August 2023

The Three Ages of Go Concurrency

Hello, I'm Lula

A software engineer and
a full-time gopher at Monzo Bank

@LulaLeus

tinyurl.com/lulaleus

Infancy

Things I knew about concurrency when I joined Monzo

(which weren't many)

Infancy

go func (“Goroutine”)

Go

To start a new go routine is as easy as adding go before any function.

Unique to Monzo

Package **`Background`**. It provides panic recovery capabilities and helper methods to run tasks with timeout or infinitely

Infancy

net/http

Go

Each HTTP handler in Go runs concurrently as `net/http` package automatically creates a new goroutine for each incoming request.

Unique to Monzo

Typhon (open-sourced internal package) is a wrapper for Go net/http. It is used for building RPC servers and clients.

It provides additional functionality of
Middlewares
Request body Encoding/Decoding to Go structs, Context cancellation
Error propagation
And more

Infancy

WaitGroup, errgroup and concurrentgroup

Go

`sync/WaitGroup` is a simple way to span multiple Goroutines and wait for them to finish.

`x/sync/errgroup` provides capabilities of error handling and cancellation support. It uses wait group internally

Unique to Monzo

Internal library **`concurrentgroup`**. It supports panic recovery and concurrency limiting.

`concurrentgroup -> errgroup ->WaitGroup`

Infancy

Mutex

Go

A mutual exclusion lock is a mechanism enabling concurrency-safe access to shared resources by ensuring that only one goroutine can execute a specific section of code at a time.

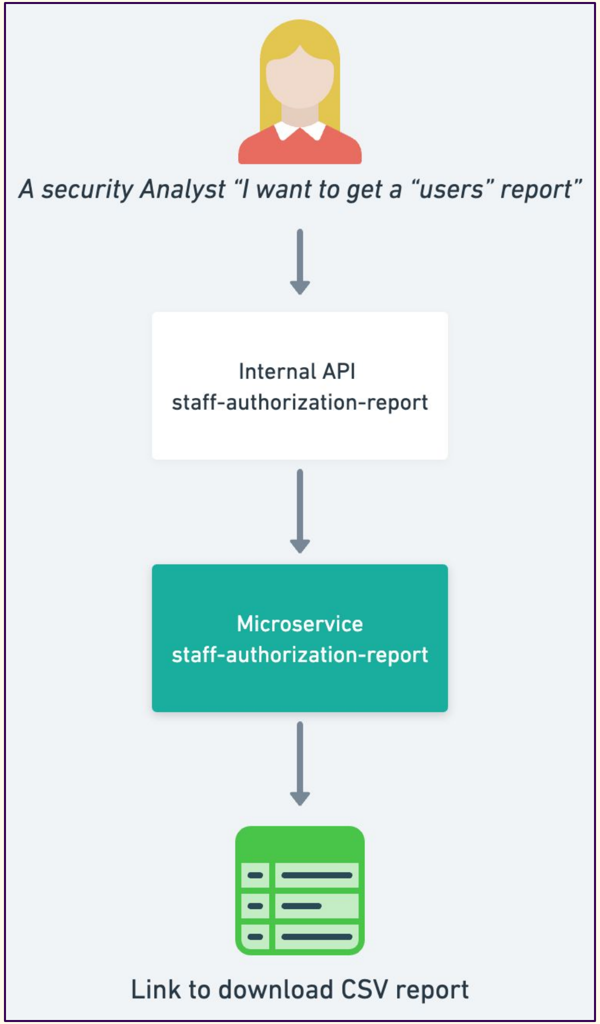
Adulthood

Writing production code and using some of the concurrency primitives

Adulthood

service.staff-authorization-report

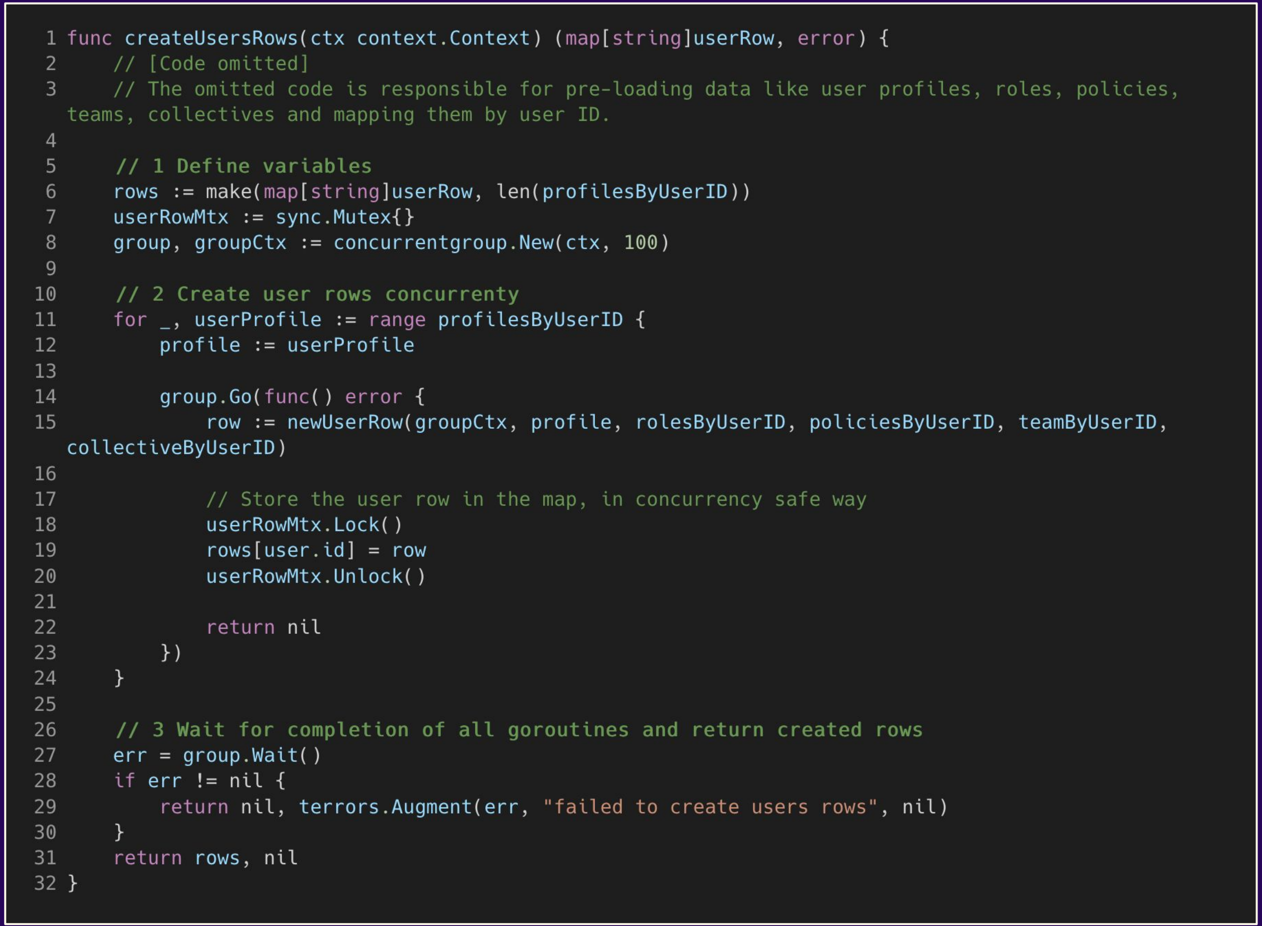
The purpose of the service is to gather data on authorizations from various services and combine it into a single CSV report.


