6 Storing data

This chapter covers

- In-memory storage with structs
- File storage with CSV and gob binary files
- Relational database storage with SQL
- Go and SQL mappers

We introduced data persistence in chapter 2, briefly touching on how to persist data into a relational database, PostgreSQL. In this chapter we'll delve deeper into data persistence and talk about how you can store data in memory, files, relational databases, and NoSQL databases.

Data persistence is technically not part of web application programming, but it's often considered the third pillar of any web application—the other two pillars are templates and handlers. This is because most web applications need to store data in one form or another.

I'm generalizing but here are the places where you can store data:

- In memory (while the program is running)
- In files on the filesystem
- In a database, fronted by a server program

In this chapter, we'll go through how Go can be used to access data (create, retrieve, update, and delete—better known as CRUD) in all these places.

6.1 In-memory storage

In-memory storage refers not to storing data in in-memory databases but in the running application itself, to be used while the application is running. In-memory data is usually stored in data structures, and for Go, this primarily means with arrays, slices, maps, and most importantly, structs.

Storing data itself is no issue—it simply involves creating the structs, slices, and maps. If we stop to think about it, what we'll eventually manipulate is likely not to be the individual structs themselves, but *containers* for the structs. This could be arrays, slices, and maps but could also be any other types of data structures like stacks, trees, and queues.

What's more interesting is how you can retrieve the data that you need back from these containers. In the following listing, you'll use maps as containers for your structs.

Listing 6.1 Storing data in memory

```
package main

import (
    "fmt"
)

type Post struct {
    Id     int
    Content string
    Author  string
}

var PostById map[int]*Post
var PostsByAuthor map[string][]*Post

func store(post Post) {
    PostById[post.Id] = &post
    PostsByAuthor[post.Author] = append(PostsByAuthor[post.Author], &post)
}

func main() {

    PostById = make(map[int]*Post)
    PostsByAuthor = make(map[string][]*Post)

    post1 := Post{Id: 1, Content: "Hello World!", Author: "Sau Sheong"}
    post2 := Post{Id: 2, Content: "Bonjour Monde!", Author: "Pierre"}
    post3 := Post{Id: 3, Content: "Hola Mundo!", Author: "Pedro"}
post4 := Post{Id: 4, Content: "Greetings Earthlings!", Author:
➥ "Sau Sheong"}
```

```
    store(post1)
    store(post2)
    store(post3)
    store(post4)

    fmt.Println(PostById[1])
    fmt.Println(PostById[2])

    for _, post := range PostsByAuthor["Sau Sheong"] {
        fmt.Println(post)
    }
    for _, post := range PostsByAuthor["Pedro"] {
        fmt.Println(post)
    }
}
```

You're going to use a `Post` struct that represents a post in a forum application. Here's the data that you'll be saving in memory:

```
type Post struct {
    Id     int
    Content string
    Author  string
}
```

The main data for this `Post` struct is the content, and there are two ways of getting the post: either by a unique ID or by the name of its author. Storing posts in a map means that you're going to map a key that represents the post with the actual `Post` struct. Because you have two ways of accessing a post, you should have two maps, one each to access the post:

```
var PostById map[int]*Post
var PostsByAuthor map[string][]*Post
```

You have two variables: `PostById` maps the unique ID to a pointer to a post; `PostsByAuthor` maps the author's name to a slice of pointers to posts. Notice that you map to pointers of the posts and not to the post themselves. The reason for this is obvious: whether you're getting the post through its ID or through the author's name, you want the same post, not two different copies of it.

To store the post, you create a `store` function:

```
func store(post Post) {
    PostById[post.Id] = &post
    PostsByAuthor[post.Author] = append(PostsByAuthor[post.Author], &post)
}
```

The `store` function stores a pointer to the post into `PostById` as well as `PostsByAuthor`. Next, you create the posts themselves, a process that involves nothing more than creating structs.

```
post1 := Post{Id: 1, Content: "Hello World!", Author: "Sau Sheong"}
post2 := Post{Id: 2, Content: "Bonjour Monde!", Author: "Pierre"}
```

```
post3 := Post{Id: 3, Content: "Hola Mundo!", Author: "Pedro"}
post4 := Post{Id: 4, Content: "Greetings Earthlings!", Author: "Sau Sheong"}
store(post1)
store(post2)
store(post3)
store(post4)
```

When you execute the program, you'll see the following:

```
&{1 Hello World! Sau Sheong}
&{2 Bonjour Monde! Pierre}
&{1 Hello World! Sau Sheong}
&{4 Greetings Earthlings! Sau Sheong}
&{3 Hola Mundo! Pedro}
```

Note that you're getting back the same post regardless of whether you access it through the author or the post ID.

This process seems simple and obvious enough—trivial even. Why would we want to even talk about storing data in memory?

Very often in our web applications we start off with using relational databases (as in chapter 2) and then as we scale, we realize that we need to cache the data that we retrieve from the database in order to improve performance. As you'll see in the rest of this chapter, most of the methods used to persist data involve structs in one way or another. Instead of using an external in-memory database like Redis, we have the option of refactoring our code and storing the cache data in memory.

I'll also introduce you to storing data in structs, which is going to be the recurrent pattern for data storage for this chapter and much of the book.

6.2 File storage

Storing in memory is fast and immediate because there's no retrieval from disk. But there's one very important drawback: in-memory data isn't actually persistent. If you never shut down your machine or program, or if it doesn't matter if the data is lost (as in a cache), then that's probably fine. But you usually want data to be persisted when the machine or program is shut down, even if it's in cache. There are a number of ways data can be persisted, but the most common method is to store it to some sort of nonvolatile storage such as a hard disk or flash memory.

You have a number of options for storing data to nonvolatile storage. The technique we'll discuss in this section revolves around storing data to the filesystem. Specifically we'll explore two ways of storing data to files in Go. The first is through a commonly used text format, CSV (comma-separated values), and the second is specific to Go—using the `gob` package.

