Workshop: Building Concurrent Web applications

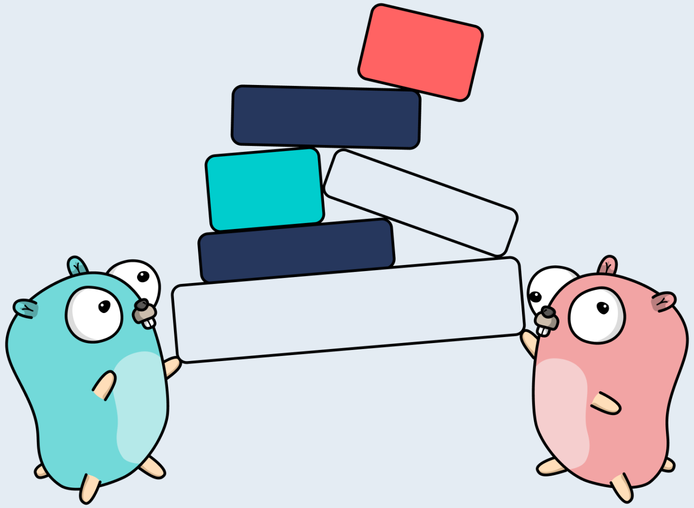
GopherCon 2022

Let’s explore concurrency with our ”Digital Ice Cream Van” web application! 🍦

We're Gophers from Form3!

Adelina Simion
Technology Evangelist
classic_addetz

Joseph Woodward
Senior Software Engineer
_josephwoodward



FORM3

Workshop outline

Introductions & setup

Concurrency basics

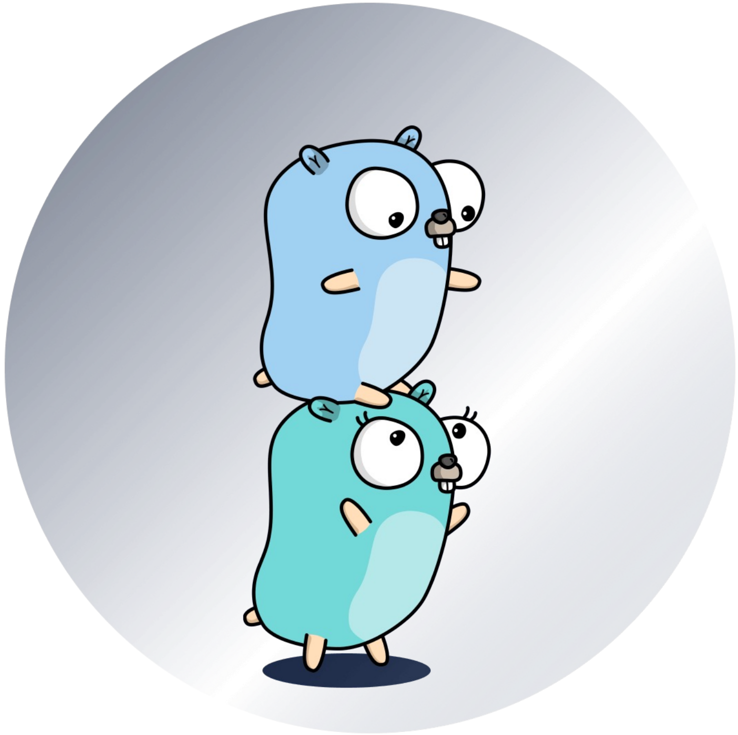
Asynchronous request handling

Shutting down

Wrap up

Workshop format

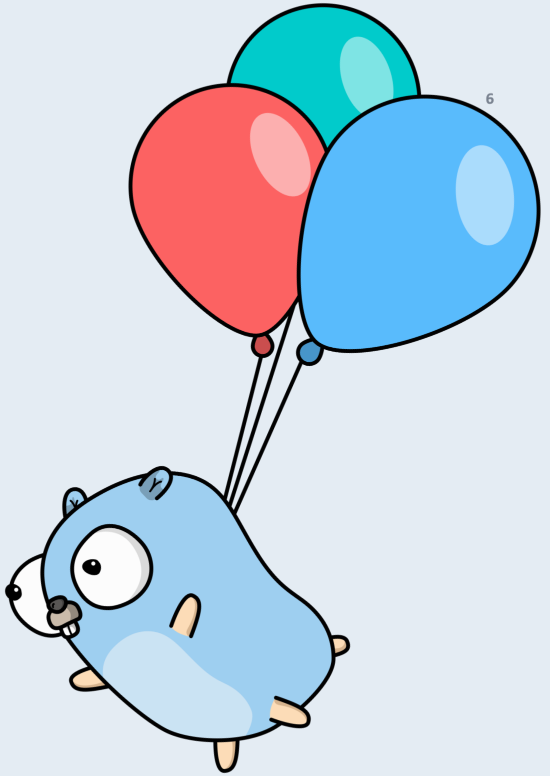
These slides are for your reference, and we will have hands on exercises throughout. We will pair and demonstrate all solutions.
Clone this repository 👉
github.com/form3tech-oss/gc22-concurrent-web-apps-workshop



Intro to the Digital Ice Cream Van

Navigating our simple web app

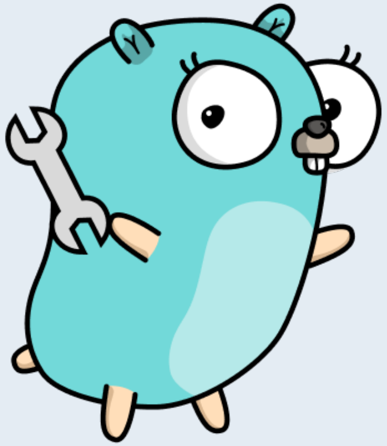
```
.
├── LICENSE
├── README.md
├── body.txt               <-- create order request
├── cmd                    <-- our executables
│   ├── load
│   │   └── main.go
│   └── server
│       ├── main.go
│       └── stock.json
├── db                     <-- our databases
│   ├── inventory.go
│   ├── order_status.go
│   └── orders.go
├── go.mod
├── go.sum
├── handlers               <-- our HTTP handlers
│   ├── config.go
│   └── handlers.go
└── request.http           <-- our pre-written requests

5 directories, 14 files
```



Endpoints

```
GET  /
POST /orders
GET  /orders/{id}
GET  /sales
```

We have a variety of ways for you to interact with these endpoints. Let's have a look at them.



```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Date: Thu, 22 Sep 2022 17:20:49 GMT
Content-Length: 158
Connection: close

{
  "message": "Welcome to the Digital Ice Cream Van!",
  "menu": [
    {
      "name": "Solero",
      "quantity": 5
    },
    {
      "name": "Magnum",
      "quantity": 5
    },
    {
      "name": "ScrewBalls",
      "quantity": 3
    }
  ]
}
```

Branches

We will be using different branches for our workshop.

Each exercise has its solution branch, if you prefer to only follow along with us. Use git checkout for each branch.

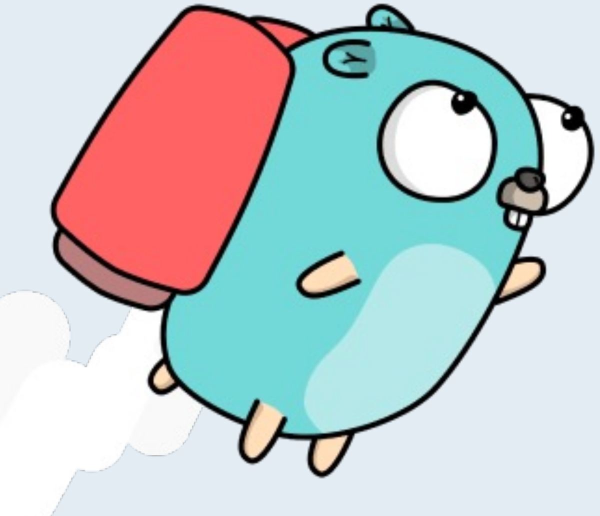
For every hour in this workshop, we will roughly chat for **40 min** and break off for **20 min**: **10 min** for your exercise and **10 min** break.

Alternatively, you can take the full 20 min break if you want to skip the exercises. We will discuss our solutions with you.

Demo Time

Let’s see the ”Digital Ice Cream Van” in action!

