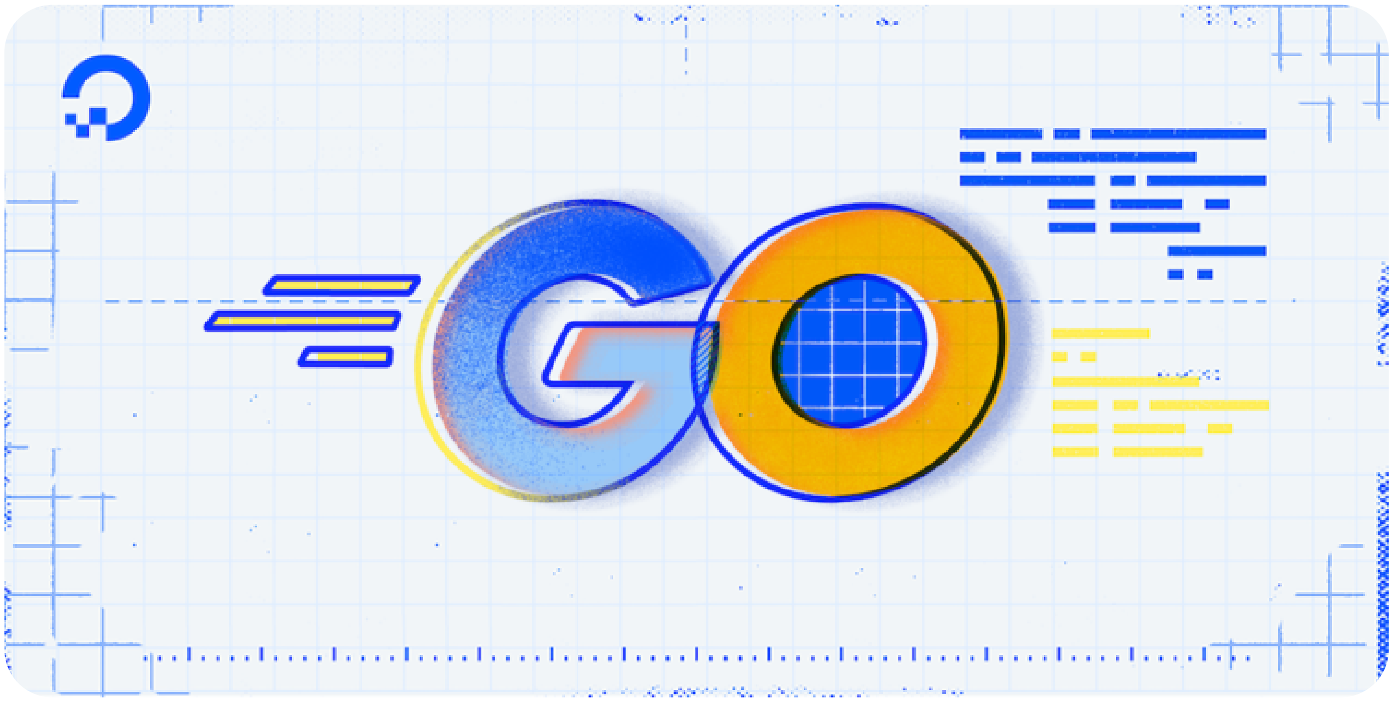
How To Use Contexts in Go

Published on February 15, 2022

Development Go

By Kristin Davidson
Bit Transducer



The author selected the Diversity in Tech Fund to receive a donation as part of the Write for DOnations program.

Introduction

When developing a large application, especially in server software, sometimes it's helpful for a function to know more about the environment it's being executed in aside from the information needed for a function to work on its own. For example, if a web server function is handling an HTTP request for a specific client, the function may only need to know which URL the client is requesting to serve the response. The function might only take that URL as a parameter. However, things can always happen when serving a response, such as the client disconnecting before receiving the response. If the function serving the response doesn't know the client disconnected, the server software may end up spending more computing time than it needs calculating a response that will never be used.