CSV is a common file format that's used for transferring data from the user to the system. It can be quite useful when you need to ask your users to provide you with a large amount of data and it's not feasible to ask them to enter the data into your forms. You can ask your users to use their favorite spreadsheet, enter all their data, and then save it as CSV and upload it to your web application. Once you have the file,

you can decode the data for your purposes. Similarly, you can allow your users to get their data by creating a CSV file out of their data and sending it to them from your web application.

Gob is a binary format that can be saved in a file, providing a quick and effective means of serializing in-memory data to one or more files. Binary data files can be pretty useful too. You can use them to quickly store your structs for backup or for orderly shutdown. Just as a caching mechanism is useful, being able to store and load data temporarily in files is useful for things like sessions or shopping carts, or to serve as a temporary workspace.

Let's start with the simple exercise of opening up a file and writing to it, shown in the following listing. You'll see this repeated as we discuss saving to CSV and gob binary files.

Listing 6.2 Reading and writing to a file

```
package main

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

func main() {
    data := []byte("Hello World!\n")
    err := ioutil.WriteFile("data1", data, 0644)
    if err != nil {
        panic(err)
    }
    read1, _ := ioutil.ReadFile("data1")
    fmt.Print(string(read1))

    file1, _ := os.Create("data2")
    defer file1.Close()

    bytes, _ := file1.Write(data)
    fmt.Printf("Wrote %d bytes to file\n", bytes)

    file2, _ := os.Open("data2")
    defer file2.Close()

    read2 := make([]byte, len(data))
    bytes, _ = file2.Read(read2)
    fmt.Printf("Read %d bytes from file\n", bytes)
    fmt.Println(string(read2))
}
```

Writes to file and reads from file using WriteFile and ReadFile

Writes to file and reads from file using the File struct

To reduce the amount of code on the page, in the previous listing I've replaced the errors returned by the function with a blank identifier.

In the listing, you can see two ways of writing to and reading from a file. The first is short and simple, and uses `WriteFile` and `ReadFile` from the `ioutil` package.

Writing to a file uses `WriteFile`, passing in the name of the file, the data to be written, and a number representing the permissions to set for the file. Reading from a file simply uses `ReadFile` with the filename. The data that's passed to both `WriteFile` and read from `ReadFile` is byte slices.

Writing to and reading from a file using a `File` struct is more verbose but gives you more flexibility. To write a file, you first create it using the `Create` function in the `os` package, passing it the name of the file you want to create. It's good practice to use `defer` to close the file so that you won't forget. A `defer` statement pushes a function call on a stack. The list of saved calls is then executed after the surrounding function returns. In our example, this means at the end of the main function `file2` will be closed, followed by `file1`. Once you have the `File` struct, you can write to it using the `Write` method. There are a number of other methods you can call on the `File` struct to write data to a file.

Reading a file with the `File` struct is similar. You need to use the `Open` function in the `os` package, and then use the `Read` method on the `File` struct, or any of the other methods to read the data. Reading data using the `File` struct is much more flexible because `File` has several other methods you can use to locate the correct part of the file you want to read from.

When you execute the program, you should see two files being created: data1 and data2, both containing the text "Hello World!".

6.2.1 Reading and writing CSV files

The CSV format is a file format in which tabular data (numbers and text) can be easily written and read in a text editor. CSV is widely supported, and most spreadsheet programs, such as Microsoft Excel and Apple Numbers, support CSV. Consequently, many programming languages, including Go, come with libraries that produce and consume the data in CSV files.

In Go, CSV is manipulated by the `encoding/csv` package. The next listing shows code for reading and writing CSV.

Listing 6.3 Reading and writing CSV

```
package main

import (
    "encoding/csv"
    "fmt"
    "os"
    "strconv"
)

type Post struct {
    Id     int
    Content string
    Author  string
}
```

```
func main() {
    csvFile, err := os.Create("posts.csv")
    if err != nil {
        panic(err)
    }
    defer csvFile.Close()

    allPosts := []Post{
        Post{Id: 1, Content: "Hello World!", Author: "Sau Sheong"},
        Post{Id: 2, Content: "Bonjour Monde!", Author: "Pierre"},
        Post{Id: 3, Content: "Hola Mundo!", Author: "Pedro"},
        Post{Id: 4, Content: "Greetings Earthlings!", Author: "Sau Sheong"},
    }

    writer := csv.NewWriter(csvFile)
    for _, post := range allPosts {
        line := []string{strconv.Itoa(post.Id), post.Content, post.Author}
        err := writer.Write(line)
        if err != nil {
            panic(err)
        }
    }
    writer.Flush()

    file, err := os.Open("posts.csv")
    if err != nil {
        panic(err)
    }
    defer file.Close()

    reader := csv.NewReader(file)
    reader.FieldsPerRecord = -1
    record, err := reader.ReadAll()
    if err != nil {
        panic(err)
    }

    var posts []Post
    for _, item := range record {
        id, _ := strconv.ParseInt(item[0], 0, 0)
        post := Post{Id: int(id), Content: item[1], Author: item[2]}
        posts = append(posts, post)
    }
    fmt.Println(posts[0].Id)
    fmt.Println(posts[0].Content)
    fmt.Println(posts[0].Author)
}
```

Creating a CSV file

Reading a CSV file

First let's look at writing to a CSV file. You create a file called posts.csv and a variable named `csvFile`. Your objective is to write the posts in the `allPosts` variable into this file. Step one is to create a writer using the `NewWriter` function, passing in the file. Then for each post, you create a slice of strings. Finally, you call the `Write` method on the writer to write the slice of strings into the CSV file and you're done.

If the program ends here and exits, all is well and the data is written to file. Because you'll need to read the same posts.csv file next, we need to make sure that any buffered data is properly written to the file by calling the `Flush` method on the writer.

