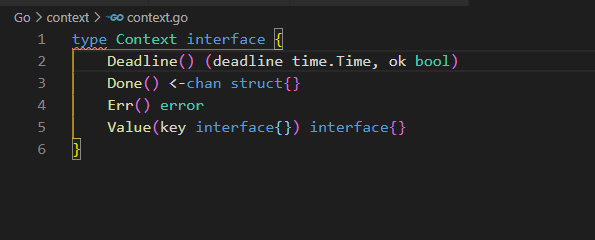
A context is a deadline you can pass into a running process in your code. This deadline can indicate to a process to stop running and return after a condition is met. This becomes useful when reaching out to external APIs, databases as shown above, or system commands.

The following supposes that the reader knows about goroutines and channels and how they work together. I am going to deep dive into concurrency after writing about context as the context library is part of concurrency. For now, though, goroutines are lightweight threads that can be started for processes and channels are the pipelines used to pass data between these new processes.

**Context Interface**

The context library defines a new interface called Context.



The Deadline field returns the expected time the work is finished and indicates when the context should be cancelled.

The Done field is a channel that is closed when work done for the context should be cancelled. This operation can happen asynchronously. The channel can return as nil if the associated context can never be cancelled. Different context types will arrange for work to be cancelled depending on the circumstances, which we will get into.

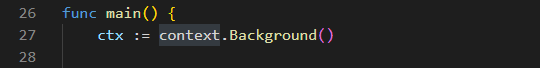
Err will return nil until Done is closed. After which Err will either return Canceled if the context was canceled or DealineExceeded if the context’s deadline has passed.

The Value field is a key-value interface which will return a value associated with the context as a key or nil if there was no value associated. Values should be used carefully as they are not for passing parameters into a function but for [request-scoped data transits processes and API boundaries](https://github.com/golang/go/blob/master/src/context/context.go#L185).

### Context in context

**Context.Background**

The “Background” function returns an empty non-nil context. There is no associated deadline and no cancelation to speak of. This can be typically used in the main function, for testing, or for creating a top-level context to be made into something else. Looking into the source code you can see that it doesn’t have any logic other than returning an [empty context](https://github.com/golang/go/blob/master/src/context/context.go#L208):

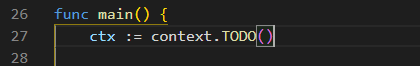


QuickNote:  
Typically, the context is named ctx when it is declared. I’ve seen this in most implementations of context so if you come across ctx in random spots in source code there’s a good chance that it is referring to a context.

### Context.TODO

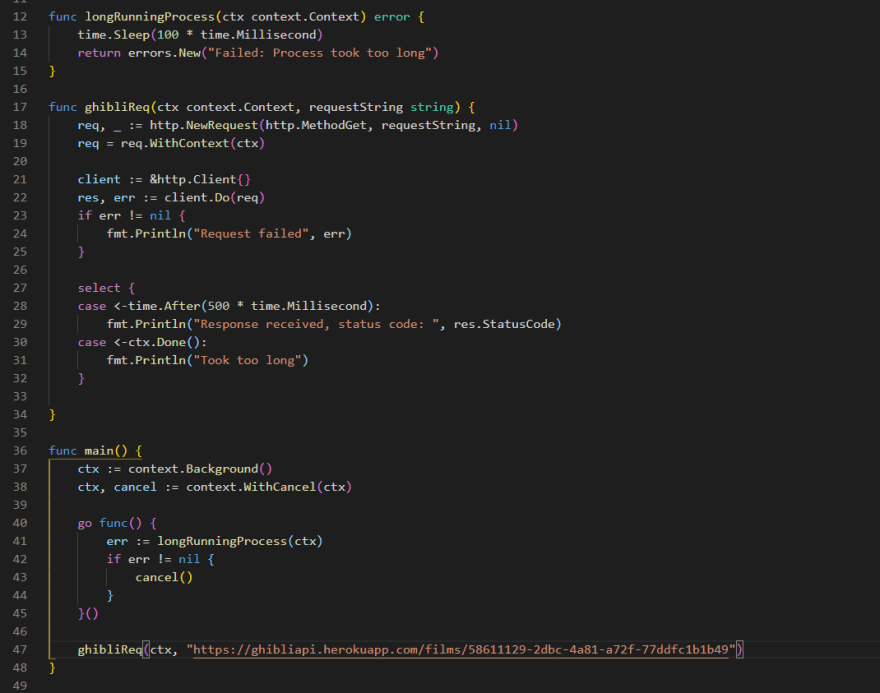
The TODO function does the same thing. It returns an empty non-nil context. This again is a use case for higher-level functions that may not yet have a function available to use them. In many cases, this would be used as a placeholder when extending your program to use the context library. If you checked out the talk by Sameer Ajmani about the introduction of the context library while refactoring their code at Google they would use the context.TODO to start introducing context into the Google code base without breaking anything.

QuickNote:  
One thing I will also mention is that somewhere along the way it was suggested that the TODO would be compatible for use in static analysis tools for seeing context propagation across a program.



### Context.WithCancel

Let’s say I’m building a website to review movies. There is a myriad of APIs designed for serving movie information. One of the recent ones I’ve come across is the [Studio Ghibli API](https://ghibliapi.herokuapp.com/#section/Studio-Ghibli-API) which is a public API we can just grab stuff from. So, for the special section of the website for Studio Ghibli movies, we’ll use this. The WithCancel function returns a copy of the parent context passed into it with a new Done channel. The new Done channel is closed either when the cancel function is called or when the parent context’s Done channel is closed. Whichever event happens first.



Here we are going to simulate a process that is hanging up using the longRunningProcess function. In this example, the function is screwing up but we must run it before we request the JSON data from the API. The "longRunningProcess\* function will return an error that will cause the cancel() function within the context to fire.

For the ghibliReq function we will set up a simple HTTP request using the API and pass a string for locating stuff from the API. Once we set up the request, we have a case statement which will receive channel data. Depending on what happens first the select statement will be sent either the current time or the “Done” channel from the passed in context. If the Done channel is closed we error out, if not we will return the status code from our request.

Our main code starts with setting up the context with a new Background() context which is then passed into a WithCancel() context. The new ctx was passed in an empty context so nothing has happened yet. We then create a new goroutine to create a new thread and call our longRunningProcess. Once that is called we check for errors, which will return since we engineered it that way, and if there are errors we can call the cancel() function in our context. Finally, we use our context to call our request. After we run this we find that the request errored out since it took too long and the cancel() function was called.

In this example, we are running our longRunningProcess before our request because that is needed before we call our request. If the function errors out we need to be able to call “cancel()” so that we can error out the ghibliReq() function. The way we set it up we are calling cancel for our context before the function has a chance to run. This is intentional to show how the cancel works. We could easily change the time.Sleep() in longRunningProcess to say 1000 milliseconds and our request function will run before cancel() is called but in a production environment if the goal is to make sure we maintain the flow of the call stack we would make sure we’re not returning errors and not calling cancel() for this context.

QuickNote:  
Keep in mind that a context-specific call shouldn’t be a blocking action unless necessary. It's all about keeping stuff running.