In this case, being aware of the context of the request, such as the client's connection status, allows the server to stop processing the request once the client disconnects. This saves valuable compute resources on a busy server and frees them up to handle another client's request. This type of information can also be helpful in other contexts where functions take time to execute, such as making database calls. To enable ubiquitous access to this type of information, Go has included a `context` package in its standard library.

In this tutorial, you will start by creating a Go program that uses a context within a function. Then, you will update that program to store additional data in the context and retrieve it from another function. Finally, you will use a context's ability to signal it's done to stop processing additional data.

Prerequisites

- Go version 1.16 or greater installed. To set this up, follow the How To Install Go tutorial for your operating system.
- An understanding of goroutines and channels, which you can find in the tutorial, How to Run Multiple Functions Concurrently in Go.
- Familiarity with using dates and times in Go, which you can find in the tutorial, How to Use Dates and Times in Go.
- Experience with the `switch` statement, which you can read more about in the tutorial, How To Write Switch Statements in Go.

Creating a Context

Many functions in Go use the `context` package to gather additional information about the environment they're being executed in, and will typically provide that context to the functions they also call. By using the `context.Context` interface in the `context` package and passing it from function to function, programs can convey that information from the beginning function of a program, such as `main`, all the way to the deepest function call in the program. The `Context` function of an `http.Request`, for example, will provide a `context.Context` that includes information about the client making the request and will end if the client disconnects before the request is finished. By passing this `context.Context` value into a function that then makes a call to the `QueryContext` function of a `sql.DB`, the database query will also be stopped if it's still running when the client disconnects.

In this section, you will create a program with a function that receives a context as a parameter. You will also call that function using an empty context you create with the `context.TODO` and `context.Background` functions.

To get started using contexts in a program, you'll need to have a directory to keep the program in. Many developers keep their projects in a directory to keep them organized. In this tutorial, you'll use a directory named `projects`.

First, make the `projects` directory and navigate to it:

```
mkdir projects
cd projects
```

Copy

Next, make the directory for your project. In this case, use the directory `contexts`:

```
mkdir contexts
cd contexts
```

Copy

Inside the `contexts` directory use `nano`, or your favorite editor, to open the `main.go` file:

```
nano main.go
```

Copy

In the `main.go` file, you'll create a `doSomething` function that accepts a `context.Context` as a parameter. Then, you'll add a `main` function that creates a context and calls `doSomething` using that context.

Add the following lines to `main.go`:

projects/contexts/main.go

```
package main

import (
	"context"
	"fmt"
)

func doSomething(ctx context.Context) {
	fmt.Println("Doing something!")
}

func main() {
	ctx := context.TODO()
	doSomething(ctx)
}
```

Copy

In the `main` function, you used the `context.TODO` function, one of two ways to create an empty (or starting) context. You can use this as a placeholder when you're not sure which context to use.

In this code, the `doSomething` function you added accepts a `context.Context` as its only parameter, even though it doesn't quite do anything with it yet. The variable's name is `ctx`, which is commonly used for context values. It's also recommended to put the `context.Context` parameter as the first parameter in a function, and you'll see it there in the Go standard library. But that doesn't apply yet because it's the only parameter for `doSomething`.

To run your program, use the `go run` command on the `main.go` file:

```
go run main.go
```

Copy

The output will look similar to this:

```
Output
Doing something!
```

The output shows that your function was called and printed `Doing something!` using the `fmt.Println` function.

Now, open your `main.go` file again and update your program to use the second function that will create an empty context, `context.Background`:

projects/contexts/main.go

```
...

func main() {
	ctx := context.Background()
	doSomething(ctx)
}
```

Copy

The `context.Background` function creates an empty context like `context.TODO` does, but it's designed to be used where you intend to start a known context. Fundamentally the two functions do the same thing: they return an empty context that can be used as a `context.Context`. The biggest difference is how you signal your intent to other developers. If you're unsure which one to use, `context.Background` is a good default option.

Now, run your program again using the `go run` command:

```
go run main.go
```

Copy

The output will look similar to this:

```
Output
```

```
Doing something!
```

Your output will be the same because the code's functionality didn't change, only what a developer would see when reading the code.