Reading the CSV file works much the same way. First, you need to open the file. Then call the `NewReader` function, passing in the file, to create a reader. Set the `FieldsPerRecord` field in the reader to be a negative number, which indicates that you aren't that bothered if you don't have all the fields in the record. If `FieldsPerRecord` is a positive number, then that's the number of fields you expect from each record and Go will throw an error if you get less from the CSV file. If `FieldsPerRecord` is 0, you'll use the number of fields in the first record as the `FieldsPerRecord` value.

You call the `ReadAll` method on the reader to read all the records in at once, but if the file is large you can also retrieve one record at a time from the reader. This results in a slice of slices, which you can then iterate through and create the `Post` structs. If you run the program now, it'll create a CSV file called posts.csv, which contains lines of comma-delimited text:

```
1,Hello World!,Sau Sheong
2,Bonjour Monde!,Pierre
3,Hola Mundo!,Pedro
4,Greetings Earthlings!,Sau Sheong
```

It'll also read from the same file and print out the data from the first line of the CSV file:

```
1
Hello World!
Sau Sheong
```

6.2.2 The gob package

The `encoding/gob` package manages streams of gobs, which are binary data, exchanged between an encoder and a decoder. It's designed for serialization and transporting data but it can also be used for persisting data. Encoders and decoders wrap around writers and readers, which conveniently allows you to use them to write to and read from files. The following listing demonstrates how you can use the `gob` package to create binary data files and read from them.

Listing 6.4 Reading and writing binary data using the `gob` package

```
package main

import (
    "bytes"
    "encoding/gob"
    "fmt"
    "io/ioutil"
)

type Post struct {
    Id      int
    Content string
```

```
    Author  string
}

func store(data interface{}, filename string) {
    buffer := new(bytes.Buffer)
    encoder := gob.NewEncoder(buffer)
    err := encoder.Encode(data)
    if err != nil {
        panic(err)
    }
    err = ioutil.WriteFile(filename, buffer.Bytes(), 0600)
    if err != nil {
        panic(err)
    }
}

func load(data interface{}, filename string) {
    raw, err := ioutil.ReadFile(filename)
    if err != nil {
        panic(err)
    }
    buffer := bytes.NewBuffer(raw)
    dec := gob.NewDecoder(buffer)
    err = dec.Decode(data)
    if err != nil {
        panic(err)
    }
}

func main() {
    post := Post{Id: 1, Content: "Hello World!", Author: "Sau Sheong"}
    store(post, "post1")
    var postRead Post
    load(&postRead, "post1")
    fmt.Println(postRead)
}
```

Store data

Load data

As before, you're using the `Post` struct and you'll be saving a post to binary, then retrieving it, using the `store` and `load` functions, respectively. Let's look at the `store` function first.

The `store` function takes an empty interface (meaning it can take anything, as well as a filename for the binary file it'll be saved to). In our example code, you'll be passing a `Post` struct. First, you need to create a `bytes.Buffer` struct, which is essentially a variable buffer of bytes that has both `Read` and `Write` methods. In other words, a `bytes.Buffer` can be both a `Reader` and a `Writer`.

Then you create a gob encoder by passing the buffer to the `NewEncoder` function. You use the encoder to encode the data (the `Post` struct) into the buffer using the `Encode` method. Finally, you write the buffer to a file.

To use the `store` function, you pass a `Post` struct and a filename to it, creating a binary data file named post1. Now let's look at the `load` function. Loading data from the binary data file post1 is the reverse of creating it. First, you need to read the raw data out from the file.

Next, you'll create a buffer from the raw data. Doing so will essentially give the raw data the `Read` and `Write` methods. You create a decoder from the buffer using the `NewDecoder` function. The decoder is then used to decode the raw data into the `Post` struct that you passed in earlier.

You define a `Post` struct called `postRead`, and then pass a reference to it into the `load` function, along with the name of the binary data file. The `load` function will load the data from the binary file into the struct.

When you run the program, a post1 file, which contains the binary data, will be created. You can open it and it'll look like gibberish. The post1 file is also read into another `Post` struct, and you'll see the struct being printed on the console:

```
{1 Hello World! Sau Sheong}
```

We're done with files. For the rest of this chapter, we'll be discussing data stored in specialized server-side programs called database servers.

6.3 Go and SQL

Storing and accessing data in the memory and on the filesystem is useful, but if you need robustness and scalability, you'll need to turn to *database servers*. Database servers are programs that allow other programs to access data through a client-server model. The data is normally protected from other means of access, except through the server. Typically, a client (either a library or another program) connects to the database server to access the data through a Structured Query Language (SQL). Database management systems (DBMSs) often include a database server as part of the system.

Perhaps the most well-known and popularly used database management system is the *relational database management system* (RDBMS). RDBMSs use *relational databases*, which are databases that are based on the relational model of data. Relational databases are mostly accessed through relational database servers using SQL.

Relational databases and SQL are also the most commonly used means of storing data in a scalable and easy-to-use way. I discussed this topic briefly in chapter 2, and I promised that we'll go through it properly in this chapter, so here goes.

6.3.1 Setting up the database

Before you start, you need to set up your database. In chapter 2 you learned how to install and set up Postgres, which is the database we're using for this section. If you haven't done so, now is a great time to do it.

Once you've created the database, you'll follow these steps:

1. Create the database user.
2. Create the database for the user.
3. Run the setup script that'll create the table that you need.

Let's start with creating the user. Run this command at the console:

```
createuser -P -d gwp
```

This command creates a Postgres database user called *gwp*. The option `-P` tells the `createuser` program to prompt you for a password for gwp, and the option `-d` tells the program to allow gwp to create databases. You'll be prompted to enter gwp's password, which I assume you'll set to gwp as well.