A security Analyst "I want to get a "users" report"
Internal API
staff-authorization-report
Microservice
staff-authorization-report
Link to download CSV report

Adulthood

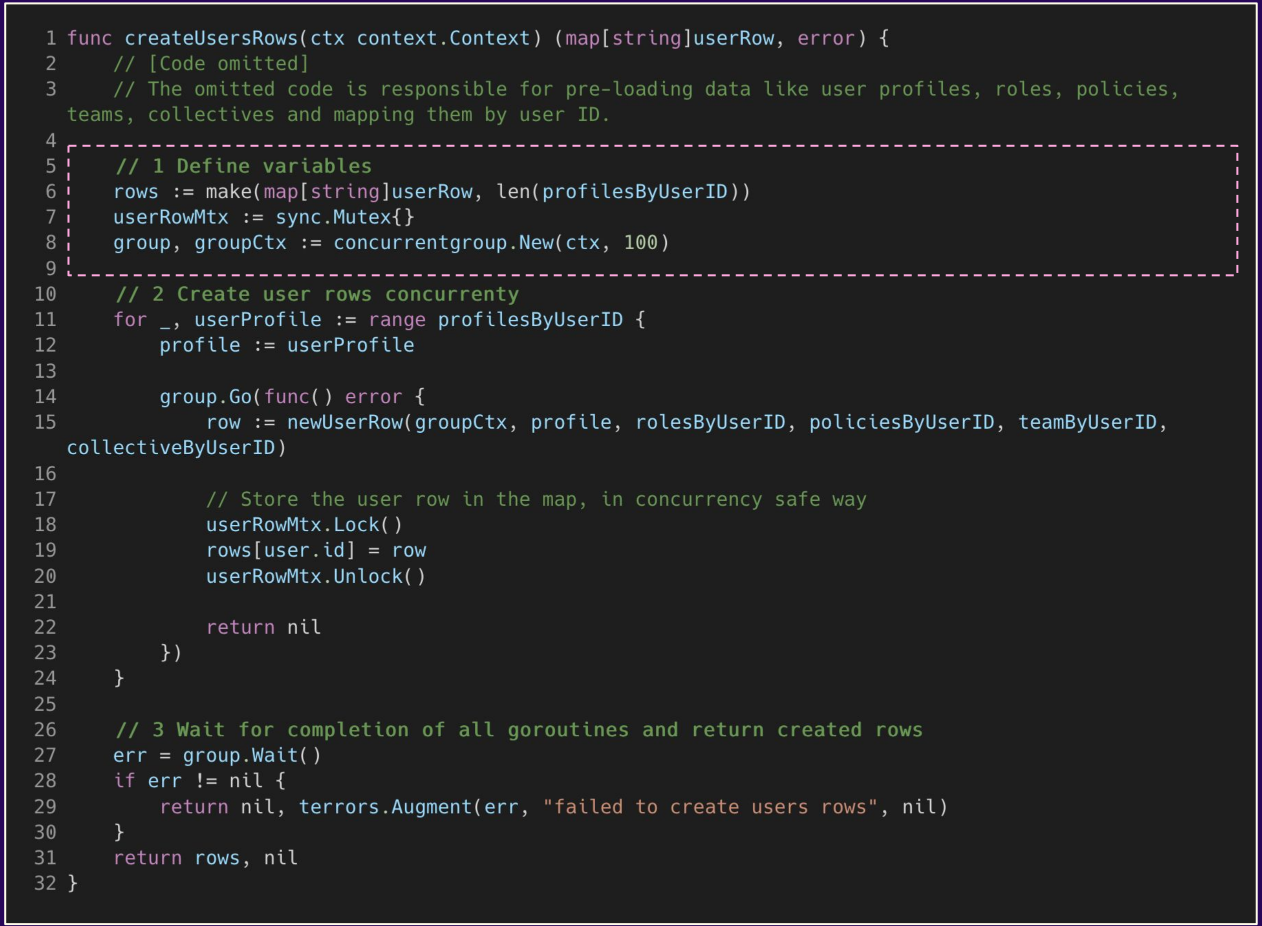
Create report rows with concurrentgroup and mutex

Function `createUsersRows` collects data from many Monzo services and creates a one report of all staff users.



```go
func createUsersRows(ctx context.Context) (map[string]userRow, error) {
    // [Code omitted]
    // The omitted code is responsible for pre-loading data like user profiles, roles, policies, teams, collectives and mapping them by user ID.

    // 1 Define variables
    rows := make(map[string]userRow, len(profilesByUserID))
    userRowMtx := sync.Mutex{}
    group, groupCtx := concurrentgroup.New(ctx, 100)

    // 2 Create user rows concurrenty
    for _, userProfile := range profilesByUserID {
        profile := userProfile

        group.Go(func() error {
            row := newUserRow(groupCtx, profile, rolesByUserID, policiesByUserID, teamByUserID, collectiveByUserID)

            // Store the user row in the map, in concurrency safe way
            userRowMtx.Lock()
            rows[user.id] = row
            userRowMtx.Unlock()

            return nil
        })
    }

    // 3 Wait for completion of all goroutines and return created rows
    err = group.Wait()
    if err != nil {
        return nil, terrors.Augment(err, "failed to create users rows", nil)
    }
    return rows, nil
}
```

Adulthood



```go
func createUsersRows(ctx context.Context) (map[string]userRow, error) {
    // [Code omitted]
    // The omitted code is responsible for pre-loading data like user profiles, roles, policies, teams, collectives and mapping them by user ID.

    // 1 Define variables
    rows := make(map[string]userRow, len(profilesByUserID))
    userRowMtx := sync.Mutex{}
    group, groupCtx := concurrentgroup.New(ctx, 100)

    // 2 Create user rows concurrenty
    for _, userProfile := range profilesByUserID {
        profile := userProfile

        group.Go(func() error {
            row := newUserRow(groupCtx, profile, rolesByUserID, policiesByUserID, teamByUserID, collectiveByUserID)

            // Store the user row in the map, in concurrency safe way
            userRowMtx.Lock()
            rows[user.id] = row
            userRowMtx.Unlock()

            return nil
        })
    }

    // 3 Wait for completion of all goroutines and return created rows
    err = group.Wait()
    if err != nil {
        return nil, terrors.Augment(err, "failed to create users rows", nil)
    }
    return rows, nil
}
```