Hold onto your hats! 👒🎩



FORM3

Concurrency basics

Goroutines

- We start a goroutine using the `go` keyword.
- Starting a goroutine is non-blocking by design.
- The starting goroutine has a parent-child relationship to its spawned goroutines.
- All runnable applications start in the `main` goroutine.

```
package main

import "fmt"

func sayHello(name string) {
	fmt.Println("Hello from ", name)
}
func main() {
	sayHello("main")
	go sayHello("child1")
}
```

🚀 Run it 🚀

Waiting for completion - sleep

- Once the parent goroutine completes and shuts down, it terminates its children also.
- The `main` goroutine must be blocked for its child goroutines to be complete.
- A sleep statement blocks the main goroutine but wastes resources and can potentially introduce bugs.
- Never introduce sleeps to fix concurrency!

```
package main

import (
	"fmt"
	"time"
)

func sayHello(name string) {
	fmt.Println("Hello from ", name)
}
func main() {
	sayHello("main")
	go sayHello("child1")

	time.Sleep(1 * time.Second)
}
```

🚀 [Run it](#) 🚀

The sync package

- The `sync` package provides synchronization mechanisms.
- The `sync.Mutex` is a typical implementation of a lock. It should be used to wrap around **critical code sections**, which is the section of code that modifies shared state.
- The `sync.WaitGroup` is a specialized lock that waits for multiple goroutines.
- The `Mutex` and `WaitGroup` should be **passed by pointer**.

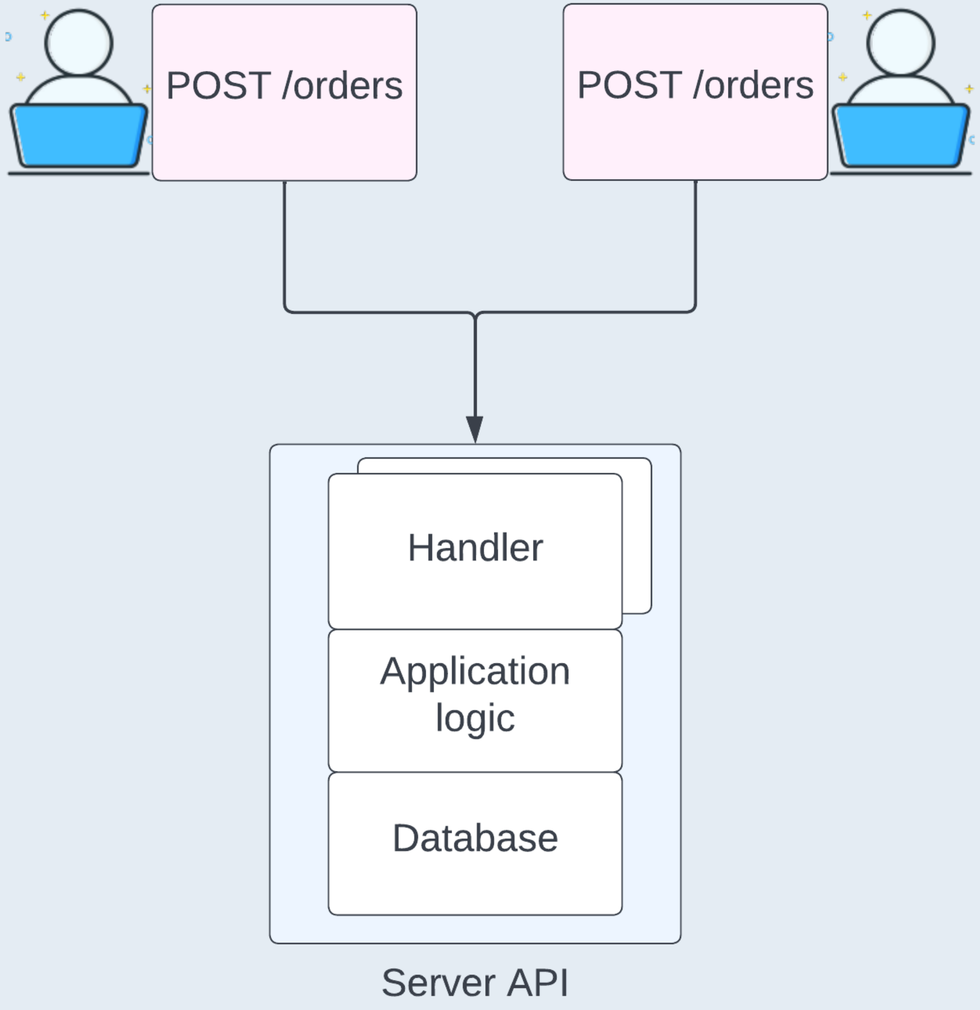
```go
package main

import (
	"fmt"
	"sync"
)

func sayHello(name string,
	wg *sync.WaitGroup) {
	defer wg.Done()
	fmt.Println("Hello from ", name)
}
func main() {
	var wg sync.WaitGroup
	wg.Add(2)

	go sayHello("child1", &wg)
	go sayHello("child2", &wg)

	wg.Wait()
}
```

🚀 Run it 🚀

FORM3

The HTTP package

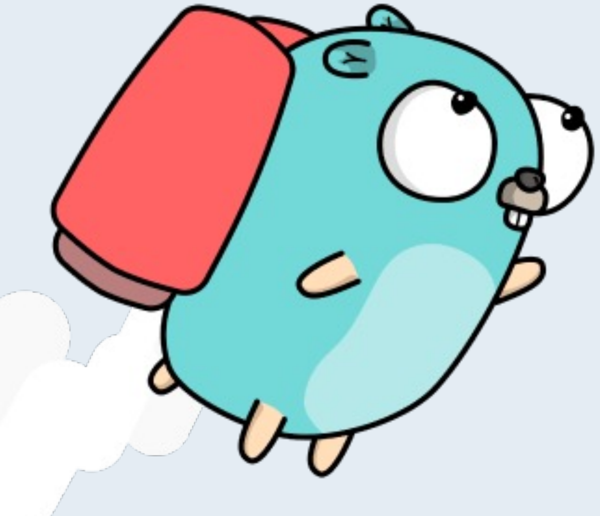
- The `net/http` package uses concurrency to be able to provide production grade performance.
- Handlers are run separately in a goroutine for each request.
- While this makes Go's `HTTP` package one of the language's strengths, this can cause issues further down the stack if we don't implement it with concurrency in mind.



Load Testing

Let's see the "Digital Ice Cream Van" under some load!

Hold onto your hats! 👒🎩



Data races

- A condition of a program where its behavior depends on relative timing or interleaving of multiple goroutines

- Can lead to inconsistent/undesirable results or hard to detect bugs.🐞

- Often occurs in **check-then-act** or **read-then-write** operations. The decision of whether to perform an operation is based on potentially stale data at the time of check/read.

```
package main

import (
	"fmt"
	"sync"
	"time"
)

var b int = 5

func sub(id int, wg *sync.WaitGroup) {
	fmt.Printf("G[%d]: %d\n", id, b)
	defer wg.Done()
	if b > 0 {
		time.Sleep(10 *
time.Millisecond)
		b = b - 1
	}
}
func main() {
	var wg sync.WaitGroup
	wg.Add(10)
	for i := 0; i < 10; i++ {
		go sub(i, &wg)
	}
	wg.Wait()
	fmt.Println("Final: ", b)
}
```