Next, you'll create the database for the gwp user. The database name has to be the same as the user's name. You can create databases with other names but that will require setting up permissions and so on. For simplicity's sake let's use the default database for our database user. To create a database named gwp, run this command at the console:

```
createdb gwp
```

Now that you have a database, let's create our one and only table. Create a file named setup.sql with the script shown in this listing.

Listing 6.5 Script that creates our database

```
create table posts (
  id     serial primary key,
  content text,
  author  varchar(255)
);
```

To execute the script, run this command on the console

```
psql -U gwp -f setup.sql -d gwp
```

and you should now have your database. Take note that you'll likely need to run this command over and over again to clean and set up the database every time before running the code.

Now that you have your database and it's set up properly, let's connect to it. The next listing shows the example we'll be going through, using a file named store.go.

Listing 6.6 Go and CRUD with Postgres

```
package main

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

type Post struct {
    Id     int
    Content string
    Author  string
}
```

```
var Db *sql.DB
func init() {
    var err error
    Db, err = sql.Open("postgres", "user=gwp dbname=gwp password=gwp
    ➥ sslmode=disable")
    if err != nil {
        panic(err)
    }
}

func Posts(limit int) (posts []Post, err error) {
    rows, err := Db.Query("select id, content, author from posts limit $1",
    ➥ limit)
    if err != nil {
        return
    }
    for rows.Next() {
        post := Post{}
        err = rows.Scan(&post.Id, &post.Content, &post.Author)
        if err != nil {
            return
        }
        posts = append(posts, post)
    }
    rows.Close()
    return
}

func GetPost(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
}

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

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
}

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

Connects to the database

Gets a single post

Creates a new post

Updates a post

Deletes a post

```
    return
}

func main() {
    post := Post{Content: "Hello World!", Author: "Sau Sheong"}

    fmt.Println(post)                          // {0 Hello World! Sau Sheong}
    post.Create()
    fmt.Println(post)                          // {1 Hello World! Sau Sheong}

    readPost, _ := GetPost(post.Id)
    fmt.Println(readPost)                      // {1 Hello World! Sau Sheong}

    readPost.Content = "Bonjour Monde!"
    readPost.Author = "Pierre"
    readPost.Update()

    posts, _ := Posts()
    fmt.Println(posts)                         // [{1 Bonjour Monde! Pierre}]

    readPost.Delete()
}
```

6.3.2 Connecting to the database

You need to connect to the database before doing anything else. Doing so is relatively simple; in the following listing you first declare a variable `Db` as a pointer to an `sql.DB` struct, and then use the `init` function (which is called automatically for every package) to initialize it.

Listing 6.7 Function that creates a database handle

```
var Db *sql.DB

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

The `sql.DB` struct is a handle to the database and represents a pool of zero or database connections that's maintained by the `sql` package. Setting up the connection to the database is a one-liner using the `Open` function, passing in the database driver name (in our case, it's *postgres*) and a data source name. The data source name is a string that's specific to the database driver and tells the driver how to connect to the database. The `Open` function then returns a pointer to a `sql.DB` struct.

Note that the `Open` function doesn't connect to the database or even validate the parameters yet—it simply sets up the necessary structs for connection to the database later. The connection will be set up lazily when it's needed.

Also, `sql.DB` doesn't needed to be closed (you can do so if you like); it's simply a handle and not the actual connection. Remember that this abstraction contains a pool of database connections and will maintain them. In our example, you'll be using the globally defined `Db` variable from our various CRUD methods and functions, but an alternative is to pass the `sql.DB` struct around once you've created it.

So far we've discussed the `Open` function, which returns a `sql.DB` struct given the database driver name and a data source name. How do you get the database driver? The normal way that you'd register a database driver involves using the `Register` function, with the name of the database driver, and a struct that implements the `driver.Driver` interface like this:

```
sql.Register("postgres", &drv{})
```

In this example, `postgres` is the name of the driver and `drv` is a struct that implements the `Driver` interface. You'll notice that we didn't do this earlier. Why not?

The Postgres driver we used (a third-party driver) registered itself when we imported the driver, that's why.

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

The `github.com/lib/pq` package you imported is the Postgres driver, and when it's imported, its `init` function will kick in and register itself. Go doesn't provide any official database drivers; all database drivers are third-party libraries that should conform to the interfaces defined in the `sql.driver` package. Notice that when you import the database driver, you set the name of the package to be an underscore (_). This is because you shouldn't use the database driver directly; you should use `database/sql` only. This way, if you upgrade the version of the driver or change the implementation of the driver, you don't need to make changes to all your code.

To install this driver, run this command on the console:

```
go get "github.com/lib/pq"
```

This command will fetch the code from the repository and place it in the package repository, and then compile it along with your other code.

6.3.3 Creating a post

With the initial database setup done, let's start creating our first record in the database. In this example, you'll the same `Post` struct you've used in the previous few sections. Instead of storing to memory or file, you'll be storing and retrieving the same information from a Postgres database.

In our sample application, you'll use various functions to perform create, retrieve, update, and delete the data. In this section, you'll learn how to create posts using the

`Create` function. Before we get into the `Create` function, though, we'll discuss how you want to create posts.

You'll begin by creating a `Post` struct, with the Content and Author fields filled in. Note that you're not filling in the Id field because it'll be populated by the database (as an auto-incremented primary key).

```
post := Post{Content: "Hello World!", Author: "Sau Sheong"}
```

If you pause here and insert a `fmt.Println` statement to debug, you'll see that the Id field is set to 0:

```
fmt.Println(post)                    ← {0 Hello World! Sau Sheong}
```

Now, let's create this post as a database record:

```
post.Create()
```

The `Create` method should return an error if something goes wrong, but for brevity's sake, let's ignore that. Let's print out the value in the variable again:

```
fmt.Println(post)                    ← {1 Hello World! Sau Sheong}
```

This time you'll see that the Id field should be set to 1. Now that you know what you want the `Create` function to do, let's dive into the code.