In this section, you created an empty context using both the `context.TODO` and the `context.Background` functions. However, an empty context isn't entirely useful to you if it stays that way. You can pass them to other functions to use, but if you want to use them in your own code, all you'd have so far is an empty context. One of the things you can do to add more information to a context is to add data that can be retrieved from them in other functions, which you'll do in the next section.

Using Data Within a Context

One benefit of using `context.Context` in a program is the ability to access data stored inside a context. By adding data to a context and passing the context from function to function, each layer of a program can add additional information about what's happening. For example, the first function may add a username to the context. The next function may add the file path to the content the user is trying to access. Finally, a third function could then read the file from the system's disk and log whether it was loaded successfully or not as well as which user tried to load it.

To add a new value to a context, use the `context.WithValue` function in the `context` package. The function accepts three parameters: the parent `context.Context`, the key, and the value. The parent context is the context to add the value to while preserving all the other information about the parent context. The key is then used to retrieve the value from the context. The key and the value can be any data type, but this tutorial will be using `string` keys and values. The `context.WithValue` will then return a new `context.Context` value with the value added to it.

Once you have a `context.Context` with a value added to it, you can access those values using the `Value` method of a `context.Context`. Providing the `Value` method with a key will return the value stored.

Now, open your `main.go` file again and update it to add a value to the context using `context.WithValue`. Then, update the `doSomething` function to print that value to the output using `fmt.Printf`:

projects/contexts/main.go

Copy

```
...

func doSomething(ctx context.Context) {
	fmt.Printf("doSomething: myKey's value is %s\n", ctx.Value("myKey"))
}
```

```
func main() {
	ctx := context.Background()

	ctx = context.WithValue(ctx, "myKey", "myValue")

	doSomething(ctx)
}
```

In this code, you're assigning the new context back to the `ctx` variable that is used to hold the parent context. This is a relatively common pattern to use if you don't have a reason to reference a specific parent context. If you did need access to the parent context as well, you can assign this value to a new variable, as you'll see shortly.

To see the program's output, run it using the `go run` command:

```
go run main.go
```

Copy

The output will look similar to this:

```
Output
doSomething: myKey's value is myValue
```

In the output, you'll see the value you stored in the context from the `main` function is now also available in the `doSomething` function. In a larger program running on a server, this value could be something like the time the program started running, or the server the program is running on.

When using contexts, it's important to know that the values stored in a specific `context.Context` are immutable, meaning they can't be changed. When you called the `context.WithValue`, you passed in the parent context and you also received a context back. You received a context back because the `context.WithValue` function didn't modify the context you provided. Instead, it wrapped your parent context inside another one with the new value.

To see how this works, update your `main.go` file to add a new `doAnother` function that accepts a `context.Context` and prints out the `myKey` value from the context. Then, update `doSomething` to set its own `myKey` value (`anotherValue`) in the context and calls `doAnother` with the resulting `anotherCtx` context. Finally, have `doSomething` print out the `myKey` value from its original context again:

projects/contexts/main.go

Copy

```
...

func doSomething(ctx context.Context) {
	fmt.Printf("doSomething: myKey's value is %s\n", ctx.Value("myKey"))

	anotherCtx := context.WithValue(ctx, "myKey", "anotherValue")
	doAnother(anotherCtx)

	fmt.Printf("doSomething: myKey's value is %s\n", ctx.Value("myKey"))
}

func doAnother(ctx context.Context) {
	fmt.Printf("doAnother: myKey's value is %s\n", ctx.Value("myKey"))
}

...
```

Next, run your program again using the `go run` command:

Copy

```
go run main.go
```

The output will look similar to this:

```
Output
doSomething: myKey's value is myValue
doAnother: myKey's value is anotherValue
doSomething: myKey's value is myValue
```

This time in the output, you'll see two lines from the `doSomething` function and one line from the `doAnother` function. In your `main` function, you set `myKey` to a value of `myValue` and passed it to the `doSomething` function. You can see in the output that `myValue` made it into the function.

The next line, though, shows that when you used `context.WithValue` in `doSomething` to set `myKey` to `anotherValue` and passed the resulting `anotherCtx` context to `doAnother`, the new value overrode the initial value.