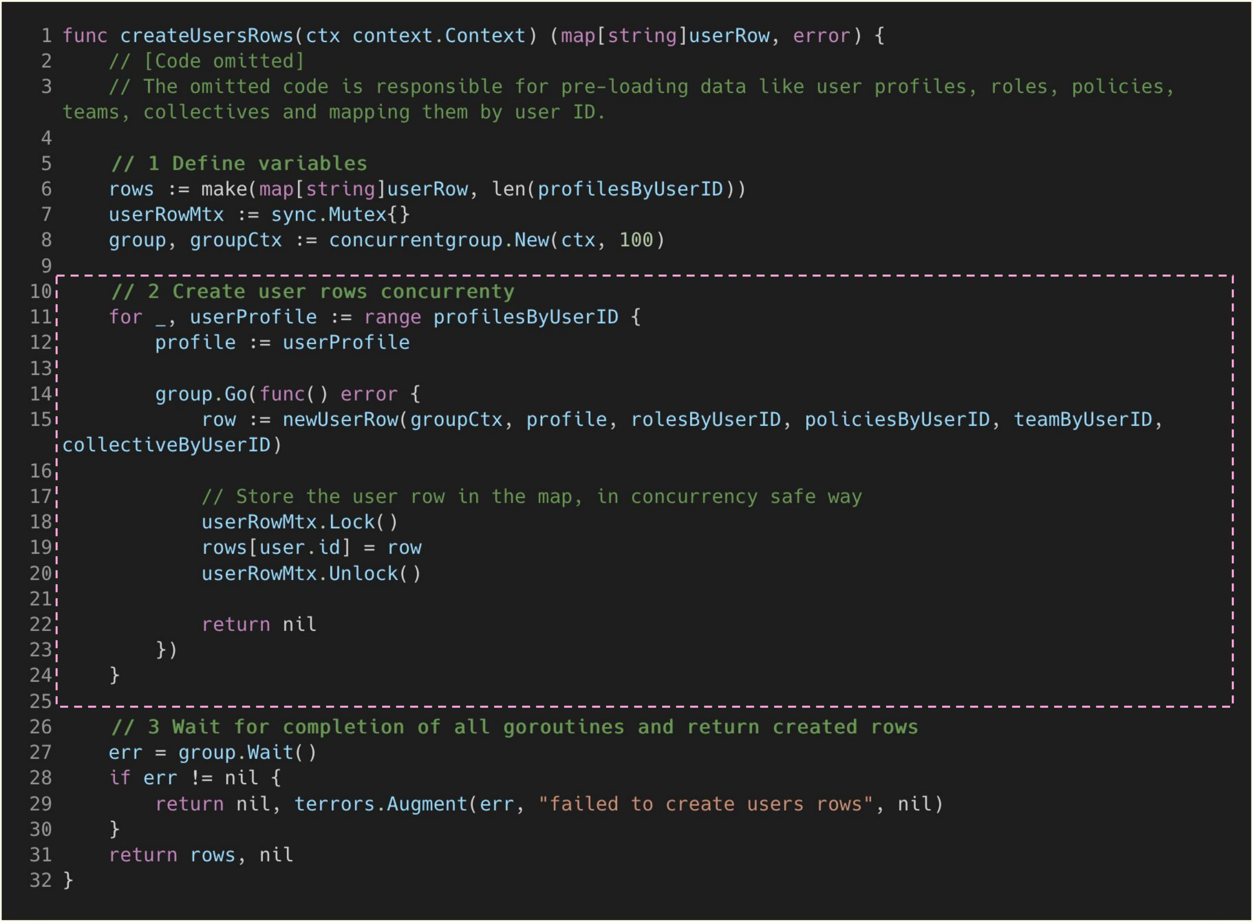
Create report rows with concurrentgroup and mutex

Function `createUsersRows` collects data from many Monzo services and creates a one report of all staff users.