Listing 6.8 Creating a post

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

The `Create` function is a method to the `Post` struct. You can see that because when you're defining the `Create` function, you place a reference to a `Post` struct between the `func` keyword and the name of the function, `Create`. The reference to a `Post` struct, `post`, is also called the receiver of the method and can be referred to without the ampersand (`&`) within the method.

You start the method by getting defining an SQL prepared statement. A *prepared statement* is an SQL statement template, where you can replace certain values during execution. Prepared statements are often used to execute statements repeatedly.

```
statement := "insert into posts (content, author) values ($1, $2) returning id"
```

Replace `$1` and `$2` with the actual values you want when creating the record. Notice that you're stating that you want the database to return the id column. Why we need to return the value of the id column will become clear soon.

To create it as a prepared statement, let's use the `Prepare` method from the `sql.DB` struct:

```
stmt, err := db.Prepare(statement)
```

This code will create a reference to an `sql.Stmt` interface (defined in the `sql.Driver` package and the struct implemented by the database driver), which is our statement.

Next, execute the prepared statement using the `QueryRow` method on the statement, passing it the data from the receiver:

```
err = stmt.QueryRow(post.Content, post.Author).Scan(&post.Id)
```

You use `QueryRow` here because you want to return only a single reference to an `sql.Row` struct. If more than one `sql.Row` is returned by the SQL statement, only the first is returned by `QueryRow`. The rest are discarded. `QueryRow` returns only the `sql.Row` struct; no errors. This is because `QueryRow` is often used with the `Scan` method on the `Row` struct, which copies the values in the row into its parameters. As you can see, the `post` receiver will have its Id field filled by the returned id field from the SQL query. This is why you need to specify the returning instruction in your SQL statement. Obviously you only need the Id field, since that's the auto-incremented value generated by the database, while you already know the Content and Author fields. As you've likely guessed by now, because the post's Id field is populated, you'll now have a fully filled `Post` struct that corresponds to a database record.

6.3.4 Retrieving a post

You've created the post, so naturally you need to retrieve it. As before, you want to see what you need before creating the function to do it. You don't have an existing `Post` struct, so you can't define a method on it. You'll have to define a `GetPost` function, which takes in a single Id and returns a fully filled `Post` struct:

```
readPost, _ := GetPost(1)
fmt.Println(readPost)                    ← {1 Hello World! Sau Sheong}
```

Note that this code snippet is slightly different from the overall listing; I'm making it clearer here that I'm retrieving a post by its id. This listing shows how the `GetPost` function works.

Listing 6.9 Retrieving a post

```
func GetPost(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
}
```

You want to return a `Post` struct, so you start by creating an empty one:

```
post = Post{}
```

As you saw earlier, you can chain the `QueryRow` method and the `Scan` method to copy the value of the returned results on the empty `Post` struct. Notice that you're using the `QueryRow` method on the `sql.DB` struct instead of `sql.Stmt` because obviously you don't have or need a prepared statement. It should also be obvious that you could have done it either way in the `Create` and `GetPost` functions. The only reason I'm showing you a different way is to illustrate the possibilities.

Now that you have the empty `Post` struct populated with the data from the database, it'll be returned to the calling function.

6.3.5 Updating a post

After you retrieve a post, you may want to update the information in the record. Assuming you've already retrieved `readPost`, let's modify it and then have the new data updated in the database as well:

```
readPost.Content = "Bonjour Monde!"
readPost.Author = "Pierre"
readPost.Update()
```

You'll create an `Update` method on the `Post` struct for this purpose, as shown in this listing.

Listing 6.10 Updating 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
}
```

Unlike when creating a post, you won't use a prepared statement. Instead, you'll jump right in with a call to the `Exec` method of the `sql.DB` struct. You no longer have to update the receiver, so you don't need to scan the returned results. Therefore, using the `Exec` method on the global database variable `Db`, which returns `sql.Result` and an error, is much faster:

```
_, err = Db.Exec(post.Id, post.Content, post.Author)
```

We aren't interested in the result (which just gives the number of rows affected and the last inserted id, if applicable) because there's nothing we want to process from it, so you can ignore it by assigning it to the underscore (_). And your post will be updated (unless there's an error).

6.3.6 Deleting a post

So far we've been able to create, retrieve, and update posts. Deleting them when they're not needed is a natural extension. Assuming that you already have the

`readPost` after retrieving it previously, you want to be able to delete it using a `Delete` method:

```
readPost.Delete()
```

That's simple enough. If you look at the `Delete` method in the `Post` struct in the following listing, there's nothing new that we haven't gone through before.

Listing 6.11 Deleting a post

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

As before, when you were updating the post, you'll jump right into using the `Exec` method on the `sql.DB` struct. This executes the SQL statement, and as before, you're not interested in the returned result and so assign it to the underscore (_).

You probably noticed that the methods and functions I created are arbitrary. You can certainly change them to however you'd like. Instead of populating a `Post` struct with your changes, then calling the `Update` method on the struct, for example, you can pass the changes as parameters to the `Update` method. Or more commonly, if you want to retrieve posts using a particular column or filter, you can create different functions to do that.

6.3.7 Getting all posts

One common function is to get all posts from the database, with a given limit. In other words, you want to do the following:

```
posts, _ := Posts(10)
```

You want to get the first 10 posts from the database and put them in a slice. This listing shows how you can do this.

Listing 6.12 Getting all posts

```
func Posts(limit int) (posts []Post, err error) {
    rows, err := Db.Query("select id, content, author from posts limit $1",
    ➥ limit)
    if err != nil {
        return
    }
    for rows.Next() {
        post := Post{}
        err = rows.Scan(&post.Id, &post.Content, &post.Author)
        if err != nil {
            return
        }
        posts = append(posts, post)
    }
    rows.Close()
```

