subsequently triggered by a main program sequentially. In OOP, you do much the same thing, except that you create models (called *objects*) to represent things and you define functions (called *methods*) and attach them to those models, which you can subsequently call. REST is an evolution of the same line of thought where instead of exposing functions as services to be called, you expose the models, called *resources*, and only allow a few actions (called *verbs*) on them.

When used over HTTP, a URL is used to represent a resource. HTTP methods are used as verbs to manipulate them, as listed in table 7.1.

Table 7.1 HTTP methods and corresponding web services

HTTP method	What to use it for	Example
POST	Creates a resource (where one doesn't exist)	POST /users
GET	Retrieves a resource	GET /users/1
PUT	Updates a resource with the given URL	PUT /users/1
DELETE	Deletes a resource	DELETE /users/1

The aha! moment that often comes to programmers who first read about REST is when they see the mapping between the use of HTTP methods for REST with the database CRUD operations. It's important to understand that this mapping is not a 1-to-1 mapping, nor is it the only mapping. For example, you can use both POST and PUT to create a new resource and either will be correctly RESTful.

The main difference between POST and PUT is that for PUT, you need to know exactly which resource it will replace, whereas a POST will create a new resource altogether, with a new URL. In other words, to create a new resource without knowing the URL, you'll use POST but if you want to replace an existing resource, you'll use PUT.

As mentioned in chapter 1, PUT is idempotent and the state of the server doesn't change regardless of the number of times you repeat your call. If you're using PUT to create a resource or to modify an existing resource, only one resource is being created at the provided URL. But POST isn't idempotent; every time you call it, POST will create a resource, with a new URL.

The second aha! moment for programmers new to REST comes when they realize that these four HTTP methods aren't the only ones that can be used. A lesser-known method called PATCH is often used to partially update a resource.

This is an example of a REST request:

```
GET /comment/123 HTTP/1.1
```

Note that there's no body associated in the GET, unlike in the corresponding SOAP request shown here:

```
POST /GetComment HTTP/1.1
Host: www.chitchat.com
Content-Type: application/soap+xml; charset=utf-8
```

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
<soap:Body xmlns:m="http://www.chitchat.com/forum">
  <m:GetCommentRequest>
    <m:CommentId>123</m:CommentId>
  </m:GetCommentRequest >
</soap:Body>
</soap:Envelope>
```

That's because you're using the GET HTTP method as the verb to get the resource (in this case, a blog post comment). You can return the same SOAP response earlier and it can still be considered a RESTful response because REST is concerned only about the design of the API and not the message that's sent. SOAP is all about the format of the messages. It's much more common to have REST APIs return JSON or at least a much simpler XML than SOAP messages. SOAP messages are so much more onerous to construct!

Like WSDL for SOAP, REST-based web services have WADL (Web Application Description Language) that describes REST-based web services, and even generate clients to access those services. But unlike WSDL, WADL isn't widely used, nor is it standardized. Also, WADL has competition in other tools like Swagger, RAML (Restful API Modeling Language), and JSON-home.

If you're looking at REST for the first time, you might be thinking that it's all well and good if we're only talking about a simple CRUD application. What about more complex services, or where you have to model some process or action?

How do you activate a customer account? REST doesn't allow you to have arbitrary actions on the resources, and you're more or less restricted to the list of available HTTP methods, so you can't have a request that looks like this:

```
ACTIVATE /user/456 HTTP/1.1
```

There are ways of getting around this problem; here are the two most common:

1. Reify the process or convert the action to a noun and make it a resource.
2. Make the action a property of the resource.

7.3.1 Convert action to a resource

Using the same example, you can convert the activate action to a resource activation. Once you do that, you can apply your HTTP methods to this resource. For example, to activate a user you can use this:

```
POST /user/456/activation HTTP/1.1

{ "date": "2015-05-15T13:05:05Z" }
```

This code will create an activation resource that represents the activation state of the user. Doing this also gives the added advantage of giving the activation resource additional properties. In our example you've added a date to the activation resource.

7.3.2 Make the action a property of the resource

If activation is a simple state of the customer account, you can simply make the action a property of the resource, and then use the PATCH HTTP method to do a partial update to the resource. For example, you can do this:

```
PATCH /user/456 HTTP/1.1

{ "active" : "true" }
```

This code will change the `active` property of the user resource to `true`.

7.4 Parsing and creating XML with Go

Now that you're armed with background knowledge of SOAP and RESTful web services, let's look at how Go can be used to create and consume them. We'll start with XML in this section and move on to JSON in the next.

XML is a popular markup language (HTML is another example of a markup language) that's used to represent data in a structured way. It's probably the most widely used format for representing structured data as well as for sending and receiving structured data. XML is a formal recommendation from the W3C, and it's defined by W3C's XML 1.0 specification.

Regardless of whether you end up writing or consuming web services, knowing how to create and parse XML is a critical part of your arsenal. One frequent use is to consume web services from other providers or XML-based feeds like RSS. Even if you'd never write an XML web service yourself, learning how to interact with XML using Go will be useful to you. For example, you might need to get data from an RSS newsfeed and use the data as part of your data source. In this case, you'd have to know how to parse XML and extract the information you need from it.

Parsing structured data in Go is quite similar, whether it's XML or JSON or any other format. To manipulate XML or JSON, you can use the corresponding XML or JSON subpackages of the encoding library. For XML, it's in the encoding/xml library.

7.4.1 Parsing XML

Let's start with parsing XML, which is most likely what you'll start doing first. In Go, you parse the XML into structs, which you can subsequently extract the data from. This is normally how you parse XML:

1. Create structs to contain the XML data.
2. Use `xml.Unmarshal` to unmarshal the XML data into the structs, illustrated in figure 7.2.

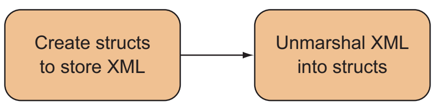

Figure 7.2 Parse XML with Go by unmarshaling XML into structs

Say you want to parse the post.xml file shown in this listing with Go.

Listing 7.1 A simple XML file, post.xml

