4

Serialization

In the previous chapters, we have learned how to scaffold Go microservices, create HTTP API endpoints, and set up service discovery to let our microservices communicate with each other. This knowledge already provides a solid foundation for building microservices; however, we are going to continue our journey with more advanced topics.

In this chapter, we will explore serialization, a process that allows the encoding and decoding of data for storing or sending between services. To illustrate how to use it, we are going to define data structures that will be transferred between our microservices using the popular and efficient Protocol Buffers format. Then, we are going to show how you can generate code the newly introduced data structures. Finally, we will demonstrate how fast our Protocol Buffers-based serialization is, compared to some other formats, such as XML and JSON. By the end of the chapter, you will learn how to leverage the serialization technique for service data encoding — this knowledge will help us for establishing efficient communication between our microservices in the following chapters.

In this chapter, we are going to cover the following topics:

* The basics of serialization
* Exploring popular serialization formats
* Using Protocol Buffers
* Serialization best practices

Technical requirements

To complete this chapter, you will need to have Go 1.18 or above and the Protocol Buffers compiler: https://grpc.io/docs/protoc-installation/.

You can find the code examples for this chapter on GitHub at the following link:

[https://github.com/PacktPublishing/Microservices-with-Go---Second-Edition/tree/main/Chapter04](https://github.com/PacktPublishing/Microservices-with-Go---Second-Edition%20/tree/main/Chapter04)

The basics of serialization

Serialization is the process of converting data into a format that allows you to transfer it, store it, and later deconstruct it.

Serialization has two primary use cases:

* Transferring the data between services, acting as a common format between them
* Encoding and decoding arbitrary data for storage, allowing you to store complex data structures as byte arrays or regular strings

In Chapter 2, while scaffolding our applications, we created our HTTP API endpoints and set them to return JSON responses to the callers. In that case, JSON played the role of a serialization format, allowing us to transform our data structures into it and then decode them back.

Let’s take our Metadata structure defined in the metadata/pkg/model/metadata.go file as an example:

// Metadata defines the movie metadata.

type Metadata struct {

    ID          string `json:"id"`

    Title       string `json:"title"`

    Description string `json:"description"`

    Director    string `json:"director"`

}

Our structure includes the records called annotations that help the JSON encoder transform our record into an output. For example, we create an instance of our structure:

Metadata{

    ID:          "123",

    Title:       "The Movie 2",

    Description: "Sequel of the legendary The Movie",

    Director:    "Foo Bars",

}

When we then encode it with JSON, the result would be the following:

{"id":"123","title":"The Movie 2","description":"Sequel of the legendary The Movie","director":"Foo Bars"}

Once the data is serialized, it can stored or transferred between services, even if they are written in different languages. In addition to data communication, serialization can be used for different other purposes:

* Store configuration: You can define your service settings using serialization formats, such as JSON, and then read them in your service code.
* Store records in a database: Serialization formats are frequently used for storing arbitrary data in databases. For example, key-value databases store records as strings or byte arrays, and developers often use serialization to encode and decode these record values.
* Logging: Application logs are often stored in JSON format, making them easy to read for both humans and various applications, such as data visualization software.

JSON is one of the most popular serialization formats at the moment and it has been essential to web development. It has the following benefits:

* Language support: Most programming languages include tools for encoding and decoding JSON data
* Browser support: JSON has been an integral part of web applications and all modern browsers include developer tools to work with it in the browser itself
* Readability: JSON records are easily readable and are often easy to use during both the development and debugging of web applications

However, it has certain limitations as well:

* Size: JSON is not a size-efficient format. In this chapter, we are going to see which formats and protocols provide output records that are smaller in size.
* Speed: As with its output size, the encoding and decoding speed with JSON is not the fastest set against other popular serialization protocols

There are many other popular serialization formats and protocols used in the industry. In the next section we are going to review some of the most popular ones.