Finally, on the last line, you'll see when you print out the `myKey` value again from the original context, the value is still `myValue`. Since the `context.WithValue` function only wraps the parent context, the parent context still has all the same values it originally did. When you use the `Value` method on a context, it will find the outer-most wrapped value for the given key and return that value. In your code, when you call `anotherCtx.Value` for `myKey` it will return `anotherValue` because it's the outer-most wrapped value for the context, effectively overriding any other `myKey` values being wrapped. When you call `ctx.Value` inside `doSomething` a second time, `anotherCtx` isn't wrapping `ctx`, so the original `myValue` value is returned.

> **Note:** Contexts can be a powerful tool with all the values they can hold, but a balance needs to be struck between data being stored in a context and data being passed to a function as parameters. It may seem tempting to put all of your data in a context and use that data in your functions instead of parameters, but that can lead to code that is hard to read and maintain. A good rule of thumb is that any data required for a function to run should be passed as parameters. Sometimes, for example, it can be useful to keep values such as usernames in context values for use when logging information for later. However, if the username is used to determine if a function should display some specific information, you'd want to include it as a function parameter even if it's already available from the context. This way when you, or someone else, looks at the function in the future, it's easier to see which data is actually being used.

In this section, you updated your program to store a value in a context, and then wrapped that context to override the value. As mentioned in an example earlier, though, this isn't the only valuable tool contexts can provide. They can also be used to signal to other parts of a program when they should stop processing to avoid unnecessary resource usage.

Ending a Context

Another powerful tool `context.Context` provides is a way to signal to any functions using it that the context has ended and should be considered complete. By signaling to these functions that the context is done, they know to stop processing any work related to the context that they may still be working on. Using this feature of a context allows your programs to be more efficient because instead of fully completing every function, even if the result will be thrown out, that processing time can be used for something else. For example, if a web page request comes to your Go web server, the user may end up hitting the “Stop” button or closing their browser before the page finishes loading. If the page they requested requires running a few database queries, the server may run the queries even though the data won't be used. However, if your functions are using a `context.Context`, your functions will know the context is done because Go's web server will cancel it, and that they can skip running any other database queries they haven't already run. This will free up web server and database server processing time so it can be used on a different request instead.

In this section, you will update your program to tell when a context is done, and use three different methods to end a context.

Determining if a Context is Done

No matter the cause of a context ending, determining if a context is done happens the same way. The `context.Context` type provides a method called `Done` that can be checked to see whether a context has ended or not. This method returns a channel that is closed when the context is done, and any functions watching for it to be closed will know they should consider their execution context completed and should stop any processing related to the context. The `Done` method works because no values are ever written to its channel, and when a channel is closed that channel will start to return `nil` values for every read attempt. By periodically checking whether the `Done` channel has closed and doing processing work in-between, you're able to implement a function that can do work but also knows if it should stop processing early. Combining this processing work, the periodic check of the `Done` channel, and the `select` statement goes even further by allowing you to send data to or receive data from other channels simultaneously.

The `select` statement in Go is used to allow a program to try reading from or writing to a number of channels all at the same time. Only one channel operation happens per `select` statement, but when performed in a loop, the program can do a number of channel operations when one becomes available. A `select` statement is created by using the keyword `select`, followed by a code block enclosed in curly braces (`{}`), with one or more `case` statements inside the code block. Each `case` statement can be either a channel read or write operation, and the `select` statement will block until one of the `case` statements can be executed. Suppose you don't want the `select` statement to block, though. In that case, you can also add a `default` statement that will be executed immediately if none of the other `case` statements can be executed. It looks and works similarly to a `switch` statement, but for channels.

The following code example shows how a `select` statement could potentially be used in a long-running function that receives results from a channel, but also watches for when a context's `Done` channel is closed:

Copy

```go
ctx := context.Background()
resultsCh := make(chan *WorkResult)

for {
	select {
	case <- ctx.Done():
		// The context is over, stop processing results
		return
	case result := <- resultsCh:
		// Process the results received
	}
}
```

In this code, the values of `ctx` and `resultsCh` would normally be passed into a function as parameters, with `ctx` being the `context.Context` the function is executing in and `resultsCh` being a read-only channel of results from a worker goroutine elsewhere. Every time the `select` statement is run, Go will stop running the function and watch all the `case` statements. When one of the `case` statements can be executed, whether it's reading from a channel in the case of `resultsCh`, writing to a channel, or a channel being closed in the case of the `Done` channel, that branch of the `select` statement is executed. However, it's not guaranteed which order the `case` statements execute if more than one of them can run simultaneously.