```
<?xml version="1.0" encoding="utf-8"?>
<post id="1">
  <content>Hello World!</content>
  <author id="2">Sau Sheong</author>
</post>
```

This listing shows the code to parse the simple XML file in the code file xml.go.

Listing 7.2 Processing XML

```
package main

import (
  "encoding/xml"
  "fmt"
  "io/ioutil"
  "os"
)

type Post struct {     //#A
  XMLName xml.Name `xml:"post"`
  Id      string   `xml:"id,attr"`
  Content string   `xml:"content"`
  Author  Author   `xml:"author"`
  Xml     string   `xml:",innerxml"`
}

type Author struct {
  Id   string `xml:"id,attr"`
  Name string `xml:",chardata"`
}

func main() {
  xmlFile, err := os.Open("post.xml")
  if err != nil {
    fmt.Println("Error opening XML file:", err)
    return
  }
  defer xmlFile.Close()
  xmlData, err := ioutil.ReadAll(xmlFile)
  if err != nil {
    fmt.Println("Error reading XML data:", err)
    return
  }

  var post Post
  xml.Unmarshal(xmlData, &post)
  fmt.Println(post)
}
```

Defines structs to represent the data

Unmarshals XML data into the struct

You need to define two structs, `Post` and `Author`, to represent the data. Here you've used an `Author` struct to represent an author but you didn't use a separate `Content` struct to represent the content because for `Author` you want to capture the `id` attribute. If you didn't have to capture the `id` attribute, you could define `Post` as shown next, with a string representing an `Author` (in bold):

```
type Post struct {
  XMLName xml.Name `xml:"post"`
  Id      string   `xml:"id,attr"`
  Content string   `xml:"content"`
  Author  string   `xml:"author"`
  Xml     string   `xml:",innerxml"`
}
```

So what are those curious-looking things after the definition of each field in the Post struct? They are called *struct tags* and Go determines the mapping between the struct and the XML elements using them, shown in figure 7.3.

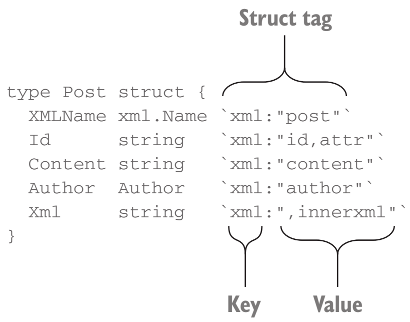

Figure 7.3 Struct tags are used to define the mapping between XML and a struct.

Struct tags are strings after each field that are a key-value pair. The key is a string and must not have a space, a quote ("), or a colon (:). The value must be a string between double quotes (""). For XML, the key must always be `xml`.

Why use backticks (`) in struct tags?

If you're wondering why backticks (`) are used to wrap around the struct tag, remember that strings in Go are created using the double quotes (") and backticks (`). Single quotes (') are used for runes (an `int32` that represents a Unicode code point) only. You're already using double quotes inside the struct tag, so if you don't want to escape those quotes, you'll have to use something else—hence the backticks.

Note that because of the way Go does the mapping, the struct and all the fields in the struct that you create must be public, which means the names need to be capitalized. In the previous code, the struct `Post` can't be just `post` and `Content` can't be `content`.

Here are some rules for the XML struct tags:

1. To store the name of the XML element itself (normally the name of the struct is the name of the element), add a field named `XMLName` with the type `xml.Name`. The name of the element will be stored in that field.
2. To store the attribute of an XML element, define a field with the same name as that attribute and use the struct tag `` `xml:"<name>,attr"` ``, where `<name>` is the name of the XML attribute.
3. To store the character data value of the XML element, define a field with the same name as the XML element tag, and use the struct tag `` `xml:",chardata"` ``.
4. To get the raw XML from the XML element, define a field (using any name) and use the struct tag `` `xml:",innerxml"` ``.
5. If there are no mode flags (like `,attr` or `,chardata` or `,innerxml`) the struct field will be matched with an XML element that has the same name as the struct's name.
6. To get to an XML element directly without specifying the tree structure to get to that element, use the struct tag `` `xml:"a>b>c"` ``, where `a` and `b` are the intermediate elements and `c` is the node that you want to get to.

Admittedly the rules can be a bit difficult to understand, especially the last couple. So let's look at some examples.

First let's look at the XML element `post` and the corresponding struct `Post`:

Listing 7.3 Simple XML element representing a `Post`

```
<post id="1">
  <content>Hello World!</content>
  <author id="2">Sau Sheong</author>
</post>
```

Compare it with this:

```
type Post struct {
  XMLName xml.Name `xml:"post"`
  Id      string   `xml:"id,attr"`
  Content string   `xml:"content"`
  Author  Author   `xml:"author"`
  Xml     string   `xml:",innerxml"`
}
```

Here you defined a struct `Post` with the same name XML element `post`. Although this is fine, if you wanted to know the name of the XML element, you'd be lost. Fortunately, the `xml` library provides a mechanism to get the XML element name by defining a struct field named `XMLName` with the type `xml.Name`. You'd also need to map this struct field to the element itself, in this case `` `xml:"post"` ``. Doing so stores the name of the element, `post`, into the field according to rule 1 in our list: to store the name of the XML element itself, you add a field named `XMLName` with the type `xml.Name`.

The `post` XML element also has an attribute named `id`, which is mapped to the struct field `Id` by the struct tag `` `xml:"id,attr"` ``. This corresponds to our second rule: to store the attribute of an XML element, you use the struct tag `` `xml:"<name>,attr"` ``.

You have the XML subelement `content`, with no attributes, but character data *Hello World!* You map this to the `Content` struct field in the `Post` struct using the struct tag `` `xml:"content"` ``. This corresponds to rule 5: if there are no mode flags the struct field will be matched with an XML element that has the same name as the struct's name.

If you want to have the raw XML within the XML element post, you can define a struct field, `Xml`, and use the struct tag `` `xml:",innerxml"` `` to map it to the raw XML within the post XML element:

```
<content>Hello World!</content>
<author id="2">Sau Sheong</author>
```

This corresponds to rule 4: to get the raw XML from the XML element, use the struct tag `` `xml:",innerxml"` ``. You also have the XML subelement `author`, which has an attribute `id`, and its subelement consists of character data *Sau Sheong*. To map this properly, you need to have another struct, `Author`:

```
type Author struct {
  Id    string `xml:"id,attr"`
  Name  string `xml:",chardata"`
}
```

Map the subelement to this struct using the struct tag `` `xml:"author"` ``, as described in rule 5. In the `Author` struct, map the attribute `id` to the struct field `Id` with `` `xml:"id,attr"` `` and the character data *Sau Sheong* to the struct field `Name` with `` `xml:",chardata"` `` using rule 3.

We've discussed the program but nothing beats running it and seeing the results. So let's give it a spin and run the following command on the console:

```
go run xml.go
```

You should see the following result:

```
{{ post} 1 Hello World! {2 Sau Sheong}
  <content>Hello World!</content>
  <author id="2">Sau Sheong</author>
}
```

Let's break down these results. The results are wrapped with a pair of braces (`{}`) because `post` is a struct. The first field in the `post` struct is another struct of type `xml.Name`, represented as `{ post }`. Next, the number `1` is the `Id`, and `"Hello World!"` is the content. After that is the `Author`, which is again another struct, `{2 Sau Sheong}`. Finally, the rest of the output is simply the inner XML.

We've covered rules 1–5. Now let's look at how rule 6 works. Rule 6 states that to get to an XML element directly without specifying the tree structure, use the struct tag `` `xml:"a>b>c"` ``, where `a` and `b` are the intermediate elements and `c` is the node that you want to get to.

The next listing is another example XML file, with the same name post.xml, showing how you can parse it.

Listing 7.4 XML file with nested elements

```
<?xml version="1.0" encoding="utf-8"?>
<post id="1">
  <content>Hello World!</content>
  <author id="2">Sau Sheong</author>
  <comments>
    <comment id="1">
      <content>Have a great day!</content>
      <author id="3">Adam</author>
    </comment>
    <comment id="2">
      <content>How are you today?</content>
      <author id="4">Betty</author>
    </comment>
  </comments>
</post>
```

Most of the XML file is similar to listing 7.3, except now you have an XML subelement, `comments` (in bold), which is a container of multiple XML subelements `comment`. In this case, you want to get the list of comments in the post, but creating a struct `Comments` to contain the list of comments seems like overkill. To simplify, you'll use rule 6 to leap-frog over the `comments` XML subelement. Rule 6 states that to get to an XML element directly without specifying the tree structure, you can use the struct tag `` `xml:"a>b>c"` ``. The next listing shows the modified `Post` struct with the new struct field and the corresponding mapping struct tag.

Listing 7.5 `Post` struct with `comments` struct field

```
type Post struct {
  XMLName  xml.Name  `xml:"post"`
  Id       string    `xml:"id,attr"`
  Content  string    `xml:"content"`
  Author   Author    `xml:"author"`
  Xml      string    `xml:",innerxml"`
  Comments []Comment `xml:"comments>comment"`
}
```

To get a list of comments, you've specified the type of the `Comments` struct field to be a slice of `Comment` structs (shown in bold). You also map this field to the `comment` XML subelement using the struct tag `` `xml:"comments>comment"` ``. According to rule 6, this will allow you to jump right into the `comment` subelement and bypass the `comments` XML element.

Here's the code for the `Comment` struct, which is similar to the `Post` struct:

```
type Comment struct {
  Id      string `xml:"id,attr"`
```

```
  Content string `xml:"content"`
  Author  Author `xml:"author"`
}
```

Now that you've defined the structs and the mapping, you can unmarshal the XML file into your structs. The input to the `Unmarshal` function is a slice of bytes (better known as a string), so you need to convert the XML file to a string first. Remember that the XML file should be in the same directory as your Go file.

```
xmlFile, err := os.Open("post.xml")
if err != nil {
  fmt.Println("Error opening XML file:", err)
  return
}
defer xmlFile.Close()
xmlData, err := ioutil.ReadAll(xmlFile)
if err != nil {
  fmt.Println("Error reading XML data:", err)
  return
}
```

Unmarshaling XML data can be a simple one-liner (two lines, if you consider defining the variable a line of its own):

```
var post Post
xml.Unmarshal(xmlData, &post)
```

If you have experience in parsing XML in other programming languages, you know that this works well for smaller XML files but that it's not efficient for processing XML that's streaming in or even in large XML files. In this case, you don't use the `Unmarshal` function and instead use the `Decoder` struct (see figure 7.4) to manually decode the XML elements. Listing 7.6 is a look at the same example, but using `Decoder`.

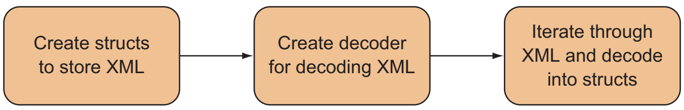

Figure 7.4 Parsing XML with Go by decoding XML into structs

Listing 7.6 Parsing XML with `Decoder`

```
package main

import (
  "encoding/xml"
  "fmt"
  "io"
  "os"
)
```

```
type Post struct {
  XMLName  xml.Name  `xml:"post"`
  Id       string    `xml:"id,attr"`
  Content  string    `xml:"content"`
  Author   Author    `xml:"author"`
  Xml      string    `xml:",innerxml"`
  Comments []Comment `xml:"comments>comment"`
}

type Author struct {
  Id   string `xml:"id,attr"`
  Name string `xml:",chardata"`
}

type Comment struct {
  Id      string `xml:"id,attr"`
  Content string `xml:"content"`
  Author  Author `xml:"author"`
}

func main() {
  xmlFile, err := os.Open("post.xml")
  if err != nil {
    fmt.Println("Error opening XML file:", err)
    return
  }
  defer xmlFile.Close()