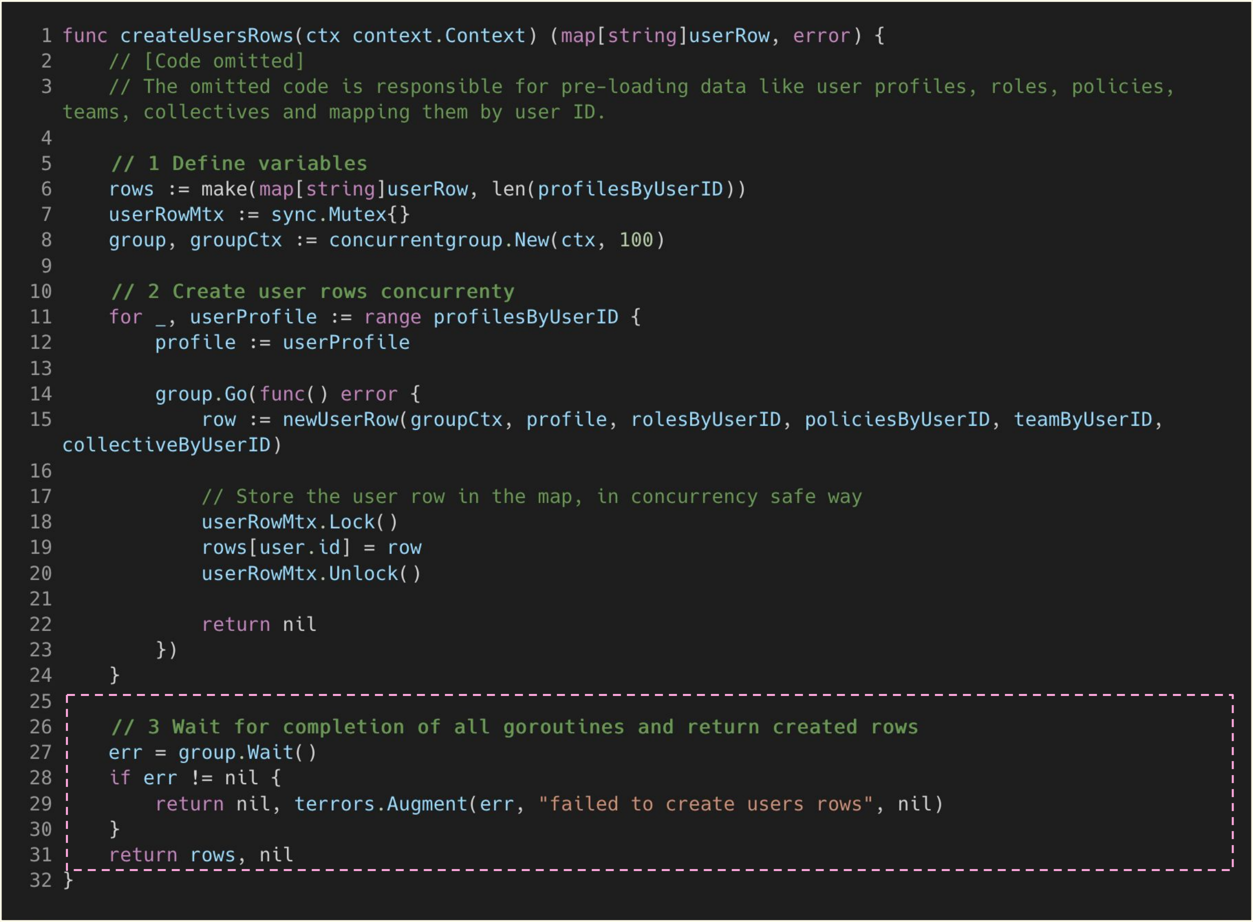
Adulthood

Create report rows with concurrentgroup and mutex

Function `createUsersRows` collects data from many Monzo services and creates a one report of all staff users.



```go
func createUsersRows(ctx context.Context) (map[string]userRow, error) {
    // [Code omitted]
    // The omitted code is responsible for pre-loading data like user profiles, roles, policies, teams, collectives and mapping them by user ID.

    // 1 Define variables
    rows := make(map[string]userRow, len(profilesByUserID))
    userRowMtx := sync.Mutex{}
    group, groupCtx := concurrentgroup.New(ctx, 100)

    // 2 Create user rows concurrenty
    for _, userProfile := range profilesByUserID {
        profile := userProfile

        group.Go(func() error {
            row := newUserRow(groupCtx, profile, rolesByUserID, policiesByUserID, teamByUserID, collectiveByUserID)

            // Store the user row in the map, in concurrency safe way
            userRowMtx.Lock()
            rows[user.id] = row
            userRowMtx.Unlock()

            return nil
        })
    }

    // 3 Wait for completion of all goroutines and return created rows
    err = group.Wait()
    if err != nil {
        return nil, terrors.Augment(err, "failed to create users rows", nil)
    }
    return rows, nil
}
```

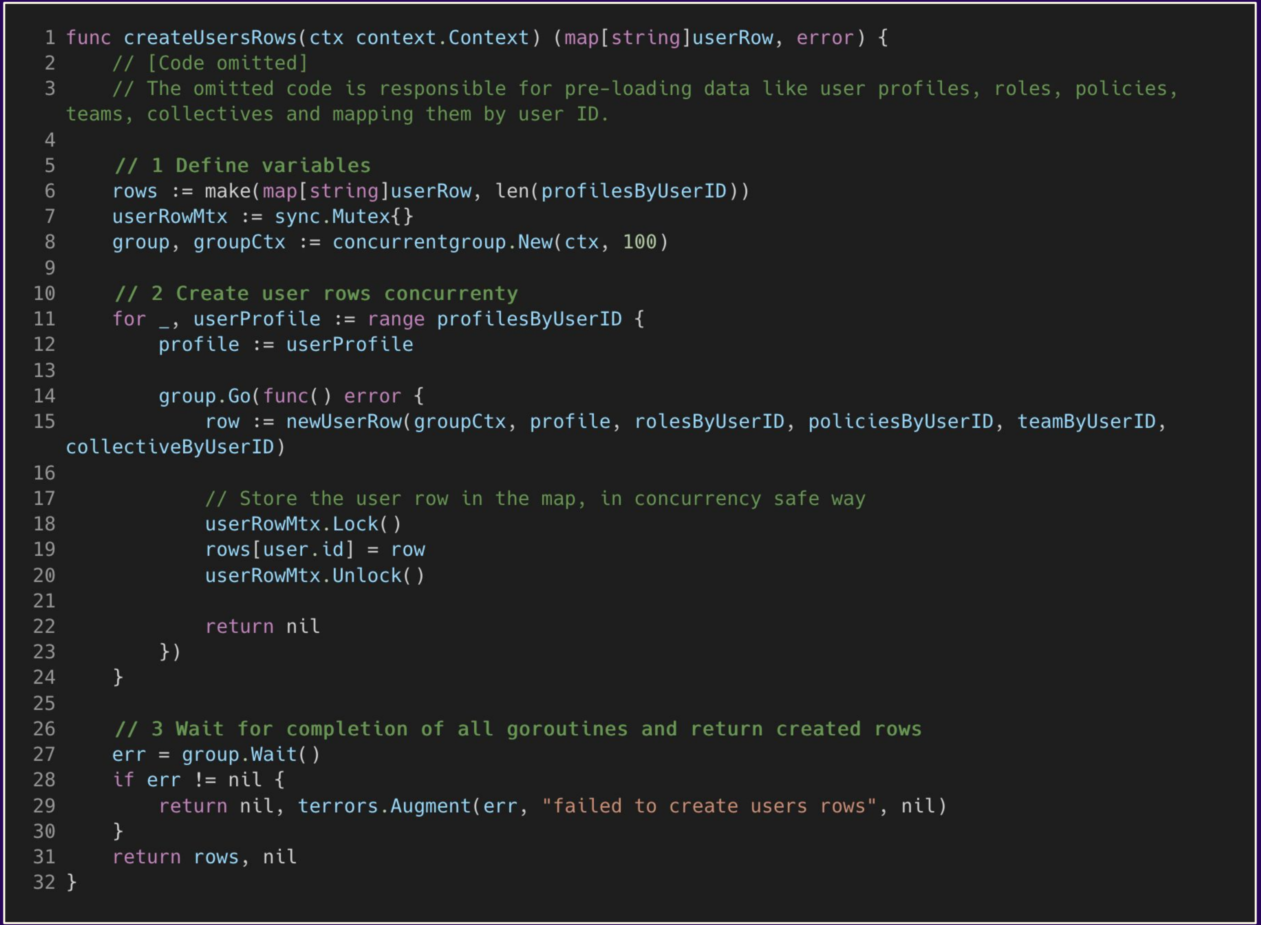


```go
func createUsersRows(ctx context.Context) (map[string]userRow, error) {
    // [Code omitted]
    // The omitted code is responsible for pre-loading data like user profiles, roles, policies, teams, collectives and mapping them by user ID.

    // 1 Define variables
    rows := make(map[string]userRow, len(profilesByUserID))
    userRowMtx := sync.Mutex{}
    group, groupCtx := concurrentgroup.New(ctx, 100)

    // 2 Create user rows concurrenty
    for _, userProfile := range profilesByUserID {
        profile := userProfile

        group.Go(func() error {
            row := newUserRow(groupCtx, profile, rolesByUserID, policiesByUserID, teamByUserID, collectiveByUserID)

            // Store the user row in the map, in concurrency safe way
            userRowMtx.Lock()
            rows[user.id] = row
            userRowMtx.Unlock()

            return nil
        })
    }

    // 3 Wait for completion of all goroutines and return created rows
    err = group.Wait()
    if err != nil {
        return nil, terrors.Augment(err, "failed to create users rows", nil)
    }
    return rows, nil
}
```

Create report rows with concurrentgroup and mutex

Function `createUsersRows` collects data from many Monzo services and creates a one report of all staff users.