For the code execution in this example, the `for` loop will continue forever until the `ctx.Done` channel is closed because the only `return` statement is inside that `case` statement. Even though the `case <-ctx.Done` doesn't assign values to any variables, it will still be triggered when `ctx.Done` is closed because the channel still has a value that can be read even if it's ignored. If the `ctx.Done` channel isn't closed, the `select` statement will wait until it is, or if `resultsCh` has a value that can be read. If `resultsCh` can be read, then that `case` statement's code block will be executed instead. Since there's no guaranteed order, if both can be read, it would seem random which one is executed.

If the example's `select` statement had a `default` branch without any code in it, it wouldn't actually change how the code works, it would just cause the `select` statement to end right away and the `for` loop would start another iteration of the `select` statement. This leads to the `for` loop executing very quickly because it will never stop and wait to read from a channel. When this happens, the `for` loop is called a *busy loop* because instead of waiting for something to happen, the loop is busy running over and over again. This can consume a lot of CPU because the program never gets a chance to stop running to let other code execute. Sometimes this functionality is useful, though, such as if you want to check if a channel is ready to do something before you go and do another non-channel operation.

Since the only way to exit the `for` loop in the example code is to close the channel returned by `Done`, and the only way to close the `Done` channel is by ending the context, you'll need a way to end the context. The Go `context` package provides a few ways to do this depending on your goal, and the most direct option is to call a function to "cancel" the context.

Canceling a Context

Canceling a context is the most straightforward and controllable way to end a context. Similar to including a value in a context with `context.WithValue`, it's possible to associate a "cancel" function with a context using the `context.WithCancel` function. This function receives a parent context as a parameter and returns a new context as well as a function that can be used to cancel the returned context. Also, similar to `context.WithValue`, calling the cancel function returned will only cancel the context returned and any contexts that use it as a parent context. This doesn't prevent the parent context from being canceled, it just means that calling your own cancel function won't do it.

Now, open your `main.go` file and update your program to use `context.WithCancel` and the cancel function:

projects/contexts/main.go

Copy

```
package main

import (
	"context"
	"fmt"
	"time"
)

func doSomething(ctx context.Context) {
	ctx, cancelCtx := context.WithCancel(ctx)

	printCh := make(chan int)
	go doAnother(ctx, printCh)

	for num := 1; num <= 3; num++ {
		printCh <- num
	}

	cancelCtx()

	time.Sleep(100 * time.Millisecond)

	fmt.Printf("doSomething: finished\n")
}

func doAnother(ctx context.Context, printCh <-chan int) {
	for {
		select {
		case <-ctx.Done():
			if err := ctx.Err(); err != nil {
				fmt.Printf("doAnother err: %s\n", err)
			}
			fmt.Printf("doAnother: finished\n")
			return
		case num := <-printCh:
			fmt.Printf("doAnother: %d\n", num)
		}
	}
}

...
```

First, you add an import for the `time` package and change the `doAnother` function to accept a new channel of numbers to print to the screen. Next, you use a `select` statement inside a `for` loop to read from that channel as well as the context's `Done` method. Then, in the `doSomething` function, you create a context that can be canceled as well as a channel to send numbers to, and run `doAnother` as a goroutine with those as parameters. Finally, you send a few numbers to the channel and cancel the context.