  decoder := xml.NewDecoder(xmlFile)            // Creates decoder from XML data
  for {                                         // Iterates through XML data in decoder
    t, err := decoder.Token()                   // Gets token from decoder at each iteration
    if err == io.EOF {
      break
    }
    if err != nil {
      fmt.Println("Error decoding XML into tokens:", err)
      return
    }

    switch se := t.(type) {                     // Checks type of token
    case xml.StartElement:
      if se.Name.Local == "comment" {
        var comment Comment
        decoder.DecodeElement(&comment, &se)    // Decodes XML data into struct
      }
    }
  }
}
```

The various structs and their respective mappings remain the same. The difference is that you'll be using the `Decoder` struct to decode the XML, element by element, instead of unmarshaling the entire XML as a string.

First, you need to create a `Decoder`, which you can do by using the `NewDecoder` function and passing in an `io.Reader`. In this case use the `xmlFile` you got using `os.Open` earlier on.

Once you have the decoder, use the `Token` method to get the next token in the XML stream. A token in this context is an interface that represents an XML element. What you want to do is to continually take tokens from the decoder until you run out. So let's wrap the action of taking tokens from the decoder in an infinite `for` loop that breaks only when you run out of tokens. When that happens, `err` will not be `nil`. Instead it will contain the `io.EOF` struct, signifying that it ran out of data from the file (or data stream).

As you're taking the tokens from the decoder, you'll inspect them and check whether they're `StartElement`s. A `StartElement` represents the start tag of an XML element. If the token is a `StartElement`, check if it's a comment XML element. If it is, you can decode the entire token into a `Comment` struct and get the same results as before.

Decoding the XML file manually takes more effort and isn't worth it if it's a small XML file. But if you get XML streamed to you, or if it's a very large XML file, it's the only way of extracting data from the XML.

A final note before we discuss creating XML: the rules described in this section are only a portion of the list. For details on all the rules, refer to the `xml` library documentation, or better yet, read the source `xml` library source code.

7.4.2 Creating XML

The previous section on parsing XML was a lengthy one. Fortunately, most of what you learned there is directly applicable to this section. Creating XML is the reverse of parsing XML. Where you unmarshal XML into Go structs, you now marshal Go structs into XML. Similarly, where you decode XML into Go structs, you now encode Go structs into XML, shown in figure 7.5.

Let's start with marshaling. The code in the file xml.go, shown in listing 7.7, will generate an XML file named post.xml.

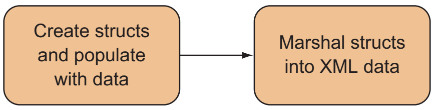

Figure 7.5 Create XML with Go by creating structs and marshaling them into XML

Listing 7.7 Using the `Marshal` function to generate an XML file

```
package main

import (
  "encoding/xml"
  "fmt"
  "io/ioutil"
)

type Post struct {
  XMLName xml.Name `xml:"post"`
  Id      string   `xml:"id,attr"`
```

```
  Content string   `xml:"content"`
  Author  Author   `xml:"author"`
}

type Author struct {
  Id   string `xml:"id,attr"`
  Name string `xml:",chardata"`
}

func main() {
  post := Post{
    Id:      "1",
    Content: " Hello World!",
    Author: Author{
      Id:   "2",
      Name: "Sau Sheong",
    },
  }
  output, err := xml.Marshal(&post)
  if err != nil {
    fmt.Println("Error marshalling to XML:", err)
    return
  }
  err = ioutil.WriteFile("post.xml", output, 0644)
  if err != nil {
    fmt.Println("Error writing XML to file:", err)
    return
  }

}
```

Creates struct with data

Marshals struct to a byte slice of XML data

As you can see, the structs and the struct tags are the same as those you used when unmarshaling the XML. Marshaling simply reverses the process and creates XML from a struct. First, you populate the struct with data. Then, using the `Marshal` function you create the XML from the `Post` struct. Here's the content of the post.xml file that's created:

```
<post id="1"><content>Hello World!</content><author id="2">Sau Sheong
     </author></post>
```

It's not the prettiest, but it's correctly formed XML. If you want to make it look prettier, use the `MarshalIndent` function:

```
output, err := xml.MarshalIndent(&post, "", "\t")
```

The first parameter you pass to `MarshalIndent` is still the same, but you have two additional parameters. The second parameter is the prefix to every line and the third parameter is the indent, and every level of indentation will be prefixed with this. Using `MarshalIndent`, you can produce prettier output:

```
<post id="1">
  <content>Hello World!</content>
  <author id="2">Sau Sheong</author>
</post>
```

Still, it doesn't look right. We don't have the XML declaration. Although Go doesn't create the XML declaration for you automatically, it does provide a constant `xml.Header` that you can use to attach to the marshaled output:

```
err = ioutil.WriteFile("post.xml", []byte(xml.Header + string(output)), 0644)
```

Prefix the output with `xml.Header` and then write it to post.xml, and you'll have the XML declaration:

```
<?xml version="1.0" encoding="UTF-8"?>
<post id="1">
    <content>Hello World!</content>
    <author id="2">Sau Sheong</author>
</post>
```

Just as you manually decoded the XML into Go structs, you can also manually encode Go structs into XML (see figure 7.6). Listing 7.8 shows a simple example.

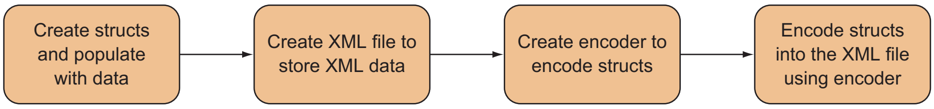

Figure 7.6 Create XML with Go by creating structs and encoding them into XML using an encoder

Listing 7.8 Manually encoding Go structs to XML

```
package main

import (
  "encoding/xml"
  "fmt"
  "os"
)

type Post struct {
  XMLName xml.Name `xml:"post"`
  Id      string   `xml:"id,attr"`
  Content string   `xml:"content"`
  Author  Author   `xml:"author"`
}

type Author struct {
  Id   string `xml:"id,attr"`
  Name string `xml:",chardata"`
}

func main() {
  post := Post{                              Creates struct with data
    Id:       "1",
    Content: "Hello World!",
```