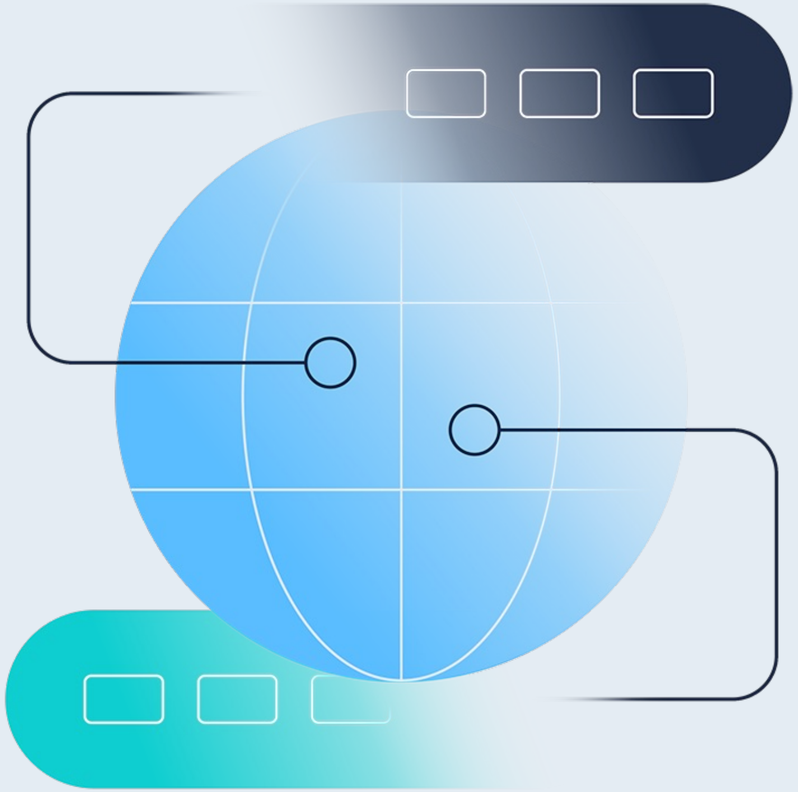
🚀 Run it 🚀

The Go race detector

- The Go race detector helps us find data races, which can be difficult to find otherwise.
- We can turn it on by adding the `-race` flag to the `go` command.
- The race detector will print out a report when it finds a data race.
- You will be able to profile your application for data races under load.
- In general, engineers profile their test environments, not production.

```
$ go run -race cmd/server/main.go
$ go run cmd/load.main.go
```

We can use the `sync` package to ensure that only one goroutine at a time can modify and read from shared resources. The Go race detector will guide our implementation.

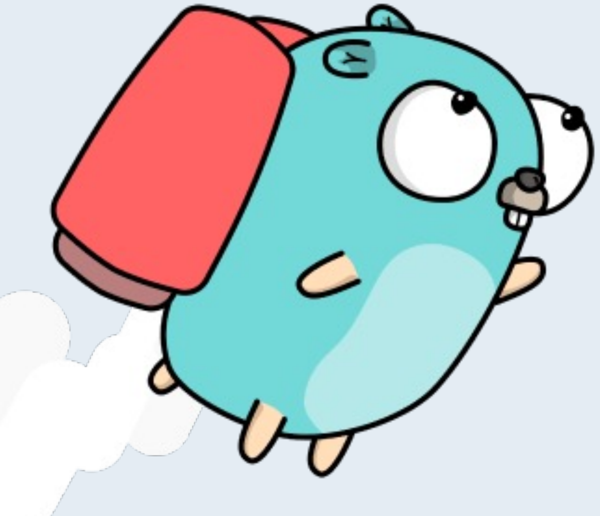


Exercise 1

Let’s fix the race condition in the inventory service!

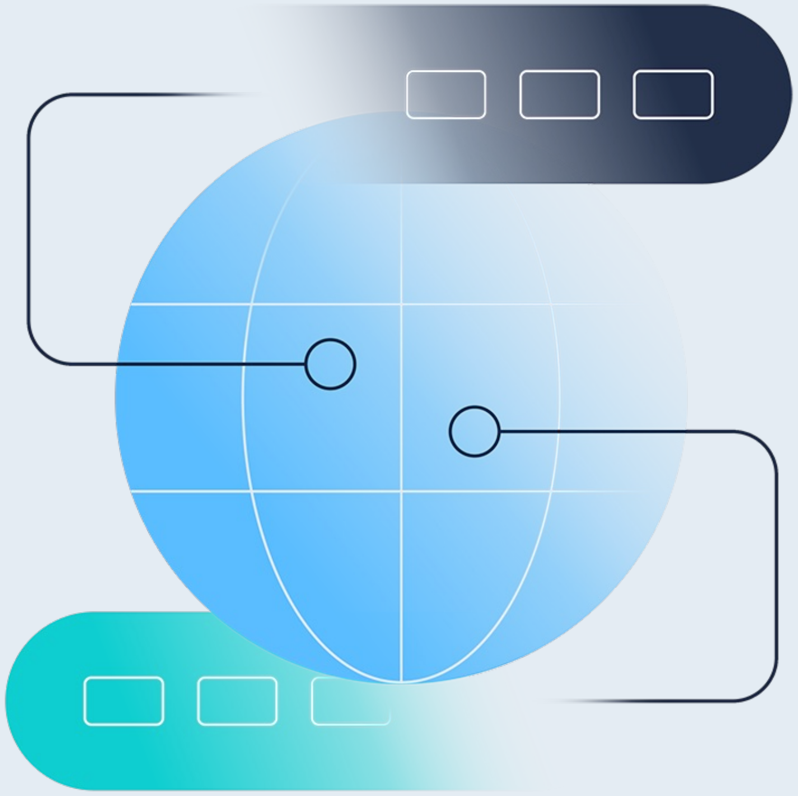
Use the race detector and load test to verify your solution.

Solution branch: **`exercise1-solution`**



FORM3

The sync.Map is an already existing concurrent map implementation. We have opted not to use it for this exercise for educational purposes.



Async request handling

Channels

- A **typed** pipe/conduit through which goroutines can send and receive information, in an **ordered** fashion.

- The channel operator is `<-`.

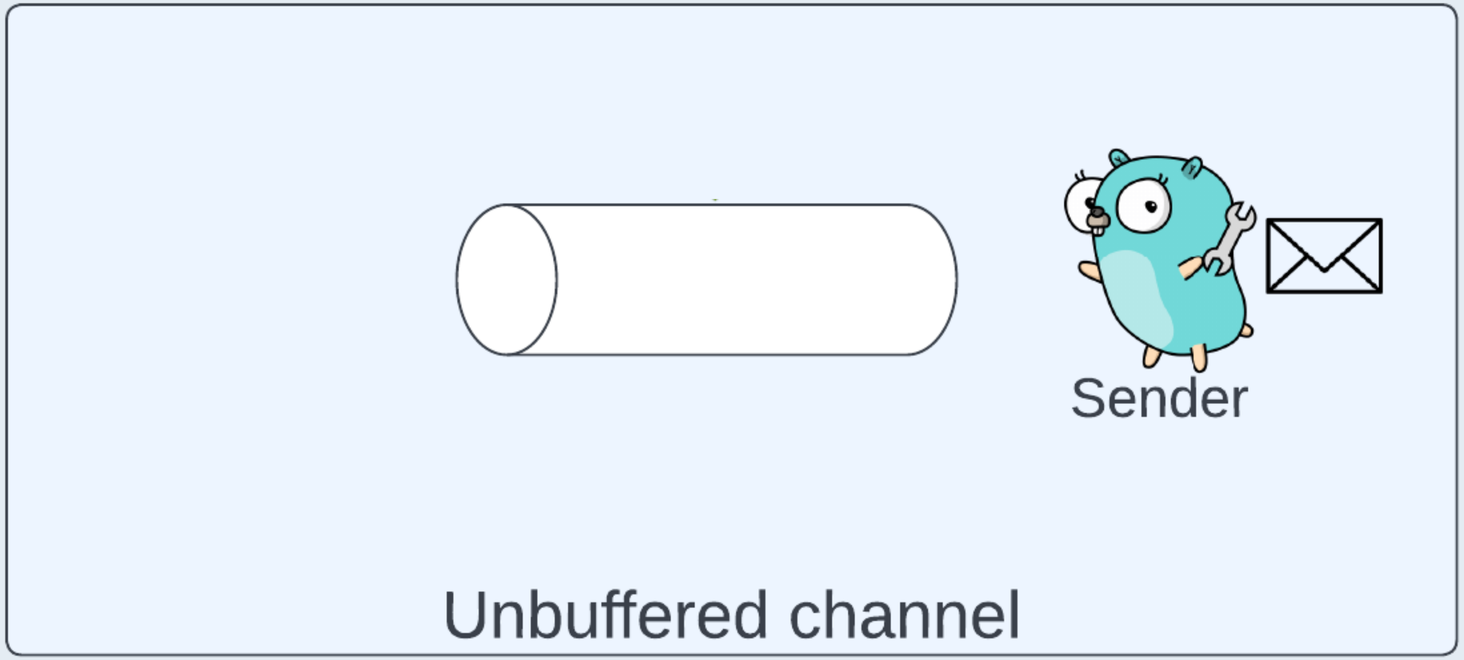
- Sending a variable `v` to channel `ch` uses the syntax `ch <- v`.