Adulthood



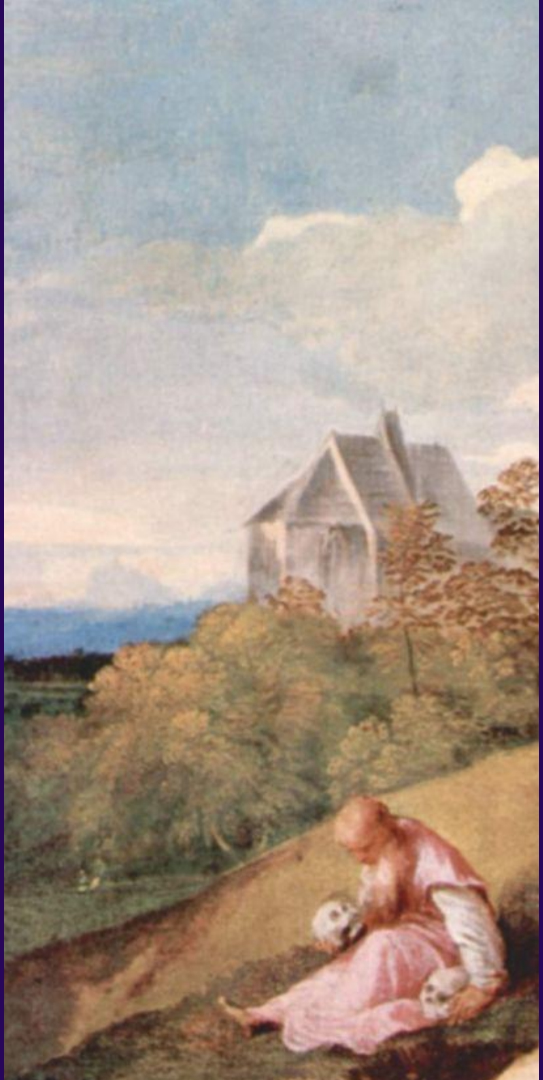
```go
func createUsersRows(ctx context.Context) (map[string]userRow, error) {
    // [Code omitted]
    // The omitted code is responsible for pre-loading data like user profiles, roles, policies, teams, collectives and mapping them by user ID.

    // 1 Define variables
    rows := make(map[string]userRow, len(profilesByUserID))
    userRowMtx := sync.Mutex{}
    group, groupCtx := concurrentgroup.New(ctx, 100)

    // 2 Create user rows concurrenty
    for _, userProfile := range profilesByUserID {
        profile := userProfile

        group.Go(func() error {
            row := newUserRow(groupCtx, profile, rolesByUserID, policiesByUserID, teamByUserID, collectiveByUserID)

            // Store the user row in the map, in concurrency safe way
            userRowMtx.Lock()
            rows[user.id] = row
            userRowMtx.Unlock()

            return nil
        })
    }

    // 3 Wait for completion of all goroutines and return created rows
    err = group.Wait()
    if err != nil {
        return nil, terrors.Augment(err, "failed to create users rows", nil)
    }
    return rows, nil
}
```

Create report rows with concurrentgroup and mutex

Function `createUsersRows` collects data from many Monzo services and creates a one report of all staff users.

Old age

Looking in bewilderment at what I’ve written in the past



Old Age

Buffered channel - a bit of theory

In Go, goroutines communicate via channels, which can be either buffered or unbuffered. Channels are unbuffered by default.

An unbuffered channel makes the sender goroutine pause after the first value is written, until that value is received.

```
// unbuffered channel
ch := make(chan int)
```

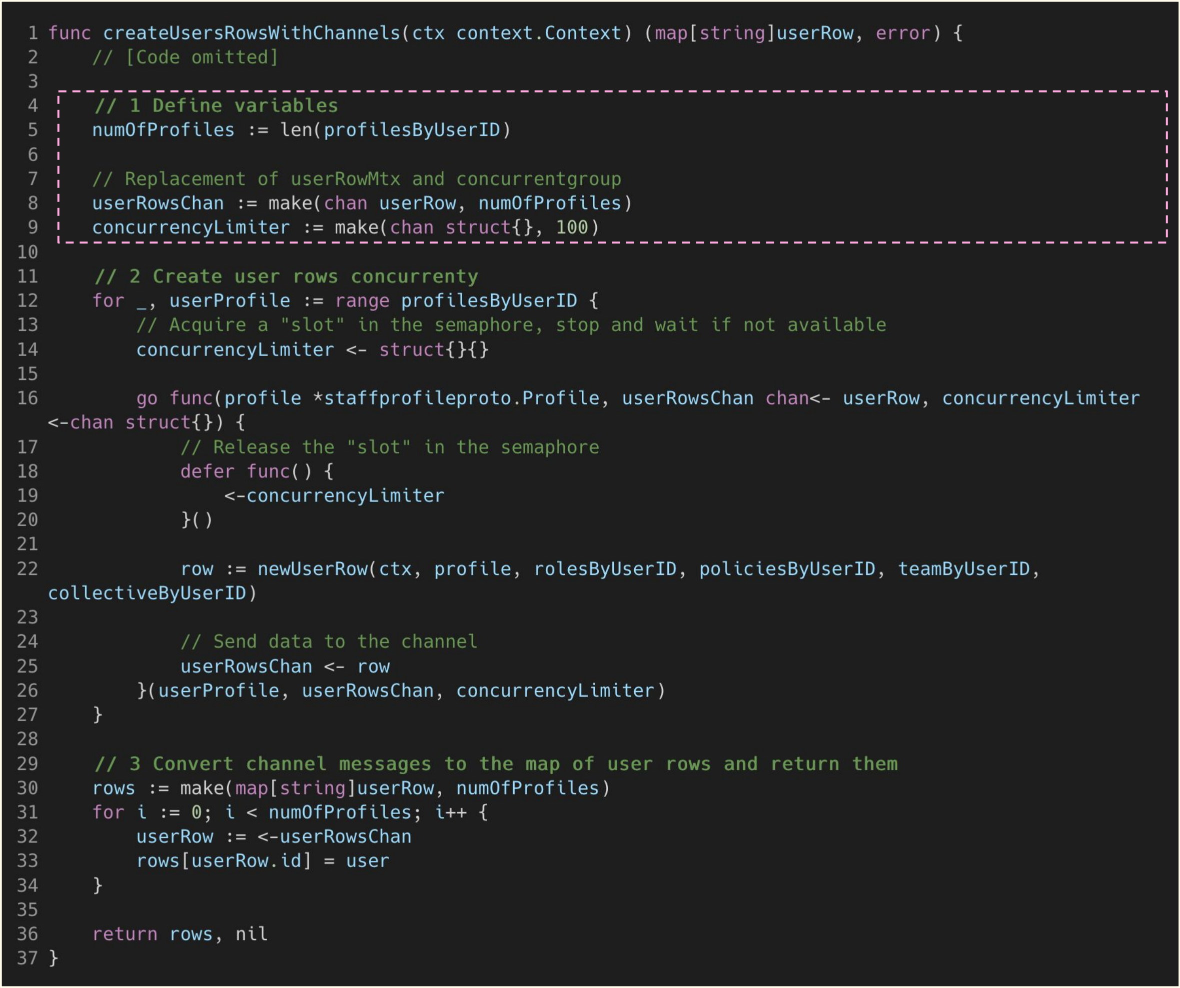
Buffered channels can hold a fixed number of values before blocking. This allows the sender to continue operations until the buffer is full, enabling a level of asynchronous processing.

```
// buffered channel with capacity 2
ch := make(chan int, 2)
```

Old Age

Create report rows with with buffered channels

The function `createUsersRows` uses two buffered channels, to generate reports rows concurrently.