```
    Author: Author{
      Id:   "2",
      Name: "Sau Sheong",
    },
  }

  xmlFile, err := os.Create("post.xml")           // Creates XML file to store data
  if err != nil {
    fmt.Println("Error creating XML file:", err)
    return
  }
  encoder := xml.NewEncoder(xmlFile)             // Creates encoder with XML file
  encoder.Indent("", "\t")
  err = encoder.Encode(&post)                    // Encodes struct into file
  if err != nil {
    fmt.Println("Error encoding XML to file:", err)
    return
  }
}
```

As before, you first create the `post` struct to be encoded. To write to a file, you need to create the file using `os.Create`. The `NewEncoder` function creates a new encoder that wraps around your file. After setting up the indentation you want, use the encoder's `Encode` method, passing a reference to the `post` struct. This will create the XML file post.xml:

```
<post id="1">
  <content>Hello World!</content>
  <author id="2">Sau Sheong</author>
</post>
```

You're done with parsing and creating XML, but note that this chapter discussed only the basics of parsing and creating XML. For more detailed information, see the documentation or the source code. (It's not as daunting as it sounds.)

7.5 Parsing and creating JSON with Go

JavaScript Serialized Object Notation (JSON) is a lightweight, text-based data format based on JavaScript. The main idea behind JSON is that it's easily read by both humans and machines. JSON was originally defined by Douglas Crockford, but is currently described by RFC 7159, as well as ECMA-404. JSON is popularly used in REST-based web services, although they don't necessarily need to accept or return JSON data.

If you're dealing with RESTful web services, you'll likely encounter JSON in one form or another, either creating or consuming JSON. Consuming JSON is commonplace in many web applications, from getting data from a web service, to authenticating your web application through a third-party authentication service, to controlling other services.

Creating JSON is equally common. Go is used in many cases to create web service backends for frontend applications, including JavaScript-based frontend applications running on JavaScript libraries such as React.js and Angular.js. Go is also used to create web services for Internet of Things (IoT) and wearables such as smart watches. In many of these cases, these frontend applications are developed using JSON, and the most natural way to interact with a backend application is through JSON.

As with Go's support for XML, Go's support for JSON is from the `encoding/json` library. As before we'll look into parsing JSON first, and then we'll see how to create JSON data.

7.5.1 Parsing JSON

The steps for parsing JSON data are similar to those for parsing XML. You parse the JSON into structs, from which you can subsequently extract the data. This is normally how you parse JSON:

1. Create structs to contain the JSON data.
2. Use `json.Unmarshal` to unmarshal the JSON data into the structs (see figure 7.7).

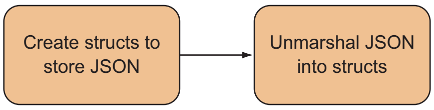

Figure 7.7 Parse JSON with Go by creating structs and unmarshaling JSON into the structs.

The rules for mapping the structs to JSON using struct tags are easier than with XML. There is only one common rule for mapping. If you want to store the JSON value, given the JSON key, you create a field in the struct (with any name) and map it with the struct tag `` `json:"<name>"` ``, where `<name>` is the name of the JSON key. Let's see in action.

The following listing shows the JSON file, post.json, that you'll be parsing. The data in this file should be familiar to you—it's the same data you used for the XML parsing.

Listing 7.9 JSON file for parsing

```
{
  "id" : 1,
  "content" : "Hello World!",
  "author" : {
    "id" : 2,
    "name" : "Sau Sheong"
  },
  "comments" : [
    {
      "id" : 3,
      "content" : "Have a great day!",
      "author" : "Adam"
    },
    {
      "id" : 4,
```

```
      "content" : "How are you today?",
      "author" : "Betty"
    }
  ]
}
```

The next listing contains the code that will parse the JSON into the respective structs, in a json.go file. Notice that the structs themselves aren't different.

Listing 7.10 JSON file for parsing

```
package main

import (
  "encoding/json"
  "fmt"
  "io/ioutil"
  "os"
)

type Post struct {
  Id       int       `json:"id"`
  Content  string    `json:"content"`
  Author   Author    `json:"author"`
  Comments []Comment `json:"comments"`
}

type Author struct {
  Id   int    `json:"id"`
  Name string `json:"name"`
}

type Comment struct {
  Id      int    `json:"id"`
  Content string `json:"content"`
  Author  string `json:"author"`
}

func main() {
  jsonFile, err := os.Open("post.json")
  if err != nil {
    fmt.Println("Error opening JSON file:", err)
    return
  }
  defer jsonFile.Close()
  jsonData, err := ioutil.ReadAll(jsonFile)
  if err != nil {
    fmt.Println("Error reading JSON data:", err)
    return
  }

  var post Post
  json.Unmarshal(jsonData, &post)
  fmt.Println(post)
}
```

Defines structs to represent the data

Unmarshals JSON data into the struct

You want to map the value of the key `id` to the Post struct's `Id` field, so we append the struct tag `` `json:"id"` `` after the field. This is pretty much what you need to do to map the structs to the JSON data. Notice that you nest the structs (a post can have zero or more comments) through slices. As before in XML parsing, unmarshaling is done with a single line of code—simply a function call.

Let's run our JSON parsing code and see the results. Run this at the console:

```
go run json.go
```

You should see the following results:

```
{1 Hello World! {2 Sau Sheong} [{3 Have a great day! Adam} {4 How are you
     today? Betty}]}
```

We looked at unmarshaling using the `Unmarshal` function. As in XML parsing, you can also use `Decoder` to manually decode JSON into the structs for streaming JSON data. This is shown in figure 7.8 and listing 7.11.

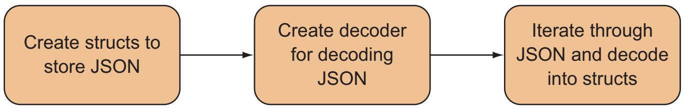

Figure 7.8 Parse XML with Go by decoding JSON into structs

Listing 7.11 Parsing JSON using `Decoder`