To see this code running, use the `go run` command as you have before:

```
go run main.go
```

Copy

The output will look similar to this:

```
Output
doAnother: 1
doAnother: 2
doAnother: 3
doAnother err: context canceled
doAnother: finished
doSomething: finished
```

In this updated code, your `doSomething` function acts like a function that sends work to one or more goroutines reading from a work channel. In this case, `doAnother` is the worker and printing the numbers is the work it's doing. Once the `doAnother` goroutine is started, `doSomething` begins sending numbers to be printed. Inside the `doAnother` function, the `select` statement is waiting for either the `ctx.Done` channel to close or for a number to be received on the `printCh` channel. The `doSomething` function sends three numbers on the channel, triggering the `fmt.Printf` for each number, and then calls the `cancelCtx` function to cancel the context. After the `doAnother` function reads the three numbers from the channel, it will wait for the next channel operation. Since the next thing to happen is `doSomething` calling `cancelCtx`, the `ctx.Done` branch is called.

When the `ctx.Done` branch is called, the code uses the `Err` function provided by `context.Context` to determine how the context ended. Since your program is using the `cancelCtx` function to cancel the context, the error you see in the output is `context canceled`.

> **Note:** If you've run Go programs before and looked at the logging output, you may have seen the `context canceled` error in the past. When using the Go http package, this is a common error to see when a client disconnects from the server before the server handles the full response.

Once the `doSomething` function has canceled the context, it uses the `time.Sleep` function to wait a short amount of time to give `doAnother` time to process the canceled context and finish running. After that, it exits the function. In many cases the `time.Sleep` would not be required, but it's needed since the example code finishes executing so quickly. If `time.Sleep` isn't included, the program may end before you see the rest of the program's output on the screen.

The `context.WithCancel` function and the cancel function it returns are most useful when you want to control exactly when a context ends, but sometimes you don't want or need that amount of control. The next function available to end contexts in the `context` package is `context.WithDeadline` and is the first function that will automatically end a context for you.

Giving a Context a Deadline

Using `context.WithDeadline` with a context allows you to set a deadline for when the context needs to be finished, and it will automatically end when that deadline passes. Setting a deadline for a context is similar to setting a deadline for yourself. You tell the context the time it needs to be finished, and Go automatically cancels the context for you if that time is exceeded.

To set a deadline on a context, use the `context.WithDeadline` function and provide it the parent context and a `time.Time` value for when the context should be canceled. You'll then receive a new context and a function to cancel the new context as return values. Similar to `context.WithCancel`, when a deadline is exceeded, it will only affect the new context and any others that use it as a parent context. The context can also be canceled manually by calling the cancel function the same as you would for `context.WithCancel`.

Next, open your `main.go` file and update it to use `context.WithDeadline` instead of `context.WithCancel`:

projects/contexts/main.go

Copy

```
...

func doSomething(ctx context.Context) {
	deadline := time.Now().Add(1500 * time.Millisecond)
	ctx, cancelCtx := context.WithDeadline(ctx, deadline)
	defer cancelCtx()

	printCh := make(chan int)
	go doAnother(ctx, printCh)

	for num := 1; num <= 3; num++ {
		select {
		case printCh <- num:
			time.Sleep(1 * time.Second)
		case <-ctx.Done():
			break
		}
	}

	cancelCtx()

	time.Sleep(100 * time.Millisecond)
```

```
        fmt.Printf("doSomething: finished\n")
}

...
```

Your updated code now uses `context.WithDeadline` in `doSomething` to automatically cancel the context 1500 milliseconds (1.5 seconds) after the function starts by using the `time.Now` function. In addition to updating the way the context is finished, a few other changes have been made. Since the code can now potentially end by calling `cancelCtx` directly or an automatic cancelation via the deadline, the `doSomething` function has been updated to use a `select` statement to send the numbers on the channel. This way, if whatever is reading from `printCh` (in this case `doAnother`) isn't reading from the channel and the `ctx.Done` channel is closed, the `doSomething` function will also notice it and stop trying to send numbers.

You'll also notice that `cancelCtx` is called twice, once via a new `defer` statement and another time in the place it was before. The `defer cancelCtx()` isn't necessarily required because the other call will always be run, but it can be useful to keep it in case there are any `return` statements in the future that cause it to be missed. When a context is canceled from a deadline, the cancel function is still required to be called in order to clean up any resources that were used, so this is more of a safety measure.