Popular serialization formats

In addition to JSON, other popular serialization formats include:

* XML
* YAML
* Apache Thrift
* Apache Avro
* Protocol Buffers

This section will provide a high-level overview of each one as well as some key differences between these protocols.

XML

XML is one of the earliest serialization formats for web service development. It was created in 1998 and is still used in the industry, especially in enterprise applications.

XML represents data as a tree of nodes called elements. An element example would be <example>Some value</example>. If we serialized our metadata structure mentioned above, the result would be the following:

<Metadata><ID>123</ID><Title>The Movie 2</Title><Description>Sequel of the legendary The Movie</Description><Director>Foo Bars</Director></Metadata>

You may notice that the serialized XML representation of our data is slightly longer than the JSON one. It is one of the downsides of XML format – the output is often the largest among all popular serialization protocols, making it harder to read and transfer the data.

YAML

YAML is a serialization format that was first released in 2001. It gained popularity over the years, becoming one of the most popular serialization formats in the industry. Designers of the language took a strong focus on its readability and compactness, making it a perfect tool for defining arbitrary human-readable data. We can illustrate this on our Metadata structure: in the YAML format, it would look in the following way:

metadata:

  id: 123

  title: The Movie 2

  description: Sequel of the legendary The Movie

  director: Foo Bars

The YAML format is widely used for storing configuration data. One of the reasons for this is the ability to include comments, which is lacking in other formats, such as JSON. The use of YAML for service-to-service communication is less common, primarily due to the greater size of the serialized data.

Apache Thrift

So far, we have reviewed JSON, XML, and YAML, and all are primarily used for defining and serializing arbitrary types of data. There are other solutions to a broader class of problems, when we want not only to serialize and deserialize the data but also to transfer it between multiple services. These solutions combine two roles: they act as both serialization formats and communication protocols — mechanisms for sending and receiving arbitrary data over the network. HTTP is an example of such a protocol but developers are not limited to using it in their applications.

Apache Thrift is a combination of serialization and a communication protocol that can be used for both defining your data types and allowing your services to communicate with each other by passing them. It was initially created at Facebook but later became a community-supported open source project under the Apache Software Foundation.

Thrift, unlike JSON and XML, requires you to define your structures in their own format first. In our example, for the Metadata structure, we would need to create a file with the .thrift extension, including the definition in Thrift language:

struct Metadata {

  1: required string id,

  2: required string title,

  3: required string description,

  4: required string director

}

Once you have a Thrift file, you can use it with an automatic Thrift code generator to generate the code for most programming languages that would contain the defined structures and logic to encode and decode it. In addition to data structures, Thrift allows you to define Thrift services — sets of functions that can be called remotely. Here’s an example of a Thrift service definition:

service MetadataService {

  Metadata get(1: string id)

}

The example here defines a service called MetadataService, which provides a get function, returning a Metadata Thrift object. A Thrift-compatible server can act as such a Thrift service, processing incoming requests from the client applications — we are going to learn how to write such servers in Chapter 5.

Let's explore the benefits and limitations of Apache Thrift. The benefits include the following:

* A smaller output size and higher encoding and decoding speed compared to XML and JSON. As we are going to illustrate later in this chapter, Thrift-serialized data can be 30 to 50% smaller in size than XML and JSON
* The ability to define not only structures but entire services and generate code for them, allowing communication between the servers and their clients

The limitations include the following:

* Relatively low popularity and adoption in recent years due to moving to more popular and efficient formats.
* It lacks official documentation. Thrift is a relatively complex technology, and most documentation is unofficial.
* Unlike JSON and XML, Thrift-serialized data is not readable, so it’s trickier to use it for debugging.
* Nearly no support in recent years – Facebook keeps maintaining a separate branch of it called Facebook Thrift, but it is much less popular than its Apache counterpart.

Let's see the other popular serialization formats that are widely used across the industry.

