Context

You can find all the code for this chapter here

Software often kicks off long-running, resource-intensive processes (often in goroutines). If the action that caused this gets cancelled or fails for some reason you need to stop these processes in a consistent way through your application.

If you don't manage this your snappy Go application that you're so proud of could start having difficult to debug performance problems.

In this chapter we'll use the package `context` to help us manage long-running processes.

We're going to start with a classic example of a web server that when hit kicks off a potentially long-running process to fetch some data for it to return in the response.

We will exercise a scenario where a user cancels the request before the data can be retrieved and we'll make sure the process is told to give up.

I've set up some code on the happy path to get us started. Here is our server code.

```
func Server(store Store) http.HandlerFunc {
        return func(w http.ResponseWriter, r *http.Request) {
                fmt.Fprint(w, store.Fetch())
        }
}
```

The function `Server` takes a `Store` and returns us a `http.HandlerFunc`. Store is defined as:

```
type Store interface {
        Fetch() string
}
```

The returned function calls the `store`'s `Fetch` method to get the data and writes it to the response.

We have a corresponding stub for `Store` which we use in a test.

```
type StubStore struct {
        response string
}

func (s *StubStore) Fetch() string {
        return s.response
}

func TestServer(t *testing.T) {
        data := "hello, world"
        svr := Server(&StubStore{data})

        request := httptest.NewRequest(http.MethodGet, "/", nil)
        response := httptest.NewRecorder()

        svr.ServeHTTP(response, request)

        if response.Body.String() != data {
                t.Errorf(`got "%s", want "%s"`, response.Body.String(), data)
        }
}
```

Now that we have a happy path, we want to make a more realistic scenario where the `Store` can't finish a `Fetch` before the user cancels the request.

Write the test first

Our handler will need a way of telling the `Store` to cancel the work so update the interface.

```
type Store interface {
        Fetch() string
        Cancel()
}
```

We will need to adjust our spy so it takes some time to return `data` and a way of knowing it has been told to cancel. We'll also rename it to `SpyStore` as we are now observing the way it is called. It'll have to add `Cancel` as a method to implement the `Store` interface.

```
type SpyStore struct {
        response  string
        cancelled bool
}

func (s *SpyStore) Fetch() string {
        time.Sleep(100 * time.Millisecond)
        return s.response
}

func (s *SpyStore) Cancel() {
        s.cancelled = true
}
```

Let's add a new test where we cancel the request before 100 milliseconds and check the store to see if it gets cancelled.

```
t.Run("tells store to cancel work if request is cancelled", func(t *testing.T) {
        data := "hello, world"
        store := &SpyStore{response: data}
        svr := Server(store)

        request := httptest.NewRequest(http.MethodGet, "/", nil)

```

```
        cancellingCtx, cancel := context.WithCancel(request.Context())
        time.AfterFunc(5*time.Millisecond, cancel)

        request = request.WithContext(cancellingCtx)

        response := httptest.NewRecorder()

        svr.ServeHTTP(response, request)

        if !store.cancelled {
                t.Error("store was not told to cancel")
        }
}
```

From the Go Blog: Context

> The context package provides functions to derive new Context values from existing ones. These values form a tree: when a Context is canceled, all Contexts derived from it are also canceled.

It's important that you derive your contexts so that cancellations are propagated throughout the call stack for a given request.

What we do is derive a new `cancellingCtx` from our `request` which returns us a `cancel` function. We then schedule that function to be called in 5 milliseconds by using `time.AfterFunc`. Finally we use this new context in our request by calling `request.WithContext`.

Try to run the test

The test fails as we'd expect.

```
--- FAIL: TestServer (0.00s)
    --- FAIL: TestServer/tells_store_to_cancel_work_if_request_is_cancelled (0.00s)
        context_test.go:62: store was not told to cancel
```

Write enough code to make it pass

Remember to be disciplined with TDD. Write the *minimal* amount of code to make our test pass.

```
func Server(store Store) http.HandlerFunc {
        return func(w http.ResponseWriter, r *http.Request) {
                store.Cancel()
                fmt.Fprint(w, store.Fetch())
        }
}
```

This makes this test pass but it doesn't feel good does it! We surely shouldn't be cancelling `Store` before we fetch on *every request*.

By being disciplined it highlighted a flaw in our tests, this is a good thing!

We'll need to update our happy path test to assert that it does not get cancelled.

```
t.Run("returns data from store", func(t *testing.T) {
        data := "hello, world"
        store := &SpyStore{response: data}
        svr := Server(store)

        request := httptest.NewRequest(http.MethodGet, "/", nil)
        response := httptest.NewRecorder()

        svr.ServeHTTP(response, request)

        if response.Body.String() != data {
                t.Errorf(`got "%s", want "%s"`, response.Body.String(), data)
        }

        if store.cancelled {
                t.Error("it should not have cancelled the store")
        }
})
```

Run both tests and the happy path test should now be failing and now we're forced to do a more sensible implementation.

```
func Server(store Store) http.HandlerFunc {
        return func(w http.ResponseWriter, r *http.Request) {
```

```
                ctx := r.Context()

                data := make(chan string, 1)

                go func() {
                        data <- store.Fetch()
                }()

                select {
                case d := <-data:
                        fmt.Fprint(w, d)
                case <-ctx.Done():
                        store.Cancel()
                }
        }
}
```

What have we done here?