Now, run your program again using `go run`:

```
go run main.go
```
Copy

The output will look similar to this:

```
Output
doAnother: 1
doAnother: 2
doAnother err: context deadline exceeded
doAnother: finished
doSomething: finished
```

This time in the output you'll see that the context was canceled due to a `context deadline exceeded` error before all three numbers were printed. Since the deadline was set to 1.5 seconds after the `doSomething` function started running and `doSomething` is set to wait one second after sending each number, the deadline will be exceeded before the third number is printed. Once the deadline is exceeded, both the `doAnother` and `doSomething` functions finish running because they're both watching for the `ctx.Done` channel to close. You can also tweak the amount of time added to `time.Now` to see how various deadlines affect the output. If the deadline ends up at or over 3 seconds, you could even see the error change back to the `context canceled` error because the deadline is no longer being exceeded.

The `context deadline exceeded` error may also be familiar to you if you've used Go applications or looked at their logs in the past. This error is common to see in web servers that take some time to complete sending responses to the client. If a database query or some processing takes a long time, it could cause the web request to take longer than is allowed by the server. Once the request surpasses the limit, the request's context would be canceled and you'd see the `context deadline exceeded` error message.

Ending a context using `context.WithDeadline` instead of `context.WithCancel` allows you to specify a specific time when a context needs to end without needing to keep track of that time yourself. If you know the `time.Time` when a context should end, `context.WithDeadline` is likely a good candidate for managing your context's endpoint. Other times, you don't care about the specific time a context ends and you just know you want it to end one minute after it's started. It's possible to do this using `context.WithDeadline` and other `time` package functions and methods, but Go also provides the `context.WithTimeout` function to make that work easier.

Giving a Context a Time Limit

The `context.WithTimeout` function can almost be considered more of a helpful function around `context.WithDeadline`. With `context.WithDeadline` you provide a specific `time.Time` for the context to end, but by using the `context.WithTimeout` function you only need to provide a `time.Duration` value for how long you want the context to last. This will be what you're looking for in many cases, but `context.WithDeadline` is available if you need to specify a `time.Time`. Without `context.WithTimeout` you would need to use `time.Now()` and the `time.Time`'s `Add` method on your own to get the specific time to end.

One last time, open your `main.go` file and update it to use `context.WithTimeout` instead of `context.WithDeadline`:

projects/contexts/main.go

Copy

```
...

func doSomething(ctx context.Context) {
	ctx, cancelCtx := context.WithTimeout(ctx, 1500*time.Millisecond)
	defer cancelCtx()

	...
}

...
```

Once you have the file updated and saved, run it using `go run`:

```
go run main.go
```

Copy

The output will look similar to this:

```
Output
doAnother: 1
doAnother: 2
doAnother err: context deadline exceeded
doAnother: finished
doSomething: finished
```

When running the program this time, you should see the same output as when you used `context.WithDeadline`. The error message is even the same, showing you that `context.WithTimeout` is really just a wrapper to do the `time.Now` math for you.

In this section, you updated your program to use three different ways to end a `context.Context`. The first, `context.WithCancel`, allowed you to call a function to cancel the context. Next, you used `context.WithDeadline` with a `time.Time` value to automatically end the context at a certain time. Lastly, you used `context.WithTimeout` and a `time.Duration` value to automatically end the context after a certain amount of time has elapsed. Using these functions, you'll be able to ensure your programs don't consume more resources than needed on your computers. Understanding the errors they cause contexts to return will also make troubleshooting errors in your own and other Go programs easier.

Conclusion

In this tutorial, you created a program to interact with Go's `context` package in various ways. First, you created a function that accepts a `context.Context` value as a parameter and used the `context.TODO` and `context.Background` functions to create empty contexts. After that you used `context.WithValue` to add values to new contexts and used the `Value` method to retrieve them in other functions. Lastly, you used a context's `Done` method to determine when a context was done running. When paired with the functions `context.WithCancel`, `context.WithDeadline`, and `context.WithTimeout`, you implemented your own context management to set limits on how long code using those contexts should run.

If you'd like to learn more about how contexts work with more examples, the Go `context` package documentation includes additional information.

This tutorial is also part of the DigitalOcean How to Code in Go series. The series covers a number of Go topics, from installing Go for the first time to how to use the language itself.