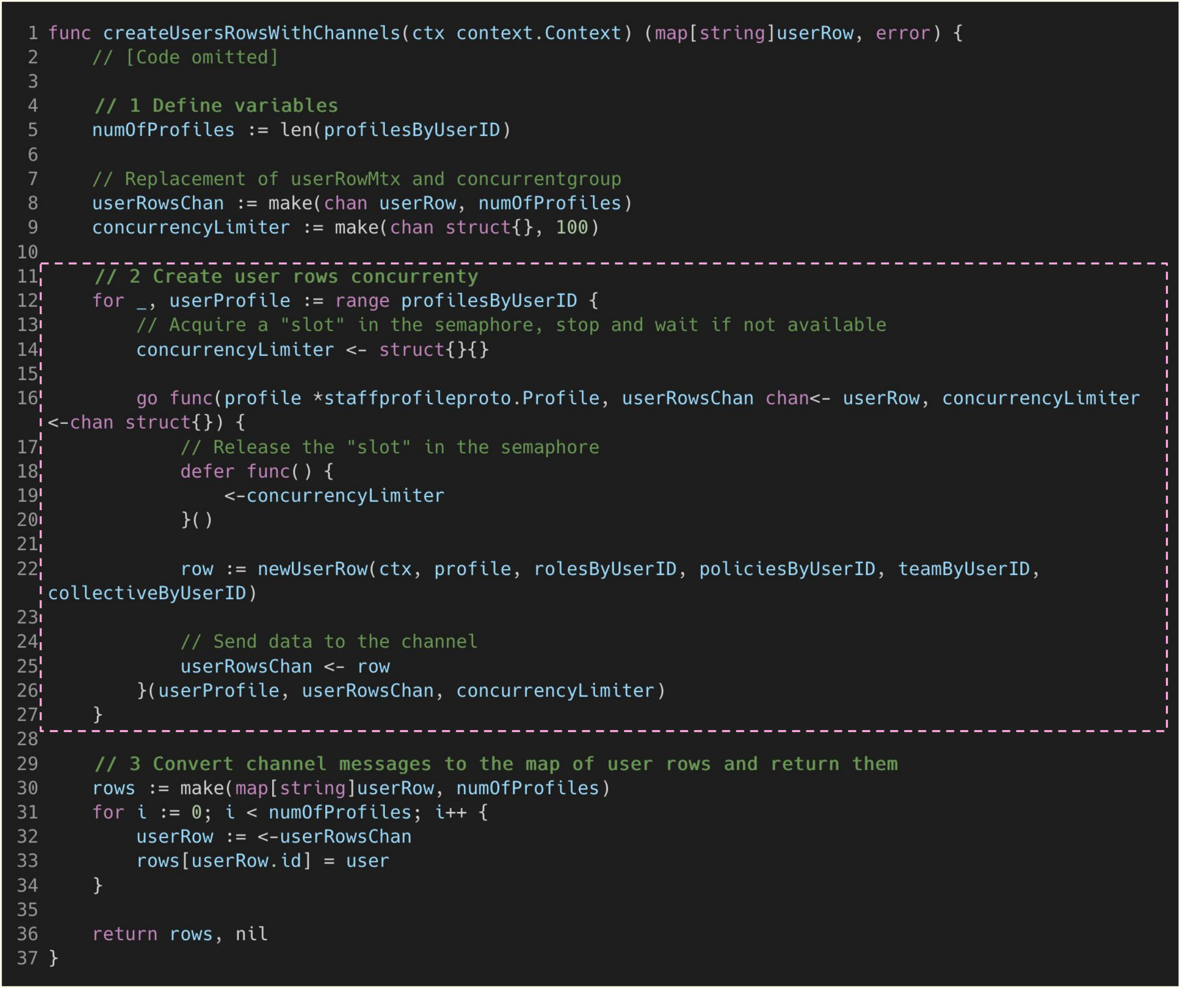


```go
func createUsersRowsWithChannels(ctx context.Context) (map[string]userRow, error) {
    // [Code omitted]

    // 1 Define variables
    numOfProfiles := len(profilesByUserID)

    // Replacement of userRowMtx and concurrentgroup
    userRowsChan := make(chan userRow, numOfProfiles)
    concurrencyLimiter := make(chan struct{}, 100)

    // 2 Create user rows concurrenty
    for _, userProfile := range profilesByUserID {
        // Acquire a "slot" in the semaphore, stop and wait if not available
        concurrencyLimiter <- struct{}{}

        go func(profile *staffprofileproto.Profile, userRowsChan chan<- userRow, concurrencyLimiter <-chan struct{}) {
            // Release the "slot" in the semaphore
            defer func() {
                <-concurrencyLimiter
            }()

            row := newUserRow(ctx, profile, rolesByUserID, policiesByUserID, teamByUserID, collectiveByUserID)

            // Send data to the channel
            userRowsChan <- row
        }(userProfile, userRowsChan, concurrencyLimiter)
    }

    // 3 Convert channel messages to the map of user rows and return them
    rows := make(map[string]userRow, numOfProfiles)
    for i := 0; i < numOfProfiles; i++ {
        userRow := <-userRowsChan
        rows[userRow.id] = user
    }

    return rows, nil
}
```

Old Age

Create report rows with with buffered channels

The function `createUsersRows` uses two buffered channels, to generate reports rows concurrently.