`context` has a method `Done()` which returns a channel which gets sent a signal when the context is "done" or "cancelled". We want to listen to that signal and call `store.Cancel` if we get it but we want to ignore it if our `Store` manages to `Fetch` before it.

To manage this we run `Fetch` in a goroutine and it will write the result into a new channel `data`. We then use `select` to effectively race to the two asynchronous processes and then we either write a response or `Cancel`.

Refactor

We can refactor our test code a bit by making assertion methods on our spy

```
type SpyStore struct {
        response  string
        cancelled bool
        t         *testing.T
}

func (s *SpyStore) assertWasCancelled() {
        s.t.Helper()
```

```
        if !s.cancelled {
                s.t.Error("store was not told to cancel")
        }
}

func (s *SpyStore) assertWasNotCancelled() {
        s.t.Helper()
        if s.cancelled {
                s.t.Error("store was told to cancel")
        }
}
```

Remember to pass in the `*testing.T` when creating the spy.

```
func TestServer(t *testing.T) {
        data := "hello, world"

        t.Run("returns data from store", func(t *testing.T) {
                store := &SpyStore{response: data, t: t}
                svr := Server(store)

                request := httptest.NewRequest(http.MethodGet, "/", nil)
                response := httptest.NewRecorder()

                svr.ServeHTTP(response, request)

                if response.Body.String() != data {
                        t.Errorf(`got "%s", want "%s"`, response.Body.String(), data)
                }

                store.assertWasNotCancelled()
        })

        t.Run("tells store to cancel work if request is cancelled", func(t *testing.T
                store := &SpyStore{response: data, t: t}
                svr := Server(store)

                request := httptest.NewRequest(http.MethodGet, "/", nil)

                cancellingCtx, cancel := context.WithCancel(request.Context())
                time.AfterFunc(5*time.Millisecond, cancel)
                request = request.WithContext(cancellingCtx)

```

```
31              response := httptest.NewRecorder()

32              svr.ServeHTTP(response, request)
33
34              store.assertWasCancelled()
35      })
36  }
```

This approach is ok, but is it idiomatic?

Does it make sense for our web server to be concerned with manually cancelling `Store`? What if `Store` also happens to depend on other slow-running processes? We'll have to make sure that `Store.Cancel` correctly propagates the cancellation to all of its dependants.

One of the main points of `context` is that it is a consistent way of offering cancellation.

From the go doc

> Incoming requests to a server should create a Context, and outgoing calls to servers should accept a Context. The chain of function calls between them must propagate the Context, optionally replacing it with a derived Context created using WithCancel, WithDeadline, WithTimeout, or WithValue. When a Context is canceled, all Contexts derived from it are also canceled.

From the Go Blog: Context again:

> At Google, we require that Go programmers pass a Context parameter as the first argument to every function on the call path between incoming and outgoing requests. This allows Go code developed by many different teams to interoperate well. It provides simple control over timeouts and cancelation and ensures that critical values like security credentials transit Go programs properly.

(Pause for a moment and think of the ramifications of every function having to send in a context, and the ergonomics of that.)

Feeling a bit uneasy? Good. Let's try and follow that approach though and instead pass through the `context` to our `Store` and let it be responsible. That way it can also pass the `context` through to its dependants and they too can be responsible for stopping themselves.

Write the test first

We'll have to change our existing tests as their responsibilities are changing. The only thing our handler is responsible for now is making sure it sends a context through to the downstream `Store` and that it handles the error that will come from the `Store` when it is cancelled.

Let's update our `Store` interface to show the new responsibilities.

```
type Store interface {
        Fetch(ctx context.Context) (string, error)
}
```

Delete the code inside our handler for now

```
func Server(store Store) http.HandlerFunc {
        return func(w http.ResponseWriter, r *http.Request) {
        }
}
```

Update our `SpyStore`

```
type SpyStore struct {
        response string
        t        *testing.T
}

func (s *SpyStore) Fetch(ctx context.Context) (string, error) {
        data := make(chan string, 1)

        go func() {
                var result string
                for _, c := range s.response {
                        select {
                        case <-ctx.Done():
                                log.Println("spy store got cancelled")
                                return
                        default:
                                time.Sleep(10 * time.Millisecond)
                                result += string(c)
                        }
```

```
                }
                data <- result

        }()

        select {
        case <-ctx.Done():
                return "", ctx.Err()
        case res := <-data:
                return res, nil
        }
}
```

We have to make our spy act like a real method that works with `context`.

We are simulating a slow process where we build the result slowly by appending the string, character by character in a goroutine. When the goroutine finishes its work it writes the string to the `data` channel. The goroutine listens for the `ctx.Done` and will stop the work if a signal is sent in that channel.

Finally the code uses another `select` to wait for that goroutine to finish its work or for the cancellation to occur.