Apache Avro

Apache Avro is a combination of a serialization format and a communication protocol that is somewhat similar to Apache Thrift. Apache Avro also requires a developer to define a schema (written either in JSON or in its own language called Avro IDL) for their data. In our case, the Metadata structure would have the following schema:

{

   "namespace": "example.avro",

   "type": "record",

   "name": "Metadata",

   "fields": [

      {"name": "id", "type": "string"},

      {"name": "title", "type": "string"},

      {"name": "description", "type": "string"},

      {"name": "director", "type": "string"},

   ]

}

Then, the schema would be used for translating the structures into a serialized state and back.

It is not uncommon for types and structures to change over time, and microservice API and structure definitions need to evolve. With Avro, developers can create a new version of a schema (represented as a separate file, often suffixed with an incremental version number), and keep both the old and new versions in the codebase. This way, the application can encode and decode data in either format, even if they have some incompatible changes, such as changes in field names. This is one of the key benefits of using Apache Avro over many other serialization protocols.

Protocol Buffers

Protocol Buffers is a serialization format that was created at Google more than 20 years ago. In 2008, the format became public and immediately gained popularity among developers. The benefits of the format include the following:

* The simplicity of the definition language
* A small data output size
* High performance of serialization and deserialization
* The ability to define services in addition to data structures and compile client and server code in multiple languages
* Protocol evolution and official support by Google

The popularity of Protocol Buffers and its simplicity, as well as the efficiency of its data encoding, makes it a great fit for using it in microservice development. We are going to use Protocol Buffers for serializing and deserializing the data transferred between our services, as well as defining our service APIs. In the next section, you will learn how to start using Protocol Buffers and move our microservice logic to Protocol Buffers from JSON.

Using Protocol Buffers

In this section, we are going to illustrate how you can use Protocol Buffers for your applications. We will use the microservice examples from the previous chapters and define our data model in the Protocol Buffers format. Then, we will be using the code generation tools with Protocol Buffers to generate our data structures. Finally, we will illustrate how to use our generated code for serializing and deserializing our data.

First, let’s prepare our application. Create the directory called api under our application’s src directory. Inside this directory, create a movie.proto file and add the following to it:

syntax = "proto3";

option go\_package = "/gen";

message Metadata {

    string id = 1;

    string title = 2;

    string description = 3;

    string director = 4;

}

message MovieDetails {

    double rating = 1;

    Metadata metadata = 2;

}

In the first line of our schema definition, we set the syntax to proto3, the latest version of the Protocol Buffers protocol. The second line defines the output path for the code generated. The rest of the file includes two structures that we need for our microservices, similar to the Go structures we created in Chapter 2.

Note

Unlike Apache Thrift, we don’t specify if Protocol Buffers message fields are required or optional. This wasn’t the case before the third version of the protocol, however the authors of Protocol Buffers intentionally removed this feature. The reason is, required fields are difficult to deprecate: if you set the field as optional and stop returning it from a server, a client that is not aware of the change might consider the data as invalid. To make serialization backward compatible, it is preferred to keep all fields as optional, and provide explicit documentation on data validation and supported message values.

Now, let’s generate the code for our structures. In the src directory of our application, run the following command:

protoc -I=api --go\_out=. movie.proto

If the command executes successfully, you should find a new directory called src/gen. The directory should include a file called movie.pb.go with the generated code that includes our structures and the code to serialize and deserialize them. For example, the generated MovieDetails structure code would be the following:

type Metadata struct {

    state         protoimpl.MessageState

    sizeCache     protoimpl.SizeCache

    unknownFields protoimpl.UnknownFields

    Id          string `protobuf:"bytes,1,opt,name=id,proto3" json:"id,omitempty"`

    Title       string `protobuf:"bytes,2,opt,name=title,proto3" json:"title,omitempty"`

    Description string `protobuf:"bytes,3,opt,name=description,proto3" json:"description,omitempty"`