```
jsonFile, err := os.Open("post.json")
if err != nil {
  fmt.Println("Error opening JSON file:", err)
  return
}
defer jsonFile.Close()

decoder := json.NewDecoder(jsonFile)      // Creates decoder from JSON data
for {                                      // Iterates until EOF is detected
  var post Post
  err := decoder.Decode(&post)            // Decodes JSON data into struct
  if err == io.EOF {
    break
  }
  if err != nil {
    fmt.Println("Error decoding JSON:", err)
    return
  }
  fmt.Println(post)
}
```

Here you use `NewDecoder`, passing in an `io.Reader` containing the JSON data, to create a new decoder. When a reference to the post struct is passed into the `Decode` method, the struct will be populated with the data and will be ready for use. Once the data runs out, the `Decode` method returns an `EOF`, which you can check and then exit the loop.

Let's run our JSON decoder and see the results. Run this at the console:

```
go run json.go
```

You should see the following results.

```
{1 Hello World! {2 Sau Sheong} [{1 Have a great day! Adam} {2 How are you
➥ today? Betty}]}
```

So when do we use `Decoder` versus `Unmarshal`? That depends on the input. If your data is coming from an `io.Reader` stream, like the `Body` of an `http.Request`, use `Decoder`. If you have the data in a string or somewhere in memory, use `Unmarshal`.

7.5.2 *Creating JSON*

We just went through parsing JSON, which as you can see, is very similar to parsing XML. Creating JSON is also is similar to creating XML (see figure 7.9).

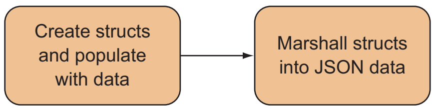

Figure 7.9 Create JSON with Go by creating structs and marshaling them into JSON data.

This listing contains the code for marshaling the Go structs to JSON.

Listing 7.12 Marshaling structs to JSON

```
package main

import (
  "encoding/json"
  "fmt"
  "io/ioutil"
)

type Post struct {
  Id       int        `json:"id"`
  Content  string     `json:"content"`
  Author   Author     `json:"author"`
  Comments []Comment  `json:"comments"`
}

type Author struct {
  Id   int    `json:"id"`
  Name string `json:"name"`
}

type Comment struct {
  Id      int    `json:"id"`
```

Creates struct with data

```
  Content string `json:"content"`
  Author  string `json:"author"`
}

func main() {
  post := Post{
    Id:      1,
    Content: "Hello World!",
    Author: Author{
      Id:   2,
      Name: "Sau Sheong",
    },
    Comments: []Comment{
      Comment{
        Id:      3,
        Content: "Have a great day!",
        Author:  "Adam",
      },
      Comment{
        Id:      4,
        Content: "How are you today?",
        Author:  "Betty",
      },
    },
  }

  output, err := json.MarshalIndent(&post, "", "\t\t")
  if err != nil {
    fmt.Println("Error marshalling to JSON:", err)
    return
  }
  err = ioutil.WriteFile("post.json", output, 0644)
  if err != nil {
    fmt.Println("Error writing JSON to file:", err)
    return
  }
}
```

Marshals struct to byte slice of JSON data

As before, the structs are the same as when you're parsing JSON. First, you create the struct. Then you call the `MarshalIndent` function (which works the same way as the one in the `xml` library) to create the JSON data in a slice of bytes. You can then save the output to file if you want to.

Finally, as in creating XML, you can create JSON manually from the Go structs using an encoder, shown in figure 7.10.

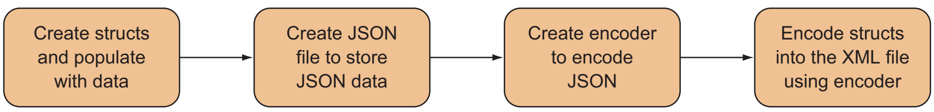

Figure 7.10 Create JSON with Go by creating structs and encoding them into JSON using an encoder.

The code in this listing, also in json.go, will generate JSON from the Go structs.

Listing 7.13 Creating JSON from structs using `Encoder`

```
package main

import (
  "encoding/json"
  "fmt"
  "io"
  "os"
)

type Post struct {                                      Creates struct
  Id       int       `json:"id"`                        with data
  Content  string    `json:"content"`
  Author   Author    `json:"author"`
  Comments []Comment `json:"comments"`
}

type Author struct {
  Id   int    `json:"id"`
  Name string `json:"name"`
}

type Comment struct {
  Id      int    `json:"id"`
  Content string `json:"content"`
  Author  string `json:"author"`
}

func main() {

  post := Post{
    Id:      1,
    Content: "Hello World!",
    Author: Author{
      Id:   2,
      Name: "Sau Sheong",
    },
    Comments: []Comment{
      Comment{
        Id:      3,
        Content: "Have a great day!",
        Author:  "Adam",
      },
      Comment{
        Id:      4,
        Content: "How are you today?",
        Author:  "Betty",
      },
    },
  }
```

```
  jsonFile, err := os.Create("post.json")
  if err != nil {
    fmt.Println("Error creating JSON file:", err)
    return
  }
  encoder := json.NewEncoder(jsonFile)
  err = encoder.Encode(&post)
  if err != nil {
    fmt.Println("Error encoding JSON to file:", err)
    return
  }
}
```

Creates JSON file to store data

Creates encoder with JSON file

Encodes struct into file

As before, you create a JSON file to store the JSON that's generated. You use this file to create an encoder using the `NewEncoder` function. Then you call the `Encode` method on the encoder and pass it a reference to the `post` struct. This will extract the data from the struct and create JSON data, which is then written to the writer you passed in earlier.

This wraps up the sections on parsing and creating XML and JSON. Going through these sections seems like plodding through similar patterns, but it provides you with the grounding you need for the next section, where you'll create a Go web service.

7.6 Creating Go web services

Creating a Go web service is relatively pain-free. If you've arrived here after going through the earlier chapters and the earlier sections in this chapter, the rest should just click and lightbulbs should start flickering on.

You're going to build a simple REST-based web service that allows you to create, retrieve, update, and retrieve forum posts. Another way of looking at it is you're wrapping a web service interface over the CRUD functions you built in chapter 6. You'll be using JSON as the data transport format. This simple web service will be reused to explain other concepts in the following chapters.

First, let's look at the database operations that you'll need. Essentially you're going to reuse—but simplify—the code from section 6.4. The code you need is placed in a file named data.go, shown in the following listing, with the package name `main`. The code isolates what you need to do with the database.

Listing 7.14 Accessing the database with data.go