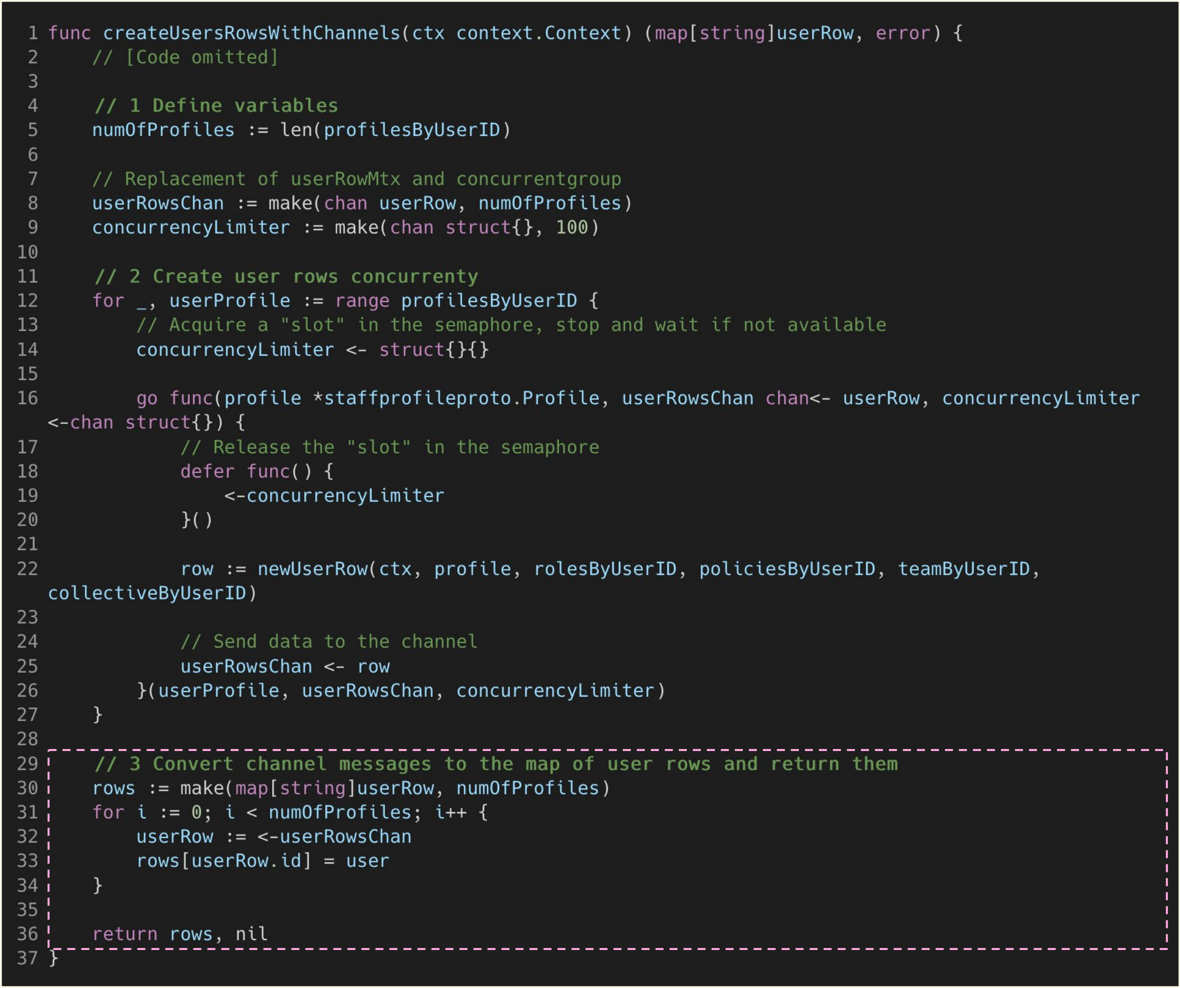


```go
func createUsersRowsWithChannels(ctx context.Context) (map[string]userRow, error) {
    // [Code omitted]

    // 1 Define variables
    numOfProfiles := len(profilesByUserID)

    // Replacement of userRowMtx and concurrentgroup
    userRowsChan := make(chan userRow, numOfProfiles)
    concurrencyLimiter := make(chan struct{}, 100)

    // 2 Create user rows concurrently
    for _, userProfile := range profilesByUserID {
        // Acquire a "slot" in the semaphore, stop and wait if not available
        concurrencyLimiter <- struct{}{}

        go func(profile *staffprofileproto.Profile, userRowsChan chan<- userRow, concurrencyLimiter <-chan struct{}) {
            // Release the "slot" in the semaphore
            defer func() {
                <-concurrencyLimiter
            }()

            row := newUserRow(ctx, profile, rolesByUserID, policiesByUserID, teamByUserID, collectiveByUserID)

            // Send data to the channel
            userRowsChan <- row
        }(userProfile, userRowsChan, concurrencyLimiter)
    }

    // 3 Convert channel messages to the map of user rows and return them
    rows := make(map[string]userRow, numOfProfiles)
    for i := 0; i < numOfProfiles; i++ {
        userRow := <-userRowsChan
        rows[userRow.id] = user
    }

    return rows, nil
}
```

Old Age

Create report rows with with buffered channels

The function `createUsersRows` uses two buffered channels, to generate reports rows concurrently.



```go
func createUsersRowsWithChannels(ctx context.Context) (map[string]userRow, error) {
    // [Code omitted]

    // 1 Define variables
    numOfProfiles := len(profilesByUserID)

    // Replacement of userRowMtx and concurrentgroup
    userRowsChan := make(chan userRow, numOfProfiles)
    concurrencyLimiter := make(chan struct{}, 100)

    // 2 Create user rows concurrenty
    for _, userProfile := range profilesByUserID {
        // Acquire a "slot" in the semaphore, stop and wait if not available
        concurrencyLimiter <- struct{}{}

        go func(profile *staffprofileproto.Profile, userRowsChan chan<- userRow, concurrencyLimiter <-chan struct{}) {
            // Release the "slot" in the semaphore
            defer func() {
                <-concurrencyLimiter
            }()

            row := newUserRow(ctx, profile, rolesByUserID, policiesByUserID, teamByUserID, collectiveByUserID)

            // Send data to the channel
            userRowsChan <- row
        }(userProfile, userRowsChan, concurrencyLimiter)
    }

    // 3 Convert channel messages to the map of user rows and return them
    rows := make(map[string]userRow, numOfProfiles)
    for i := 0; i < numOfProfiles; i++ {
        userRow := <-userRowsChan
        rows[userRow.id] = user
    }

    return rows, nil
}
```

Old Age

Semaphore - a bit of theory

A semaphore is a concept of a synchronization mechanism to control simultaneous access to shared resources.

`golang.org/x/sync/semaphore` package implements the concept.

It supports weighted tasks, which allows different tasks to consume and release different amounts of resources.

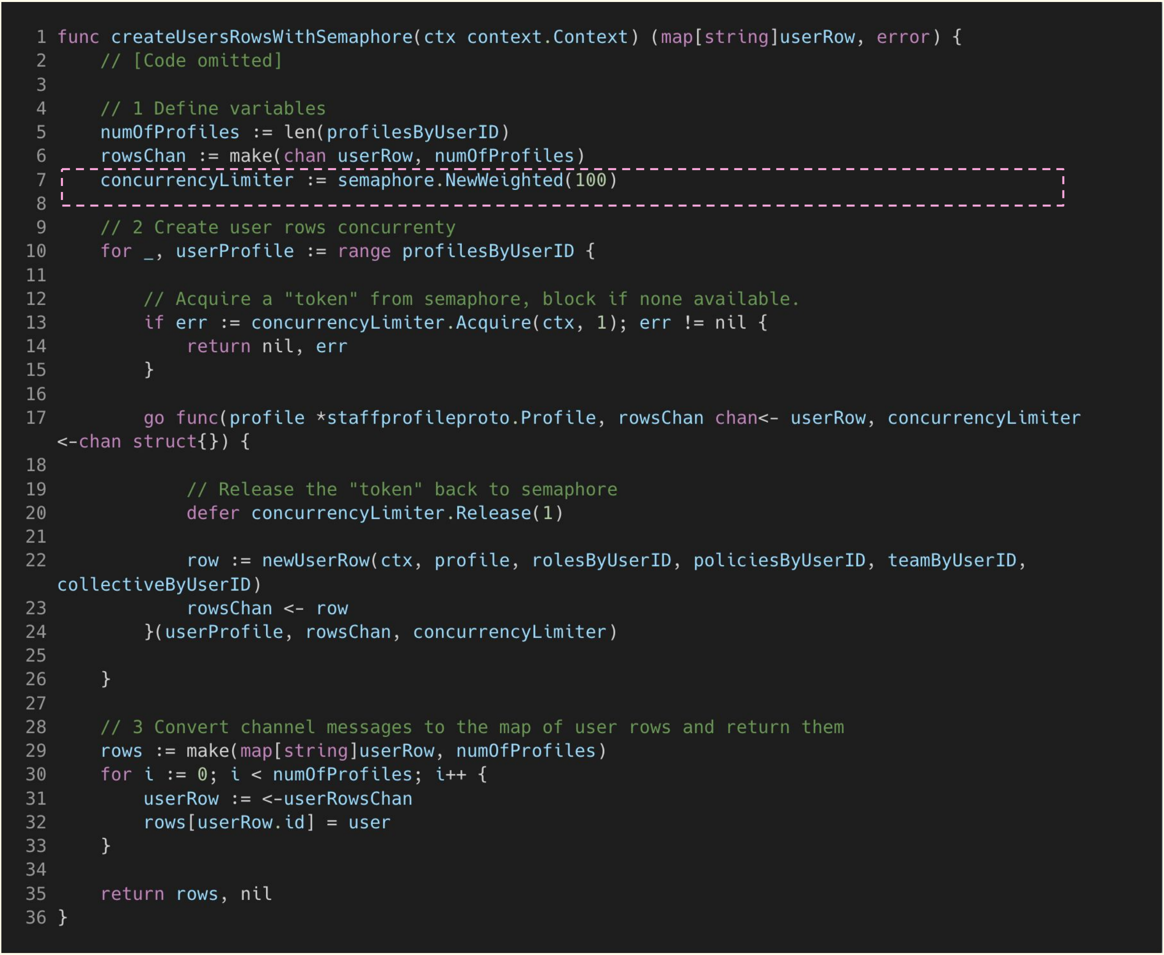
Context-aware. It allows timeout or tasks cancellation to be passed to Semaphore.