- Receiving a value from channel `ch` and assigning it to a variable `v` uses the syntax `v := <-ch`

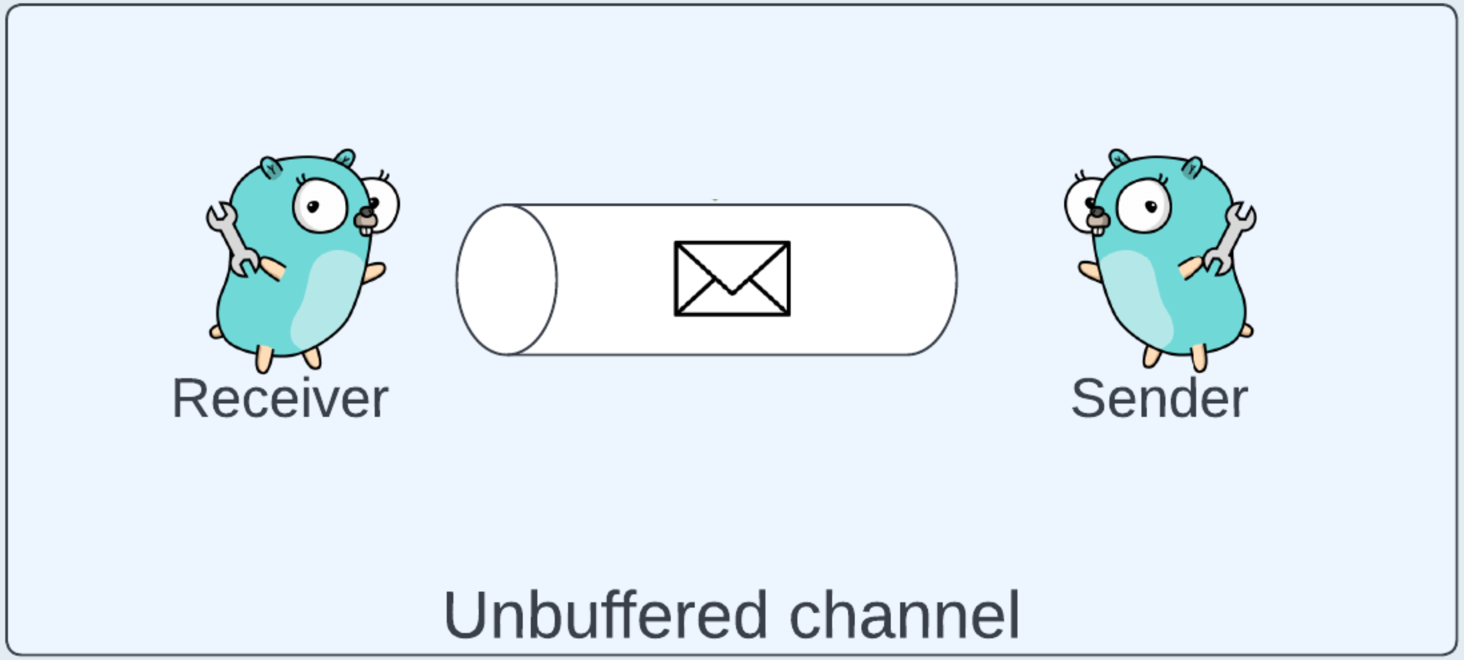
```go
package main

import "fmt"

func sayHello(name string,
        done chan string) {
        fmt.Println("Hello from ", name)
        done <- name
}
func main() {
        ch := make(chan string)

        go sayHello("child1", ch)
        go sayHello("child2", ch)

        fmt.Println(<-ch, " completed")
        fmt.Println(<-ch, " completed")
}
```