```
package main

import (
  "database/sql"
  _ "github.com/lib/pq"
)

var Db *sql.DB

func init() {
```

Connects to the Db

```
  var err error
  Db, err = sql.Open("postgres", "user=gwp dbname=gwp password=gwp ssl-
   mode=disable")
  if err != nil {
    panic(err)
  }
}
```

Gets a single post

```
func retrieve(id int) (post Post, err error) {
  post = Post{}
  err = Db.QueryRow("select id, content, author from posts where id = $1",
   id).Scan(&post.Id, &post.Content, &post.Author)
  return
}
```

Creates a new post

```
func (post *Post) create() (err error) {
  statement := "insert into posts (content, author) values ($1, $2) return-
   ing id"
  stmt, err := Db.Prepare(statement)
  if err != nil {
    return
  }
  defer stmt.Close()
  err = stmt.QueryRow(post.Content, post.Author).Scan(&post.Id)
  return
}
```

Updates a post

```
func (post *Post) update() (err error) {
  _, err = Db.Exec("update posts set content = $2, author = $3 where id =
   $1", post.Id, post.Content, post.Author)
  return
}
```

Deletes a post

```
func (post *Post) delete() (err error) {
  _, err = Db.Exec("delete from posts where id = $1", post.Id)
  return
}
```

As you can see, the code is similar to that of listing 6.6, with slightly different function and method names, so we won't go through it again. If you need a refresher, please flip back to section 6.4.

Now that you can do CRUD on the database, let's turn to the actual web service. The next listing shows the entire web service in a server.go file.

Listing 7.15 Go web service in server.go

```
package main

import (
  "encoding/json"
  "net/http"
  "path"
  "strconv"
)
```

```
type Post struct {
  Id      int    `json:"id"`
  Content string `json:"content"`
  Author  string `json:"author"`
}

func main() {
  server := http.Server{
    Addr: "127.0.0.1:8080",
  }
  http.HandleFunc("/post/", handleRequest)
  server.ListenAndServe()
}

func handleRequest(w http.ResponseWriter, r *http.Request) {
  var err error
  switch r.Method {
  case "GET":
    err = handleGet(w, r)
  case "POST":
    err = handlePost(w, r)
  case "PUT":
    err = handlePut(w, r)
  case "DELETE":
    err = handleDelete(w, r)
  }
  if err != nil {
    http.Error(w, err.Error(), http.StatusInternalServerError)
    return
  }
}

func handleGet(w http.ResponseWriter, r *http.Request) (err error) {
  id, err := strconv.Atoi(path.Base(r.URL.Path))
  if err != nil {
    return
  }
  post, err := retrieve(id)
  if err != nil {
    return
  }
  output, err := json.MarshalIndent(&post, "", "\t\t")
  if err != nil {
    return
  }
  w.Header().Set("Content-Type", "application/json")
  w.Write(output)
  return
}

func handlePost(w http.ResponseWriter, r *http.Request) (err error) {
  len := r.ContentLength
  body := make([]byte, len)
  r.Body.Read(body)
  var post Post
```

Handler function to multiplex request to the correct function

Retrieves post

Creates post

```
  json.Unmarshal(body, &post)
  err = post.create()
  if err != nil {
    return
  }
  w.WriteHeader(200)
  return
}

func handlePut(w http.ResponseWriter, r *http.Request) (err error) {
  id, err := strconv.Atoi(path.Base(r.URL.Path))
  if err != nil {
    return
  }
  post, err := retrieve(id)
  if err != nil {
    return
  }
  len := r.ContentLength
  body := make([]byte, len)
  r.Body.Read(body)
  json.Unmarshal(body, &post)
  err = post.update()
  if err != nil {
    return
  }
  w.WriteHeader(200)
  return
}

func handleDelete(w http.ResponseWriter, r *http.Request) (err error) {
  id, err := strconv.Atoi(path.Base(r.URL.Path))
  if err != nil {
    return
  }
  post, err := retrieve(id)
  if err != nil {
    return
  }
  err = post.delete()
  if err != nil {
    return
  }
  w.WriteHeader(200)
  return
}
```

Updates post

Deletes post

The structure of the code is straightforward. You use a single handler function called `handleRequest` that will multiplex to different CRUD functions according to the method that was used. Each of the called functions takes in a `ResponseWriter` and a `Request` while returning an error, if any. The `handleRequest` handler function will also take care of any errors that are floated up from the request, and throw a 500 status code (`StatusInternalServerError`) with the error description, if there's an error.

Let's delve into the details and start by creating a post, shown in this listing.

Listing 7.16 Function that creates a post

```
func handlePost(w http.ResponseWriter, r *http.Request) (err error) {
  len := r.ContentLength
  body := make([]byte, len)          // Creates slice of bytes
  r.Body.Read(body)                  // Reads request body into slice
  var post Post
  json.Unmarshal(body, &post)        // Unmarshals slice into Post struct
  err = post.create()                // Creates database record
  if err != nil {
    return
  }
  w.WriteHeader(200)
  return
}
```

First, you create a slice of bytes with the correct content length size, and read the contents of the body (which is a JSON string) into it. Next, you declare a `Post` struct and unmarshal the content into it. Now that you have a `Post` struct with the fields populated, you call the `create` method on it to save it to the database.

To call the web service, you'll be using cURL (see chapter 3). Run this command on the console:

```
curl -i -X POST -H "Content-Type: application/json"  -d '{"content":"My
[CA} first post","author":"Sau Sheong"}' http://127.0.0.1:8080/post/
```

You're using the `POST` method and setting the `Content-Type` header to `application/json`. A JSON string request body is sent to the URL http://127.0.0.1/post/. You should see something like this:

```
HTTP/1.1 200 OK
Date: Sun, 12 Apr 2015 13:32:14 GMT
Content-Length: 0
Content-Type: text/plain; charset=utf-8
```

This doesn't tell us anything except that the handler function didn't encounter any errors. Let's peek into the database by running this single line SQL query from the console:

```
psql -U gwp -d gwp -c "select * from posts;"
```

You should see this:

```
 id |    content     |   author
----+---------------+------------
  1 | My first post | Sau Sheong
(1 row)
```

In each of the handler functions (except for the create handler function, `postPost`), you assume the URL will contain the `id` to the targeted post. For example, when you want to retrieve a post, you assume the web service will be called by a request to a URL:

```
/post/<id>
```

where `<id>` is the `id` of the post. The next listing shows how this works in retrieving the post.

Listing 7.17 Function that retrieves a post

```
func handleGet(w http.ResponseWriter, r *http.Request) (err error) {
  id, err := strconv.Atoi(path.Base(r.URL.Path))
  if err != nil {
    return
  }
  post, err := retrieve(id)                    // Gets data from database into Post struct
  if err != nil {
    return
  }
  output, err := json.MarshalIndent(&post, "", "\t\t")    // Marshals the Post struct into JSON string
  if err != nil {
    return
  }
  w.Header().Set("Content-Type", "application/json")      // Writes JSON to ResponseWriter
  w.Write(output)
  return
}
```

You extract the URL's path, and then get the `id` using the `path.Base` function. The `id` is a string, but you need an integer to retrieve the post, so you convert it into an integer using `strconv.Atoi`. Once you have the `id`, you can use the `retrievePost` function, which gives you a `Post` struct that's filled with data.

Next, you convert the `Post` struct into a JSON-formatted slice of bytes using the `json.MarshalIndent` function. Then you set the `Content-Type` header to `application/json` and write the bytes to the `ResponseWriter` to be returned to the calling program.

To see how this works, run this command on the console:

```
curl -i -X GET http://127.0.0.1:8080/post/1
```

This tells you to use the GET method on the URL, with the `id` 1. The results would be something like this:

```
HTTP/1.1 200 OK
Content-Type: application/json
Date: Sun, 12 Apr 2015 13:32:18 GMT
Content-Length: 69

{
    "id": 1,
    "content": "My first post",
    "author": "Sau Sheong"
}
```

You need the results when updating the post too, shown in this listing.

Listing 7.18 Function that updates a post

```
func handlePut(w http.ResponseWriter, r *http.Request) (err error) {
id, err := strconv.Atoi(path.Base(r.URL.Path))
  if err != nil {
    return
  }
  post, err := retrieve(id)          Gets data from the database into Post struct
  if err != nil {
    return
  }
  len := r.ContentLength
  body := make([]byte, len)
  r.Body.Read(body)                  Reads JSON data from request body
  json.Unmarshal(body, &post)        Unmarshals JSON data into Post struct
  err = post.update()                Updates the database
  if err != nil {
    return
  }
  w.WriteHeader(200)
  return
}
```

Updating the post involves retrieving the post and then updating its information with the information sent through the PUT request. Once you've retrieved the post, you read the body of the request, and then unmarshal the contents into the retrieved post and call the `update` method on it.

To see this in action, run this command through the console:

```
curl -i -X PUT -H "Content-Type: application/json"  -d '{"content":"Updated
➥ post","author":"Sau Sheong"}' http://127.0.0.1:8080/post/1
```

Note that unlike when you're creating the post using POST, you need to send in the `id` of the post you want to update through the URL. You should see something like this:

```
HTTP/1.1 200 OK
Date: Sun, 12 Apr 2015 14:29:39 GMT
Content-Length: 0
Content-Type: text/plain; charset=utf-8
```

Now check the database and see what you have. Run this single line SQL query from the console again:

```
psql -U gwp -d gwp -c "select * from posts;"
```

You should see this:

```
 id |    content     |   author
----+----------------+------------
  1 | Updated post   | Sau Sheong
(1 row)
```

Deleting the post through the web service, shown in the following listing, involves simply retrieving the post and calling the `delete` method on it.

Listing 7.19 Function that deletes a post

```
func handleDelete(w http.ResponseWriter, r *http.Request) (err error) {
  id, err := strconv.Atoi(path.Base(r.URL.Path))
  if err != nil {
    return
  }
  post, err := retrieve(id)          // Gets data from database into Post struct
  if err != nil {
    return
  }
  err = post.delete()                // Deletes post data in database
  if err != nil {
    return
  }
  w.WriteHeader(200)
  return
}
```

Notice that in both updating and deleting the post, you write the 200 status code to indicate all is well. If there was an error along the way, it would've been returned to the calling function (the handler function `handlePost`) and a 500 status code would've been sent back.

Let's make a final call to cURL to delete the post record:

```
curl -i -X DELETE http://127.0.0.1:8080/post/1
```

You should see something like this:

```
HTTP/1.1 200 OK
Date: Sun, 12 Apr 2015 14:38:59 GMT
Content-Length: 0
Content-Type: text/plain; charset=utf-8
```

Don't forget to run the single line SQL query again, and this time you should see nothing in the table:

```
id | content | author
----+---------+--------
(0 rows)
```

7.7 Summary

- A major use of Go today is to write web services, so being able to at least understand how to build web services is a valuable skill.
- There are mainly two types of web services: SOAP-based and REST-based web services:
  - SOAP is a protocol for exchanging structured data that's defined in XML. Because their WSDL messages can become quite complicated, SOAP-based web services aren't as popular as REST-based web services.

- REST-based web services expose resources over HTTP and allow specific actions on them.

- Creating and parsing XML and JSON are similar and involve creating a struct and either generating (unmarshaling) XML or JSON from it, or creating a struct and extracting (marshaling) XML or JSON into it.