```
    return
}
```

You use the `Query` method on the `sql.DB` struct, which returns a `Rows` interface. The `Rows` interface is an iterator. You can call a `Next` method on it repeatedly and it'll return `sql.Row` until it runs out of rows, when it returns `io.EOF`.

For each iteration, you create a `Post` struct and scan the row into the struct, and then append it to the slice that you'll be returning to the caller.

6.4 Go and SQL relationships

One of the reasons relational databases are so popular for storing data is because tables can be related. This allows pieces of data to be related to each other in a consistent and easy-to-understand way. There are essentially four ways of relating a record to other records.

- *One to one* (has one)—A user has one profile, for example.
- *One to many* (has many)—A user has many forum posts, for example.
- *Many to one* (belongs to)—Many forum posts belong to one user, for example.
- *Many to many*—A user can participate in many forum threads, while a forum thread can also have many users contributing to it, for example.

We've discussed the standard CRUD for a single database table, but now let's look at how we can do the same for two related tables. For this example, we'll use a one-to-many relationship where a forum post has many comments. As you may realize, many-to-one is the inverse of one-to-many, so we'll see how that works as well.

6.4.1 Setting up the databases

Before we start, let's set up our database again, this time with two tables. You'll use the same commands on the console as before, but with a slightly different setup.sql script, shown in this listing.

Listing 6.13 Setting up two related tables

```
drop table posts cascade if exists;
drop table comments if exists;

create table posts (
  id     serial primary key,
  content text,
  author  varchar(255)
);

create table comments (
  id     serial primary key,
  content text,
  author  varchar(255),
  post_id integer references posts(id)
);
```

First, you'll see that because the tables are related, when you drop the posts table you need to cascade it; otherwise you won't be able to drop posts because the comments table depends on it. We've added the table comments, which has the same columns as posts but with an additional column, post_id, that references the column id in the posts table. This will set up post_id as a foreign key that references the primary key `id` in the posts table.

With the tables set up, let's look at the code in a single listing. The code in the following listing is found in a file named store.go.

Listing 6.14 One-to-many and many-to-one with Go

```
package main

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

type Post struct {
    Id      int
    Content string
    Author  string
  Comments []Comment
}

type Comment struct {
    Id      int
    Content string
    Author  string
    Post    *Post
}

var Db *sql.DB

func init() {
    var err error
    Db, err = sql.Open("postgres", "user=gwp dbname=gwp password=gwp
    ➥ sslmode=disable")
    if err != nil {
        panic(err)
    }
}

func (comment *Comment) Create() (err error) {
    if comment.Post == nil {
        err = errors.New("Post not found")
        return
    }
    err = Db.QueryRow("insert into comments (content, author, post_id)
    ➥ values ($1, $2, $3) returning id", comment.Content, comment.Author,
    ➥ comment.Post.Id).Scan(&comment.Id)
```

Creates a single comment

```
    return
}

func GetPost(id int) (post Post, err error) {
    post = Post{}
      post.Comments = []Comment{}
    err = Db.QueryRow("select id, content, author from posts where id =
    ➥ $1", id).Scan(&post.Id, &post.Content, &post.Author)

    rows, err := Db.Query("select id, content, author from comments")
    if err != nil {
        return
    }
    for rows.Next() {
        comment := Comment{Post: &post}
        err = rows.Scan(&comment.Id, &comment.Content, &comment.Author)
        if err != nil {
            return
        }
        post.Comments = append(post.Comments, comment)
    }
    rows.Close()
    return
}

func (post *Post) Create() (err error) {
    err = Db.QueryRow("insert into posts (content, author) values ($1, $2)
    ➥ returning id", post.Content, post.Author).Scan(&post.Id)
    return
}

func main() {
    post := Post{Content: "Hello World!", Author: "Sau Sheong"}
    post.Create()

    comment := Comment{Content: "Good post!", Author: "Joe", Post: &post}
    comment.Create()
    readPost, _ := GetPost(post.Id)

  fmt.Println(readPost)
  fmt.Println(readPost.Comments)
  fmt.Println(readPost.Comments[0].Post)
}
```

[{1 Good post! Joe 0xc20802a1c0}]

{1 Hello World! Sau Sheong [{1 Good post! Joe 0xc20802a1c0}]}

&{1 Hello World! Sau Sheong [{1 Good post! Joe 0xc20802a1c0}]}

6.4.2 One-to-many relationship

As before, let's decide how to establish the relationships. First, look at the `Post` and `Comment` structs:

```
type   Post  struct {
    Id       int
    Content  string
    Author   string
    Comments []Comment
}
```

```
type Comment struct {
    Id      int
    Content string
    Author  string
    Post    *Post
}
```

Notice that `Post` has an additional field named Comments, which is a slice of `Comment` structs. `Comment` has a field named Post that's a pointer to a `Post` struct. An astute reader might ask, why are we using a field that's a pointer in `Comment` while we have a field that's an actual struct in `Post`? We don't. The Comments field in the `Post` struct is a slice, which is really a pointer to an array, so both are pointers. You can see why you'd want to store the relationship in the struct as a pointer; you don't really want another copy of the same `Post`—you want to point to the same `Post`.

Now that you've built the relationship, let's determine how you can use it. As mentioned earlier, this is a one-to-many as well as a many-to-one relationship. When you create a comment, you also want to create the relationship between the comment and the post it's meant for:

```
comment := Comment{Content: "Good post!", Author: "Joe", Post: &post}
comment.Create()
```

As before, you create a `Comment` struct, and then call the `Create` method on it. The relationship should be built upon creation of the comment. This means if you retrieve the post now, the relationship should be established.