    Director    string `protobuf:"bytes,4,opt,name=director,proto3" json:"director,omitempty"`

}

Let’s now describe what exactly we have just achieved. We have created a movie.proto file that defines our data schema — the definition of our data structures. The schema is now defined independently from our Go code, providing the following benefits to us:

* Explicit schema definition: Our data schema is now decoupled from the code and explicitly defines the application data types. This makes it easier to see the data types provided by application APIs.
* Code generation: Our schema can be converted to code via code generation. We are going to use it later in Chapter 5 for sending the data between the services.
* Cross-language support: We can generate our code not only for Go but also for other programming languages. If our model changes, we would not need to rewrite our structures for all languages. Instead, we can just re-generate the code for all languages by running a single command.

Let’s do a quick benchmark and compare the size of serialized data for three serialization protocols – XML, JSON, and Protocol Buffers. For this, let’s write a small tool to do so.

Inside the src directory, create a directory called cmd/sizecompare and add a main.go file to it with the following contents:

package main

import (

    "encoding/json"

    "encoding/xml"

    "fmt"

    "github.com/golang/protobuf/proto"

    "movieexample.com/gen"

    "movieexample.com/metadata/pkg/model"

)

var metadata = &model.Metadata{

    ID:          "123",

    Title:       "The Movie 2",

    Description: "Sequel of the legendary The Movie",

    Director:    "Foo Bars",

}

var genMetadata = &gen.Metadata{

    Id:          "123",

    Title:       "The Movie 2",

    Description: "Sequel of the legendary The Movie",

    Director:    "Foo Bars",

}

Let’s implement the main function:

func main() {

    jsonBytes, err := serializeToJSON(metadata)

    if err != nil {

        panic(err)

    }

    xmlBytes, err := serializeToXML(metadata)

    if err != nil {

        panic(err)

    }

    protoBytes, err := serializeToProto(genMetadata)

    if err != nil {

        panic(err)

    }

    fmt.Printf("JSON size:\t%dB\n", len(jsonBytes))

    fmt.Printf("XML size:\t%dB\n", len(xmlBytes))

    fmt.Printf("Proto size:\t%dB\n", len(protoBytes))

}

Additionally, add the following functions:

func serializeToJSON(m \*model.Metadata) ([]byte, error) {

    return json.Marshal(m)

}

func serializeToXML(m \*model.Metadata) ([]byte, error) {

    return xml.Marshal(m)

}

func serializeToProto(m \*gen.Metadata) ([]byte, error) {

    return proto.Marshal(m)

}

In the preceding code, we encode our Metadata structure using JSON, XML, and Protocol Buffers formats, and print the output sizes in bytes for each encoded result.

You may need to fetch the github.com/golang/protobuf/proto package required for our benchmark by running the following:

go mod tidy

Now, you can run our benchmark by executing go run \*.go inside its directory and will see following the output:

JSON size:       106B

XML size:       148B

Proto size:   63B

The result is quite interesting. The XML output is almost 40% bigger than the JSON one. At the same time, Protocol Buffers’s output is more than 40% smaller than the JSON data and more than twice as small as the XML result. This illustrates quite well how efficient the Protocol Buffers format is compared to the other two in terms of output size. By switching from JSON to Protocol Buffers, we reduce the amount of data that we need to send over the network and make our communication faster.

Let’s now do an additional experiment and test serialization speed for all three formats. For this, we are going to do a benchmark — an automated performance check that is going to measure how fast a target operation is.

Create a file called main\_test.go in the same directory and add the following to it:

package main

import (

    "testing"

)

func BenchmarkSerializeToJSON(b \*testing.B) {

    for i := 0; i < b.N; i++ {

        serializeToJSON(metadata)

    }

}

func BenchmarkSerializeToXML(b \*testing.B) {

    for i := 0; i < b.N; i++ {

        serializeToXML(metadata)

    }

}

func BenchmarkSerializeToProto(b \*testing.B) {

    for i := 0; i < b.N; i++ {

        serializeToProto(genMetadata)

    }

}

We have just created a Go benchmark, that is going to tell us how fast is JSON, XML, and Protocol Buffers encoding. We are going to cover the details of benchmarking in Chapter 8, let’s now run the code to see the output by executing the following command:

go test -bench=.