It's similar to our approach from before, we use Go's concurrency primitives to make two asynchronous processes race each other to determine what we return.

You'll take a similar approach when writing your own functions and methods that accept a `context` so make sure you understand what's going on.

Finally we can update our tests. Comment out our cancellation test so we can fix the happy path test first.

```
t.Run("returns data from store", func(t *testing.T) {
        data := "hello, world"
        store := &SpyStore{response: data, t: t}
        svr := Server(store)

        request := httptest.NewRequest(http.MethodGet, "/", nil)
        response := httptest.NewRecorder()

        svr.ServeHTTP(response, request)

        if response.Body.String() != data {
                t.Errorf(`got "%s", want "%s"`, response.Body.String(), data)
```

```
        }
})
```

Try to run the test

```
=== RUN   TestServer/returns_data_from_store
--- FAIL: TestServer (0.00s)
    --- FAIL: TestServer/returns_data_from_store (0.00s)
        context_test.go:22: got "", want "hello, world"
```

Write enough code to make it pass

```
func Server(store Store) http.HandlerFunc {
        return func(w http.ResponseWriter, r *http.Request) {
                data, _ := store.Fetch(r.Context())
                fmt.Fprint(w, data)
        }
}
```

Our happy path should be... happy. Now we can fix the other test.

Write the test first

We need to test that we do not write any kind of response on the error case. Sadly `httptest.ResponseRecorder` doesn't have a way of figuring this out so we'll have to roll our own spy to test for this.

```
type SpyResponseWriter struct {
        written bool
}
```

```
4
5  func (s *SpyResponseWriter) Header() http.Header {
6          s.written = true
7          return nil
8  }
9
10 func (s *SpyResponseWriter) Write([]byte) (int, error) {
11         s.written = true
12         return 0, errors.New("not implemented")
13 }
14
15 func (s *SpyResponseWriter) WriteHeader(statusCode int) {
16         s.written = true
17 }
```

Our `SpyResponseWriter` implements `http.ResponseWriter` so we can use it in the test.

```
1  t.Run("tells store to cancel work if request is cancelled", func(t *testing.T) {
2          store := &SpyStore{response: data, t: t}
3          svr := Server(store)
4
5          request := httptest.NewRequest(http.MethodGet, "/", nil)
6
7          cancellingCtx, cancel := context.WithCancel(request.Context())
8          time.AfterFunc(5*time.Millisecond, cancel)
9          request = request.WithContext(cancellingCtx)
10
11         response := &SpyResponseWriter{}
12
13         svr.ServeHTTP(response, request)
14
15         if response.written {
16                 t.Error("a response should not have been written")
17         }
18 })
```

Try to run the test

```
=== RUN   TestServer
=== RUN   TestServer/tells_store_to_cancel_work_if_request_is_cancelled
--- FAIL: TestServer (0.01s)
    --- FAIL: TestServer/tells_store_to_cancel_work_if_request_is_cancelled (0.01s)
        context_test.go:47: a response should not have been written
```

Write enough code to make it pass

```
func Server(store Store) http.HandlerFunc {
        return func(w http.ResponseWriter, r *http.Request) {
                data, err := store.Fetch(r.Context())

                if err != nil {
                        return // todo: log error however you like
                }

                fmt.Fprint(w, data)
        }
}
```

We can see after this that the server code has become simplified as it's no longer explicitly responsible for cancellation, it simply passes through `context` and relies on the downstream functions to respect any cancellations that may occur.

Wrapping up

What we've covered

- How to test a HTTP handler that has had the request cancelled by the client.
- How to use context to manage cancellation.
- How to write a function that accepts `context` and uses it to cancel itself by using goroutines, `select` and channels.
-

Follow Google's guidelines as to how to manage cancellation by propagating request scoped context through your call-stack
- How to roll your own spy for `http.ResponseWriter` if you need it.

What about context.Value ?

Michal Štrba and I have a similar opinion.

> If you use ctx.Value in my (non-existent) company, you're fired

Some engineers have advocated passing values through `context` as it *feels convenient*.

Convenience is often the cause of bad code.

The problem with `context.Values` is that it's just an untyped map so you have no type-safety and you have to handle it not actually containing your value. You have to create a coupling of map keys from one module to another and if someone changes something things start breaking.

In short, **if a function needs some values, put them as typed parameters rather than trying to fetch them from `context.Value`**. This makes it statically checked and documented for everyone to see.

But...

On other hand, it can be helpful to include information that is orthogonal to a request in a context, such as a trace id. Potentially this information would not be needed by every function in your call-stack and would make your functional signatures very messy.

Jack Lindamood says **Context.Value should inform, not control**

> The content of context.Value is for maintainers not users. It should never be required input for documented or expected results.

Additional material

- I really enjoyed reading Context should go away for Go 2 by Michal Štrba. His argument is that having to pass `context` everywhere is a smell, that it's pointing to a deficiency in the language in respect to cancellation. He says it would better if this was somehow solved at the language level, rather than at a library level. Until that happens, you will need `context` if you want to manage long running processes.
-

- The Go blog further describes the motivation for working with `context` and has some examples