```
readPost, _ := GetPost(post.Id)
```

The `readPost` variable should now have your newly minted comment in its Comments field. Next let's look at the `Comment` struct's `Create` method, shown in this listing.

Listing 6.15 Creating the relationship

```
func (comment *Comment) Create() (err error) {
    if comment.Post == nil {
        err = errors.New("Post not found")
        return
    }
    err = Db.QueryRow("insert into comments (content, author, post_id)
    ➥ values ($1, $2, $3) returning id", comment.Content, comment.Author,
    ➥ comment.Post.Id).Scan(&comment.Id)
    return
}
```

Before you create the relationship between the comment and the post, you need to make sure that the `Post` exists! If it doesn't, you'll return an error. The rest of the code repeats what we discussed earlier, except that now you also have to include post_id. Adding post_id will create the relationship.

With the relationship established, you want to be able to retrieve the post and be able to see the comments associated with the post. To do this, you'll modify the `GetPost` function as shown here.

Listing 6.16 Retrieving the relationship

```
func GetPost(id int) (post Post, err error) {
    post = Post{}
      post.Comments = []Comment{}
    err = Db.QueryRow("select id, content, author from posts where id =
    ➥ $1", id).Scan(&post.Id, &post.Content, &post.Author)

    rows, err := Db.Query("select id, content, author from comments where
    ➥ post_id = $1", id)
    if err != nil {
        return
    }
    for rows.Next() {
        comment := Comment{Post: &post}
        err = rows.Scan(&comment.Id, &comment.Content, &comment.Author)
        if err != nil {
            return
        }
        post.Comments = append(post.Comments, comment)
    }
    rows.Close()
    return
}
```

First, we need to initialize the Comments field in the `Post` struct and retrieve the post. We get all the comments related to this post and iterate through it, creating a `Comment` struct for each comment and appending it to the Comments field, and then return the post. As you can see, building up the relationships is not that difficult, though it can be a bit tedious if the web application becomes larger. In the next section, you'll see how relational mappers can be used to simplify establishing relationships.

Although we discussed the usual CRUD functions of any database application here, we've only scratched the surface of accessing an SQL database using Go. I encourage you to read the official Go documentation.

6.5 Go relational mappers

You've probably come to the conclusion that it's a lot of work storing data into the relational database. This is true in most programming languages; however, there's usually a number of third-party libraries that will come between the actual SQL and the calling application. In object-oriented programming (OOP) languages, these are often called object-relational mappers (ORMs). ORMs such as Hibernate (Java) and ActiveRecord (Ruby) map relational database tables and the objects in the programming language. But this isn't unique to object-oriented languages. Such mappers are

found in many other programming languages, too; for example, Scala has the Activate framework and Haskell has Groundhog.

In Go, there are a number of such relational mappers (ORMs doesn't sound as accurate to me). In this section, we'll discuss a couple of them.

6.5.1 Sqlx

Sqlx is a third-party library that provides a set of useful extensions to the `database/sql` package. Sqlx plays well with the `database/sql` package because it uses the same interfaces and provides additional capabilities such as:

- Marshaling database records (rows) into structs, maps, and slices using struct tags
- Providing named parameter support for prepared statements

The following listing shows how Sqlx makes life easier using the `StructScan` method. Remember to get the library before starting on the code by issuing the following command on the console:

```
go get "github.com/jmoiron/sqlx"
```

Listing 6.17 Using Sqlx

```
package main

import (
    "fmt"
    "github.com/jmoiron/sqlx"
    _ "github.com/lib/pq"
)

type Post struct {
    Id      int
    Content string
    AuthorName string `db: author`
}

var Db *sqlx.DB

func init() {
    var err error
    Db, err = sqlx.Open("postgres", "user=gwp dbname=gwp password=gwp
    ➥ sslmode=disable")
    if err != nil {
        panic(err)
    }
}

func GetPost(id int) (post Post, err error) {
    post = Post{}
    err = Db.QueryRowx("select id, content, author from posts where id =
    ➥ $1", id).StructScan(&post)
```

```
    if err != nil {
        return
    }
    return
}

func (post *Post) Create() (err error) {
    err = Db.QueryRow("insert into posts (content, author) values ($1, $2)
    ➥ returning id", post.Content, post.AuthorName).Scan(&post.Id)
    return
}

func main() {
    post := Post{Content: "Hello World!", AuthorName: "Sau Sheong"}
    post.Create()
    fmt.Println(post)                          <-- {1 Hello World! Sau Sheong}}
}
```

The code illustrating the difference between using Sqlx and `database/sql` is marked in bold; the other code should be familiar to you. First, instead of importing `database/sql`, you import `github.com/jmoiron/sqlx`. Normally, `StructScan` maps the struct field names to the corresponding lowercase table columns. To show how you can tell Sqlx to automatically map the correct table column to the struct field, listing 6.17 changed `Author` to `AuthorName` and used a struct tag (struct tags will be explained in further detail in chapter 7) to instruct Sqlx to get data from the correct column.

Instead of using `sql.DB`, you now use `sqlx.DB`. Both are similar, except `sqlx.DB` has additional methods like `Queryx` and `QueryRowx`.

In the `GetPost` function, instead of using `QueryRow`, you use `QueryRowx`, which returns `Rowx`. `Rowx` is the struct that has the `StructScan` method, and as you can see, it maps the table columns to the respective fields. In the `Create` method you're still using `QueryRow`, which isn't modified.

There are a few other features in Sqlx that are interesting but that I don't cover here. To learn more, visit the GitHub repository at https://github.com/jmoiron/sqlx.

Sqlx is an interesting and useful extension to `database/sql` but it doesn't add too many features. Conversely, the next library we'll explore hides the `database/sql` package and uses an ORM mechanism instead.

6.5.2 Gorm

The developer for Gorm delightfully calls it "the fantastic ORM for Go(lang)," and it's certainly an apt description. Gorm (Go-ORM) is an ORM for Go that follows the path of Ruby's ActiveRecord or Java's Hibernate. To be specific, Gorm follows the Data-Mapper pattern in providing mappers to map data in the database with structs. (In the relational database section earlier I used the ActiveRecord pattern.)

Gorm's capabilities are quite extensive. It allows programmers to define relationships, perform data migration, chain queries, and many other advanced features. It even has callbacks, which are functions that are triggered when a particular data event occurs, such as when data is updated or deleted. Describing the features would take

another chapter, so we'll discuss only basic features. The following listing explores code using Gorm. Our simple application is again in store.go.

Listing 6.18 Using Gorm