🚀 Run it 🚀

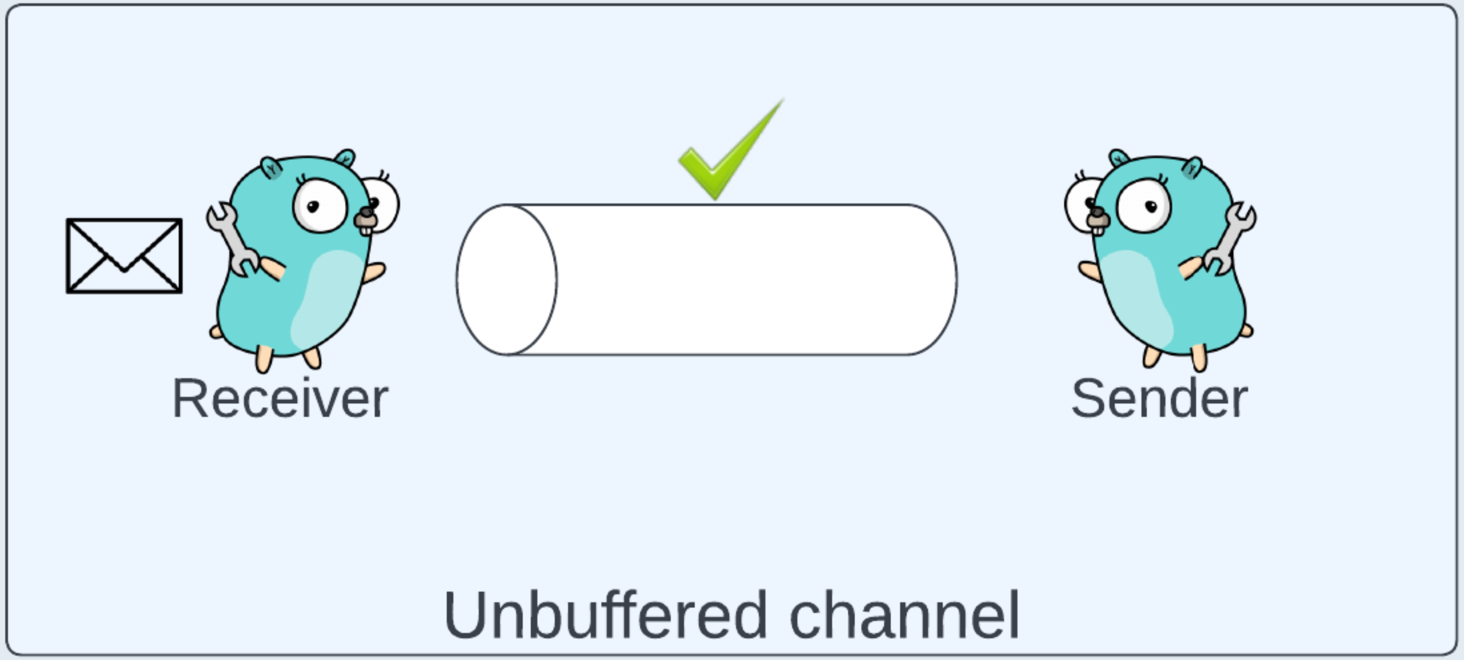
Channel visualization - unbuffered



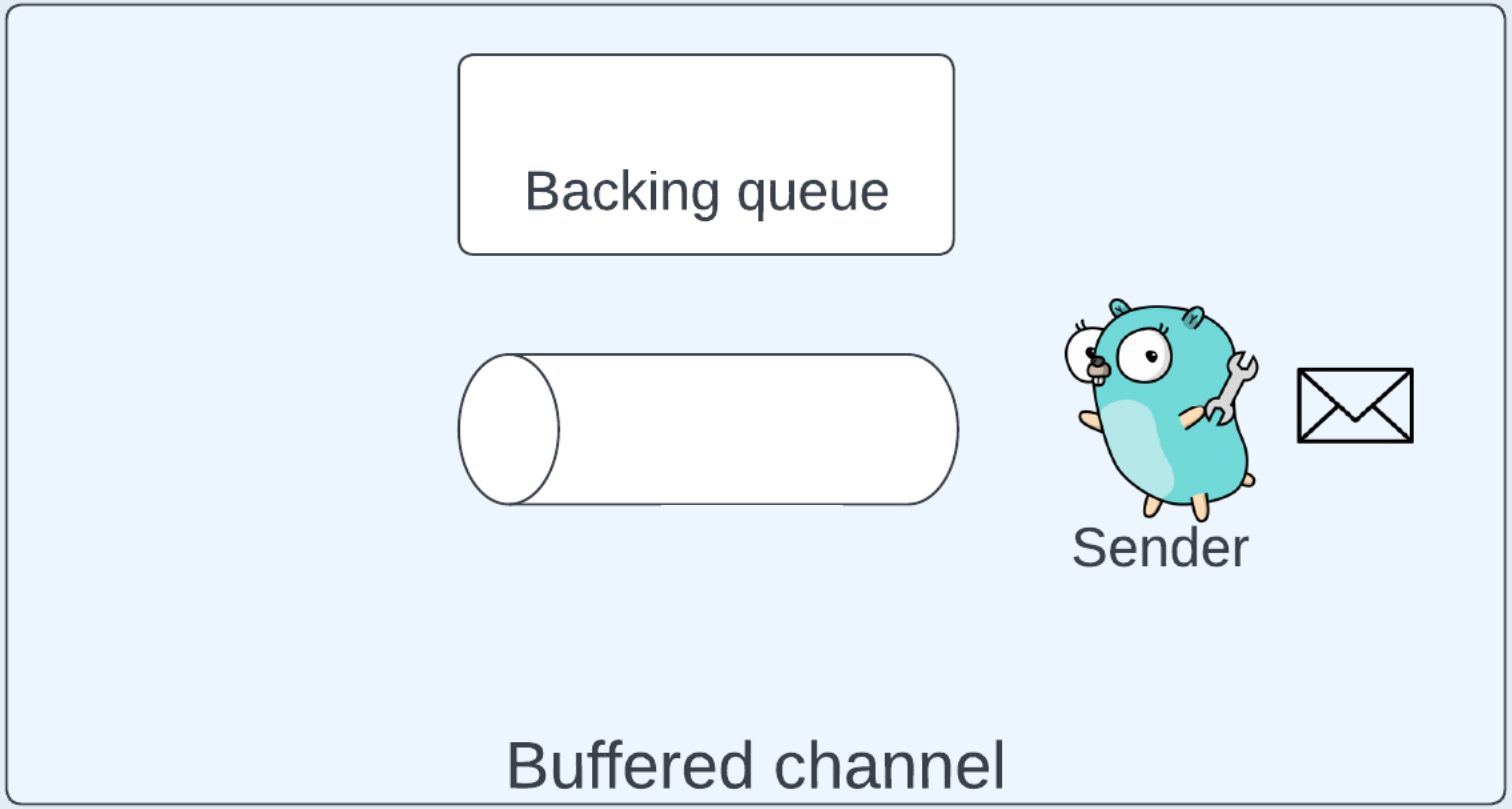
Channel visualization - unbuffered



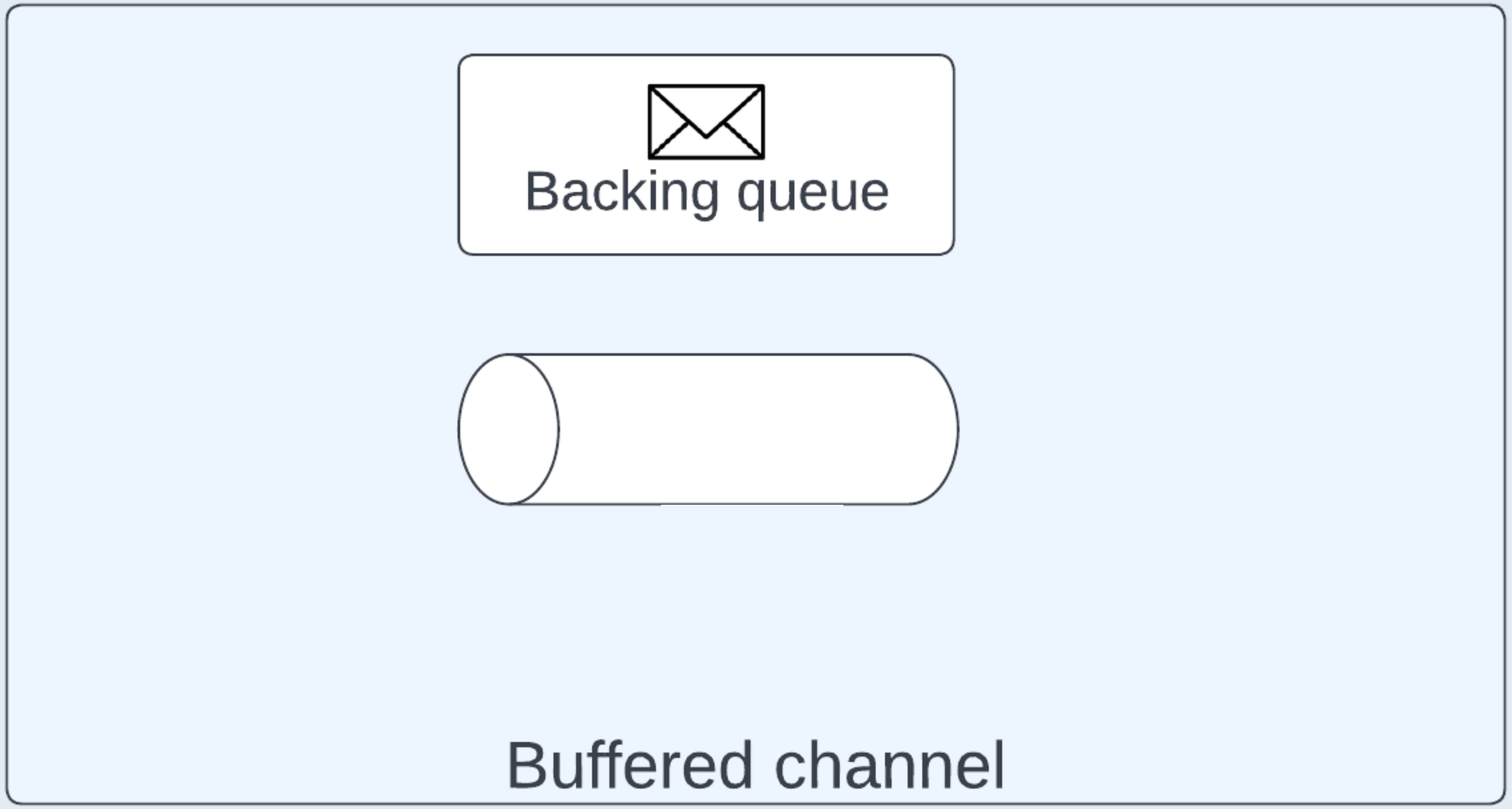
Channel visualization - unbuffered



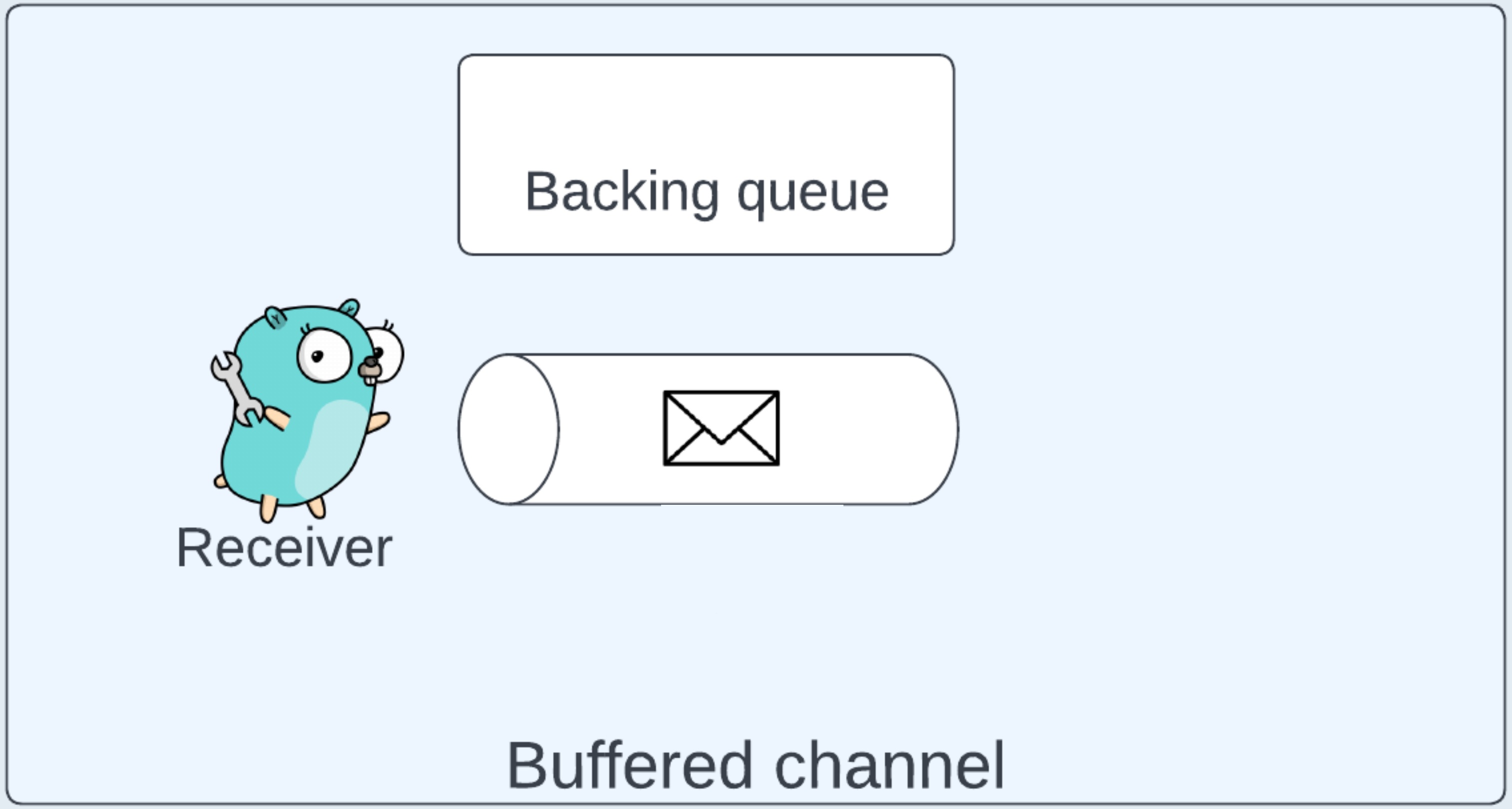
Channel visualization - buffered



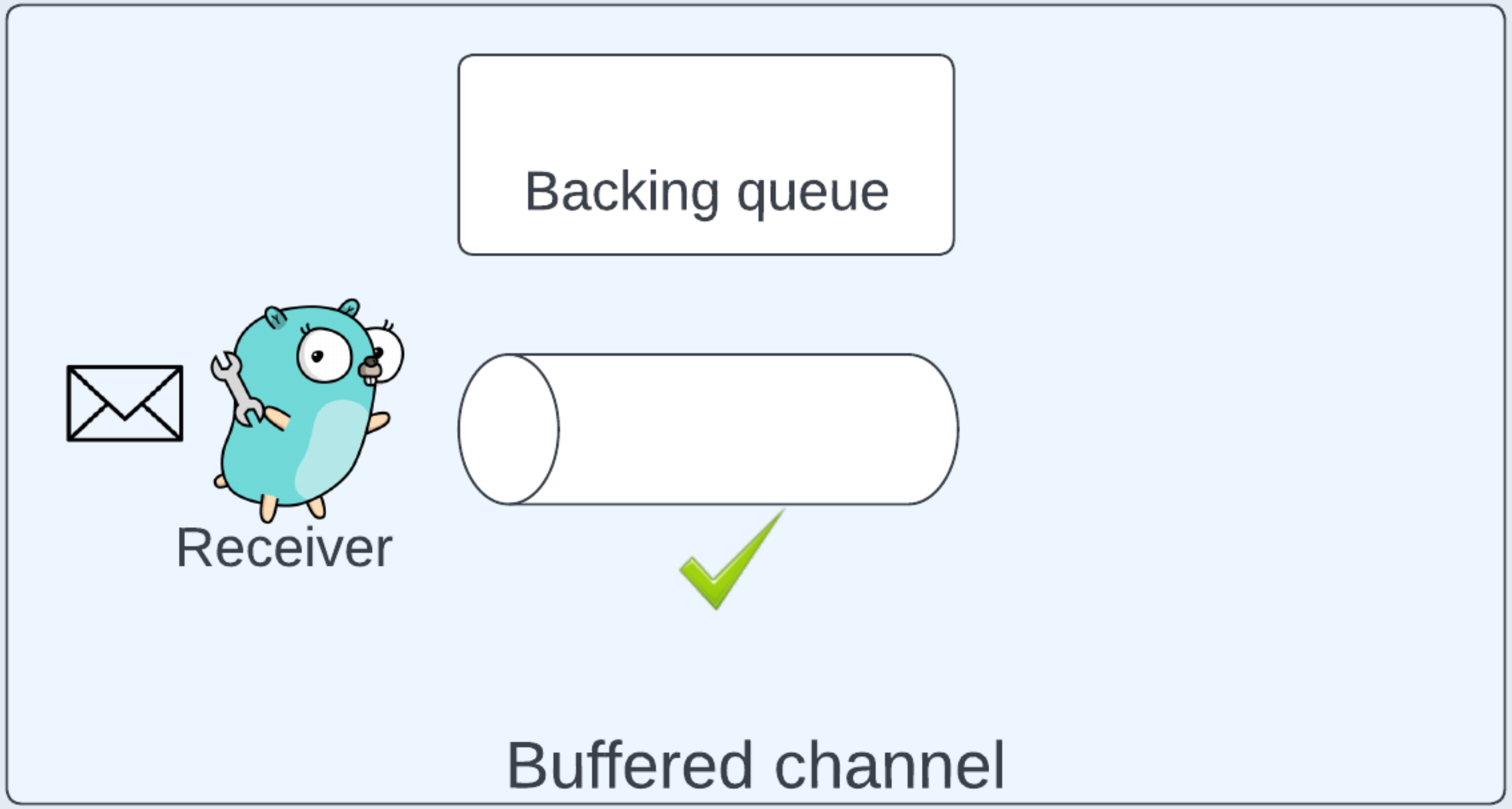
Channel visualization - buffered



Channel visualization - buffered



Channel visualization - buffered



Unbuffered vs Buffered

- By default, channels are **unbuffered** and require both sender and receiver to be available for the operation to be completed. This operation is **synchronous**.

- **Buffered** channels allow the channel to accept a limited number of values without a corresponding receiver. This allows us to create **asynchronous** operations.

```go
package main

import "fmt"

func sayHello(name string,
        ch chan string) {
        msg := fmt.Sprint("Hello from ",