The result of the command should look as follows:

goos: darwin

goarch: amd64

pkg: movieexample.com/cmd/sizecompare

cpu: Intel(R) Core(TM) i7-8850H CPU @ 2.60GHz

BenchmarkSerializeToJSON-12          3308172           342.2 ns/op

BenchmarkSerializeToXML-12            480728          2519 ns/op

BenchmarkSerializeToProto-12         6596490           185.7 ns/op

PASS

ok      movieexample.com/cmd/sizecompare    5.239s

You can see the names of three functions that we just implemented and two numbers next to them:

* The first one is the number of times the function got executed
* The second is the average processing speed, measured in nanoseconds per operation

From the output, we can see that Protocol Buffers serialization on average took 185.7 nanoseconds, while JSON serialization was almost two times slower at 342.2 nanoseconds. XML serialization on average took 2519 nanoseconds, being more than 13 times slower than Protocol Buffers, and more than 7 times slower than JSON serialization.

The benchmark is indeed interesting – it illustrates how different the average encoding speeds for various serialization formats are. If performance is important for your services, you should consider faster serialization formats to achieve a higher encoding and decoding speed.

For now, we are going to leave the generated structures in our repository. We will be using them in the next chapter, Chapter 5, to replace our JSON API handlers.

Now, let’s learn some best practices for using serialization.

Serialization best practices

This section briefly summarizes the best practices for serializing and deserializing the data. These practices will help you to maintain and evolve your data schemas over time, as well as make them easy to use:

* Keep your schema backward compatible: Avoid any changes in your data schema that would break any existing callers. Such changes include modifications (renaming or removal) of field names and types.
* Ensure that data schemas are kept in sync between clients and servers: For serialization formats with explicit schema definitions, such as Apache Thrift, Protocol Buffers, and Apache Avro, you should keep clients and servers in sync with the latest schema versions.
* Document implicit details: Let the callers know any implicit details related to your data schema. For example, if your API does not allow an empty value of a certain field of a structure, include this in the comments in the schema file.
* Use built-in structures for representing time whenever possible: Protocol Buffers and some other serialization protocols provide built-in types for timestamps and durations. Taking Protocol Buffers as an example, having a int timestamp filed would be considered a bad practice. The right approach would be to use google.protobuf.Timestamp.
* Use consistent naming: Opt for using consistent naming in your schema files, similarly to your code.
* Follow the official style guide: Get familiar with the official style guide if you are using a schema definition language, such as Thrift or Protocol Buffers. You can find the link to the official style guide for Protocol Buffers in the Further reading section below.

This list provides some high-level recommendations applicable to all serialization protocols. For protocol-specific recommendations, follow the official documentation and check the popular open source projects to get some real-world code examples.

Summary

In this chapter, we covered the basics of serialization and illustrated how our data structures could be encoded using various serialization protocols, including XML, JSON, and Protocol Buffers. You learned about the differences between the most popular serialization protocols and their main advantages and disadvantages.

We also covered the basics of Protocol Buffers serialization format and showed how to define custom data structures in its schema definition language. We used the example code to illustrate how to generate the schema files for the Go language. Finally, we compared the performance of XML, JSON, and Protocol Buffers formats, and discussed some common data serialization best practices.

In the next chapter, we are going to demonstrate how to use Protocol Buffers-based serialization for communication between services.

Further reading

* The Protocol Buffers documentation: <https://developers.google.com/protocol-buffers>
* The Protocol Buffers official style guide: https://developers.google.com/protocol-buffers/docs/style