```
package main

import (
    "fmt"
    "github.com/jinzhu/gorm"
    _ "github.com/lib/pq"
    "time"
)

type Post struct {
    Id       int
    Content   string
    Author    string `sql:"not null"`
    Comments  []Comment
    CreatedAt time.Time
}

type Comment struct {
    Id       int
    Content   string
    Author    string `sql:"not null"`
    PostId    int    `sql:"index"`
    CreatedAt time.Time
}

var Db gorm.DB

func init() {
    var err error
    Db, err = gorm.Open("postgres", "user=gwp dbname=gwp password=gwp
    ➥ sslmode=disable")
    if err != nil {
        panic(err)
    }
    Db.AutoMigrate(&Post{}, &Comment{})
}

func main() {
    post := Post{Content: "Hello World!", Author: "Sau Sheong"}
    fmt.Println(post)

    Db.Create(&post)
    fmt.Println(post)

    comment := Comment{Content: "Good post!", Author: "Joe"}
    Db.Model(&post).Association("Comments").Append(comment)

    var readPost Post
    Db.Where("author = $1", "Sau Sheong").First(&readPost)
      var comments []Comment
```

{0 Hello World! Sau Sheong []
0001-01-01 00:00:00 +0000 UTC}

Creates a post

{1 Hello World! Sau Sheong []
2015-04-12 11:38:50.91815604 +0800 SGT}

Adds a comment

Gets comments from a post

```
      Db.Model(&readPost).Related(&comments)
fmt.Println(comments[0])
}
```

{1 Good post! Joe 1 2015-04-13 11:38:50.920377 +0800 SGT}

Note that the way that you create the database handler is similar to what you've been doing all along. Also note that we no longer need to set up the database tables separately using a setup.sql file. This is because Gorm has an automatic migration capability that creates the database tables and keeps them updated whenever you change the corresponding struct. When you run the program, the database tables that are needed will be created accordingly. To run this properly you should drop the database altogether and re-create it:

```
func init() {
    var err error
    Db, err = gorm.Open("postgres", "user=gwp dbname=gwp password=gwp
      sslmode=disable")
    if err != nil {
          panic(err)
    }
    Db.AutoMigrate(&Post{}, &Comment{})
}
```

The `AutoMigrate` method is a variadic method. A variadic method or function is a method or function that can take one or more parameters. Here it's called with references to the `Post` and `Comment` structs. If you change the structs to add in a new field, a corresponding table column will be created.

Let's take a look at one of the structs, `Comment`:

```
type Comment  struct {
    Id        int
    Content   string
    Author    string `sql:"not null"`
    PostId    int
    CreatedAt time.Time
}
```

There's a field called CreatedAt, which is of type `time.Time`. If you place this field in any struct, whenever a new record is created in the database it'll be automatically populated. In this case, this is when the record is created.

You'll also notice that some of the fields have struct tags which instruct Gorm to create and map to the correct fields. In the case of the Author field, the struct tag `` `sql: "not null"` `` tells Gorm to create a column that's not `null`.

Also notice that unlike our previous example, you didn't add a Post field in the `Comments` struct. Instead, you placed a PostId field. Gorm automatically assumes that a field in this form will be a foreign key and creates the necessary relationships.

So much for the setup. Now let's take a look at creating, and retrieving, posts and comments. First, create a post:

```
post := Post{Content: "Hello World!", Author: "Sau Sheong"}
Db.Create(&post)
```

Nothing surprising here. But as you can see you're using another construct, in this case the database handler `gorm.DB`, to create the `Post` struct, following the Data-Mapper pattern. This is unlike our previous example, when you called a `Create` method on the `Post` struct, following the ActiveRecord pattern.

If you inspect the database, you'll see that a timestamp column, created_at, will be populated with the date and time it was created.

Next, you want to add a comment to the post:

```
comment := Comment{Content: "Good post!", Author: "Joe"}
Db.Model(&post).Association("Comments").Append(comment)
```

You create a comment first, and then use a combination of the `Model` method, together with the `Association` and `Append` methods, to add the comment to the post. Notice that at no time are you manually accessing the PostId.

Finally, let's look at how you can retrieve the post and the comment you created:

```
var readPost Post
Db.Where("author = $1", "Sau Sheong").First(&readPost)
var comments []Comment
Db.Model(&readPost).Related(&comments)
```

Again, you use the `Where` method on `gorm.DB` to look for the first record that has the author name `"Sau Sheong"` and push it into the `readPost` variable. This will give you the post. To get the comments, you get the post model using the `Model` method, and then get the related comments into your `comments` variable using the `Related` method.

As mentioned earlier, what we've covered briefly in this section on Gorm is only a small portion of the rich features provided by this ORM library. If you're interested, learn more at https://github.com/jinzhu/gorm.

Gorm is not the only ORM library in Go. A number of equally feature-rich libraries are available, including Beego's ORM library and GORP (which isn't exactly an ORM but close enough).

In this chapter we've covered the basic building blocks of writing a web application. In the next chapter, we switch gears and discuss how we can build web services.

6.6 Summary

- Caching data in memory using structs, which allows you to cache data for quicker response
- Storing and retrieving data in files, in both CSV as well as gob binary format, which allows you to process user-uploaded data or provide backup for cached data
- Performing CRUD on relational databases using `database/sql` and establishing relationships between data
- Using third-party data-accessing libraries, including Sqlx and Gorm, which give you more powerful tools to manipulate data in the database