              name)
        ch <- msg
}
func main() {
        ch := make(chan string, 2)

        sayHello("call1", ch)
        sayHello("call2", ch)

        fmt.Println(<-ch)
        fmt.Println(<-ch)
}
```

🚀 Run it 🚀

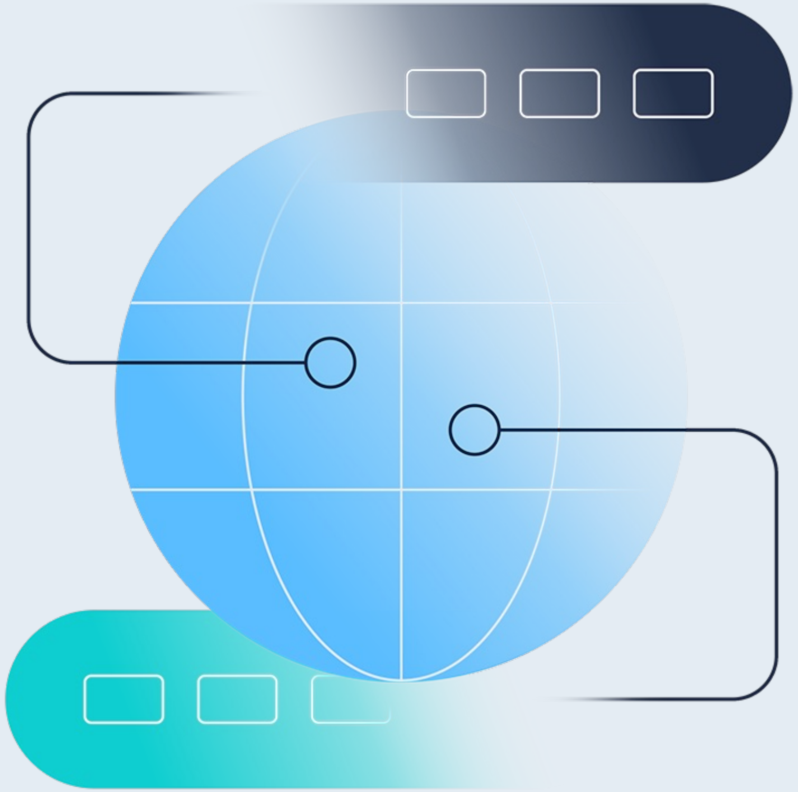
Worker pools

- A very popular concurrency pattern which allows task parallelization using goroutines.
- Often, the capacity of the buffered channel is equal to worker count.
- This pattern is particularly useful for HTTP request processing where we don't want client requests to be open while we process potentially slow requests.

```go
package main

import "fmt"

func worker(id int, in chan int) {
	for {
		input := <-in
		result := input * 2

	fmt.Printf("W[%d]:%d*2=%d\n",
			id, input, result)
	}
}
func main() {
	wcount := 3
	ch := make(chan int, wcount)
	for i := 0; i < wcount; i++ {
		go worker(i, ch)
	}
	for i := 0; i < 100; i++ {
		ch <- i
	}
}
```

🚀 Run it 🚀

The worker pool example we saw shuts down the workers together with the main goroutine. We will see how to signal completion without shutdown shortly.

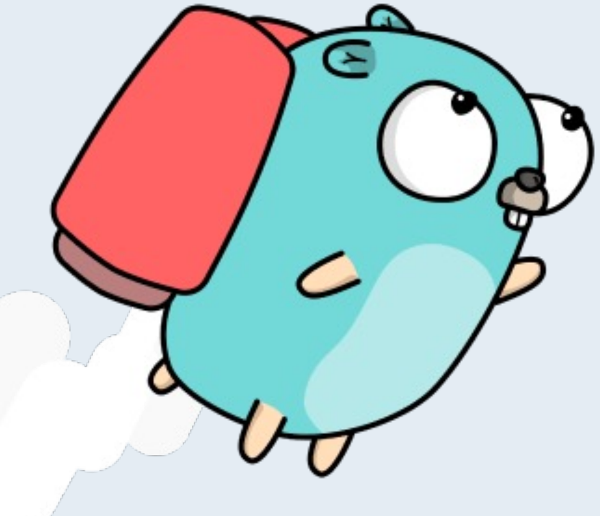


Exercise 2

Let’s make order processing asynchronous using the power of channels!

We will use 2 workers to process orders received through a channel.

Solution branch: **`exercise2-solution`**



Shutting down

Closing channels

- Channels support a third operation – the close operation.
- Closing channel `ch` uses the syntax `close(ch)`
- Signals that no more values will be sent to it.
- Receiving from a closed channel will immediately return the zero value of the channel type. The receive also return an optional `bool` value that indicates if a channel is closed.

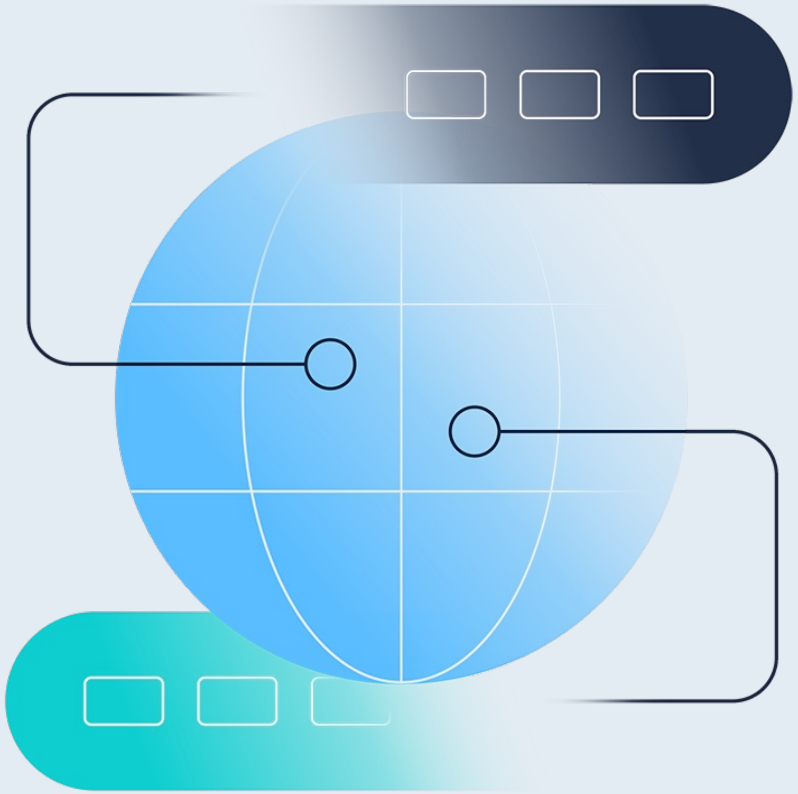
```
package main

import "fmt"

func sayHello(name string,
        done chan struct{}) {
        fmt.Println("Hello from ", name)
        close(done)
}
func main() {
        ch1 := make(chan struct{})
        ch2 := make(chan struct{})

        go sayHello("child1", ch1)
        go sayHello("child2", ch2)

        <-ch1
        <-ch2
}
```

🚀 Run it 🚀

The behavior of channels

Operation	Nil channel	Active channel	Closed channel
Sending `ch <- v`	Blocks until Active channel	Block or succeed	Panic
Receiving `v := <-ch`	Blocks until Active channel	Block or succeed	Immediately succeed with zero value
Closing `close(ch)`	Panic	Succeed	Panic

There is no way to check whether a channel is closed without interacting with it. As discussed, sends and receives are blocking operations.



Stopping work

- The `range` receive all the values of a channel and exit once it's closed.
- It is often used as a shorthand for receiving and checking whether channel is closed with the optional ok parameter.
- Closing channels can be used to signal to workers to shut down as no more values will be sent.

```
func worker(id int, in chan int) {
        name := fmt.Sprintf("W[%d]", id)
        for input := range in {
                result := input * 2
                fmt.Printf("%s:%d*2=%d\n",
                        name, input, result)
        }
}
func main() {
        wcount := 3
        ch := make(chan int, wcount)
        for i := 0; i < wcount; i++ {
                go worker(i, ch)
        }
        for i := 0; i < 100; i++ {
                ch <- i
        }
        close(ch)
        // More work in main
        time.Sleep(1 * time.Second)
}
```

🚀 Run it 🚀

Stopping work

- The `select` statement allows us to listen to multiple channels. It blocks until one of its cases can run.
- It is often used to listen to a data channel and a signal channel.
- It can be used to signal to workers to shut down without the need to shut down the main goroutine.

```
func worker(id int, out chan int,
        done chan struct{}) {
        name := fmt.Sprintf("W[%d]", id)
        n := rand.Intn(10) + 1
        for {
                select {
                case out <- n:
                        fmt.Printf("%s sent %d.\n",
                                name, n)
                case <-done:
                        fmt.Printf("%s shut down.\n",
                                name)
                        return
                }
        }
}
```

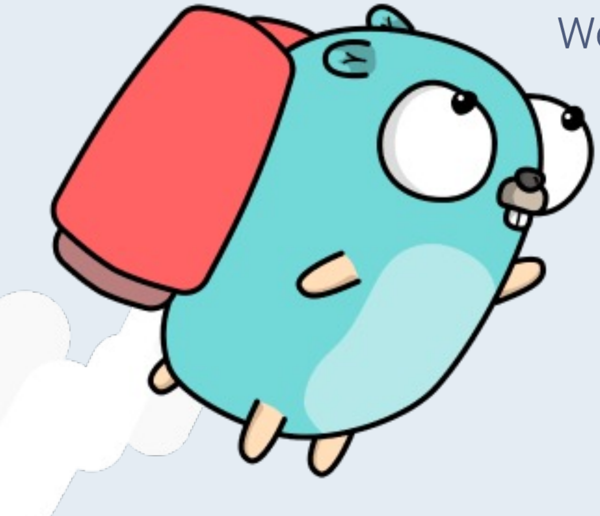
🚀 [Run it](#) 🚀

Exercise 3

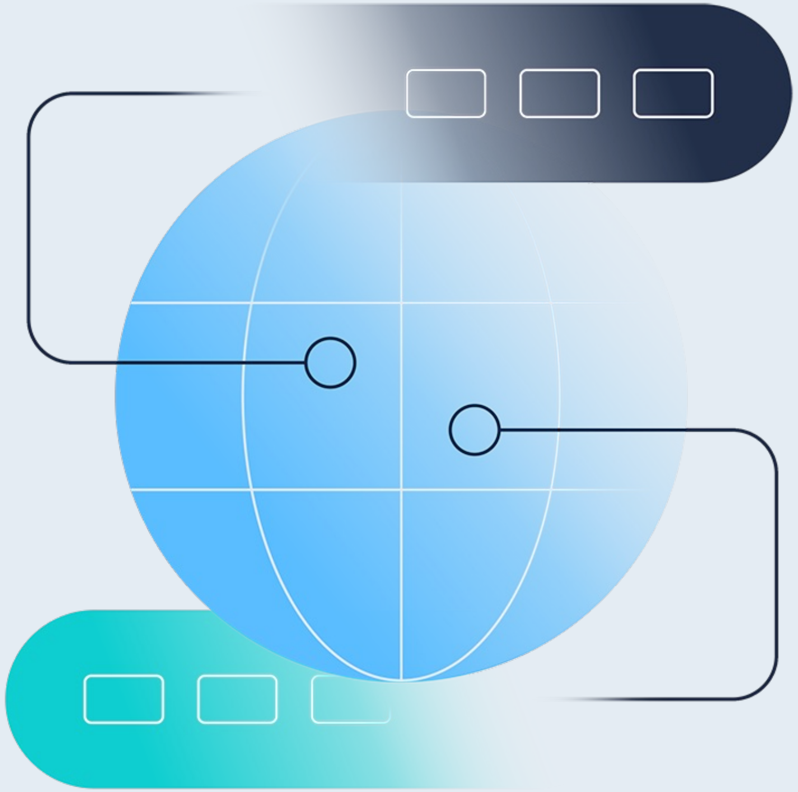
Let’s implement graceful shut down of the ice cream van!

We will implement a new `POST /close` endpoint that will stop the app from taking more orders without shutting down the server.

Solution branch: **`exercise3-solution`**



When implementing the close operation consider:
- How to handle repeated close requests
- How to handle new orders and existing orders



The errgroup

- The `errgroup` package has the type `Group`, which is a great way to handle parallel workers.
- The `Go` method starts a new goroutine and the `Wait` method waits until all goroutines are completed and then returns the first error, if any.
- Unlike `WaitGroup`, we have a guarantee that the error group will terminate in the case of a failed worker.

```
func worker(input int) error {
        if input%2 == 0 {
                return fmt.Errorf("Processing %d
failed", input)
        }
        log.Printf("%d * 2 = %d", input,
input*2)
        return nil
}
func main() {
        eg := &errgroup.Group{}
        for i := 1; i < 10; i++ {
                input := i
                eg.Go(func() error {
                        return worker(input)
                })
        }
        if err := eg.Wait(); err != nil {
                log.Println("Error:", err)
        }
}
```

🚀 Run it 🚀

Context

- The `context` type allows us to carry cancellations, timeouts and other request scoped variables values across APIs and processes.
- They are commonly used to pass timeouts and cancellations throughout the call stack.
- They are particularly useful when used together with workers and other goroutines.
- Just like goroutines, contexts carry parent-child relationships.

```
func sayHello(ctx context.Context) {
        for {
                select {
                case <-ctx.Done():
                        fmt.Println("Bye!")
                        return
                default:
                        fmt.Println("Hello!")
                        time.Sleep(1 * time.Second)
                }
        }
}
func main() {
        ctx, cancel :=
context.WithTimeout(context.Background(),
3*time.Second)
        defer cancel()
        go sayHello(ctx)
        <-ctx.Done()
        fmt.Println(ctx.Err())
        time.Sleep(2 * time.Second)
}
```

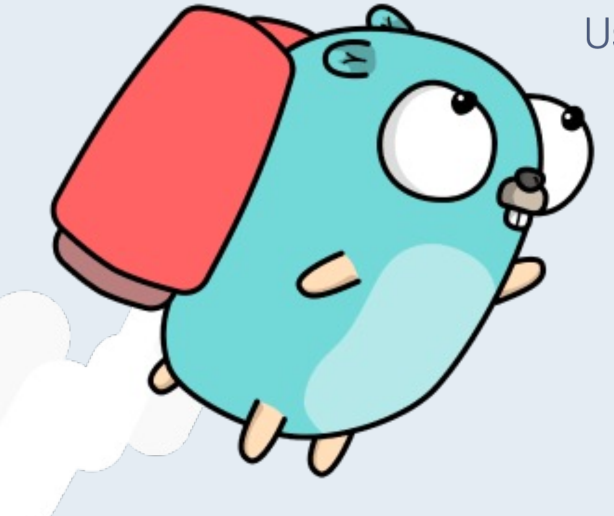
🚀 Run it 🚀

Exercise 4

Let's implement graceful shut down of the ice cream van!

Use a context to shut down the costly `GET /sales` operation if it runs for longer than 250 milliseconds.

Solution branch: `exercise4-solution`



FORM3

Wrap up

Wrap up

Concurrency allows us to scale our applications.

Think of the end conditions

The main goroutine should wait for the completion of all its workers. The workers should shut down gracefully once main signals that they should finish processing.

The race detector is your friend

Remember to use the race detector to profile and potentially find race conditions in your code.

Prefer channels over shared state

Channels are useful synchronization and information sharing mechanisms. Use them to signal shut down to workers as well as enforce processing order.

Wrap up

A word on testing...

Testing never gives us full guarantees

Due to the nature of concurrency, we cannot get full guarantees through testing that concurrent code will work. Load testing only makes us reasonably confident.

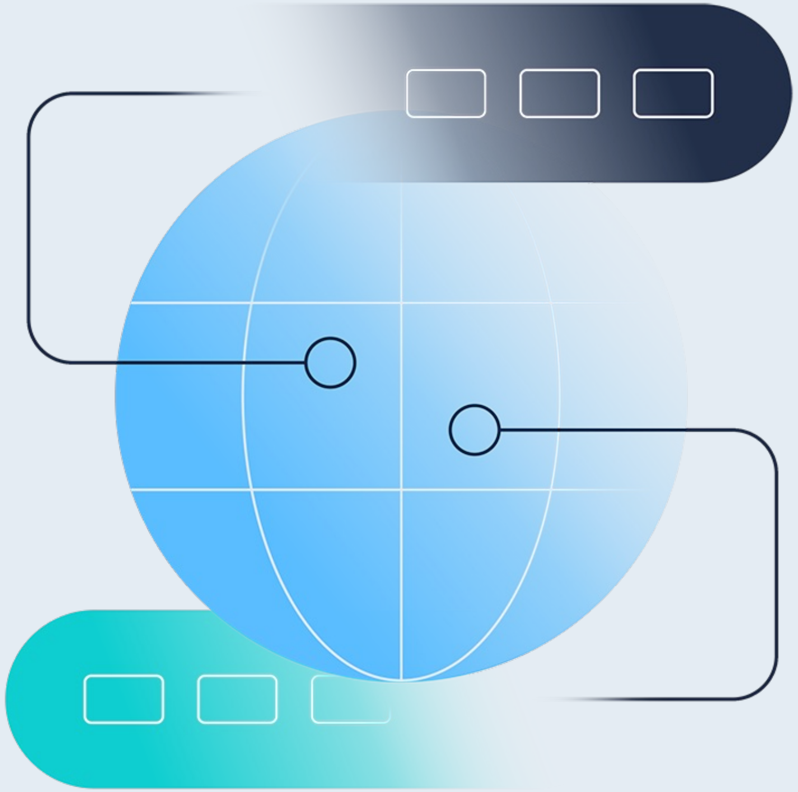
The race detector remains your BFF

We can use it in test environments to detect data races or even in production for debugging purposes. 🤩

Use load testing frameworks

Some frameworks you might consider: Grafana's k6, hey or our very own f1! Make sure to use benchmarks as well to see the behavior of your code with slow performance.

In general, data races are most often encountered, but there are many other problems that can occur such as starvation or deadlock.



Thank you!

🎙️Podcast: techpodcast.form3.tech

✍️ Blog: form3.tech/engineering

🐦@Form3Tech

FORM3