`Acquire(ctx context.Context, n int64)` (blocking) and `TryAcquire(n int64)` (non-blocking) version of getting the semaphore

Old Age

Create report rows with Buffered channel and Semaphore

The function `createUsersRows` uses a buffered channel and a semaphore concurrency primitives, to generate reports rows.

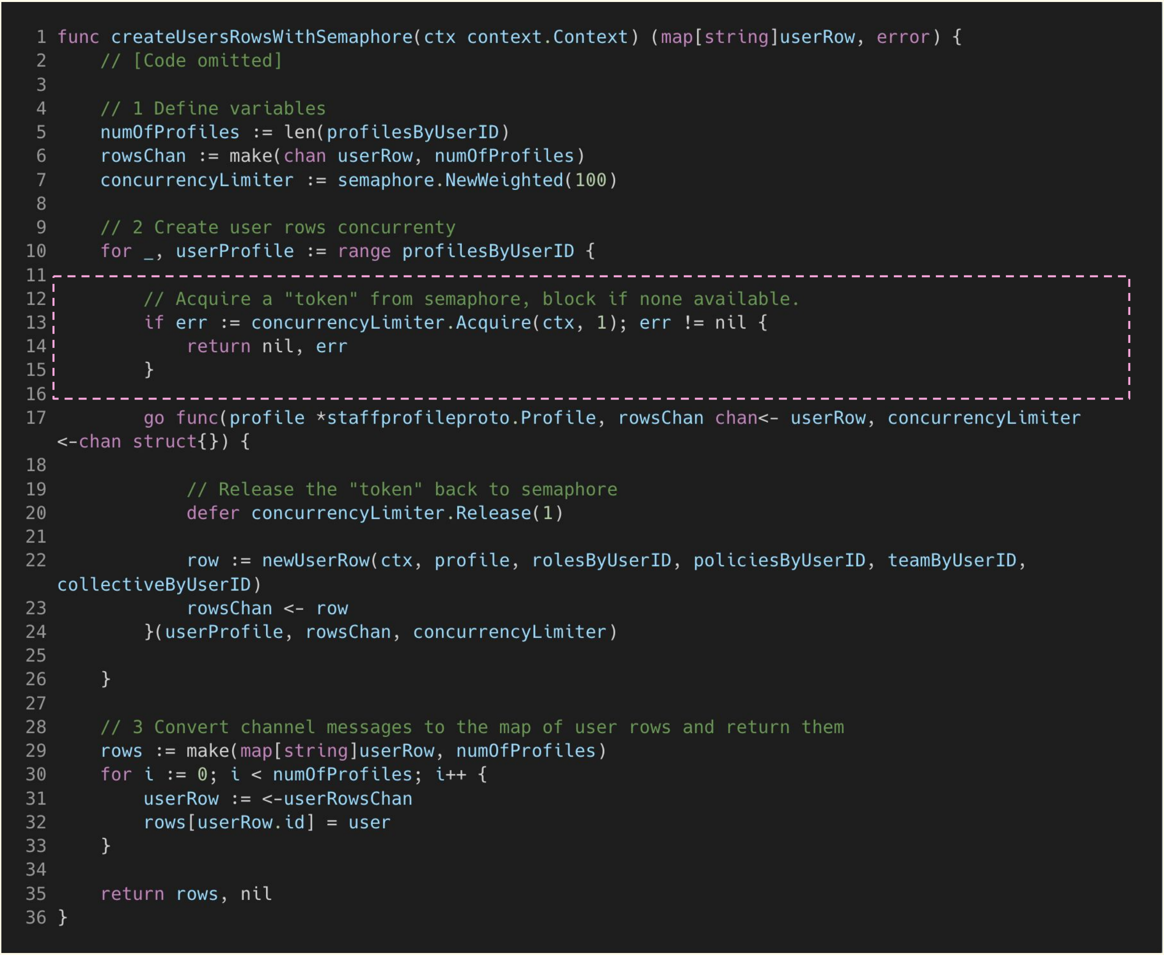


```go
func createUsersRowsWithSemaphore(ctx context.Context) (map[string]userRow, error) {
    // [Code omitted]

    // 1 Define variables
    numOfProfiles := len(profilesByUserID)
    rowsChan := make(chan userRow, numOfProfiles)
    concurrencyLimiter := semaphore.NewWeighted(100)

    // 2 Create user rows concurrenty
    for _, userProfile := range profilesByUserID {

        // Acquire a "token" from semaphore, block if none available.
        if err := concurrencyLimiter.Acquire(ctx, 1); err != nil {
            return nil, err
        }

        go func(profile *staffprofileproto.Profile, rowsChan chan<- userRow, concurrencyLimiter <-chan struct{}) {

            // Release the "token" back to semaphore
            defer concurrencyLimiter.Release(1)

            row := newUserRow(ctx, profile, rolesByUserID, policiesByUserID, teamByUserID, collectiveByUserID)
            rowsChan <- row
        }(userProfile, rowsChan, concurrencyLimiter)

    }

    // 3 Convert channel messages to the map of user rows and return them
    rows := make(map[string]userRow, numOfProfiles)
    for i := 0; i < numOfProfiles; i++ {
        userRow := <-userRowsChan
        rows[userRow.id] = user
    }

    return rows, nil
}
```

Old Age

Create report rows with Buffered channel and Semaphore

The function `createUsersRows` uses a buffered channel and a semaphore concurrency primitives, to generate reports rows.

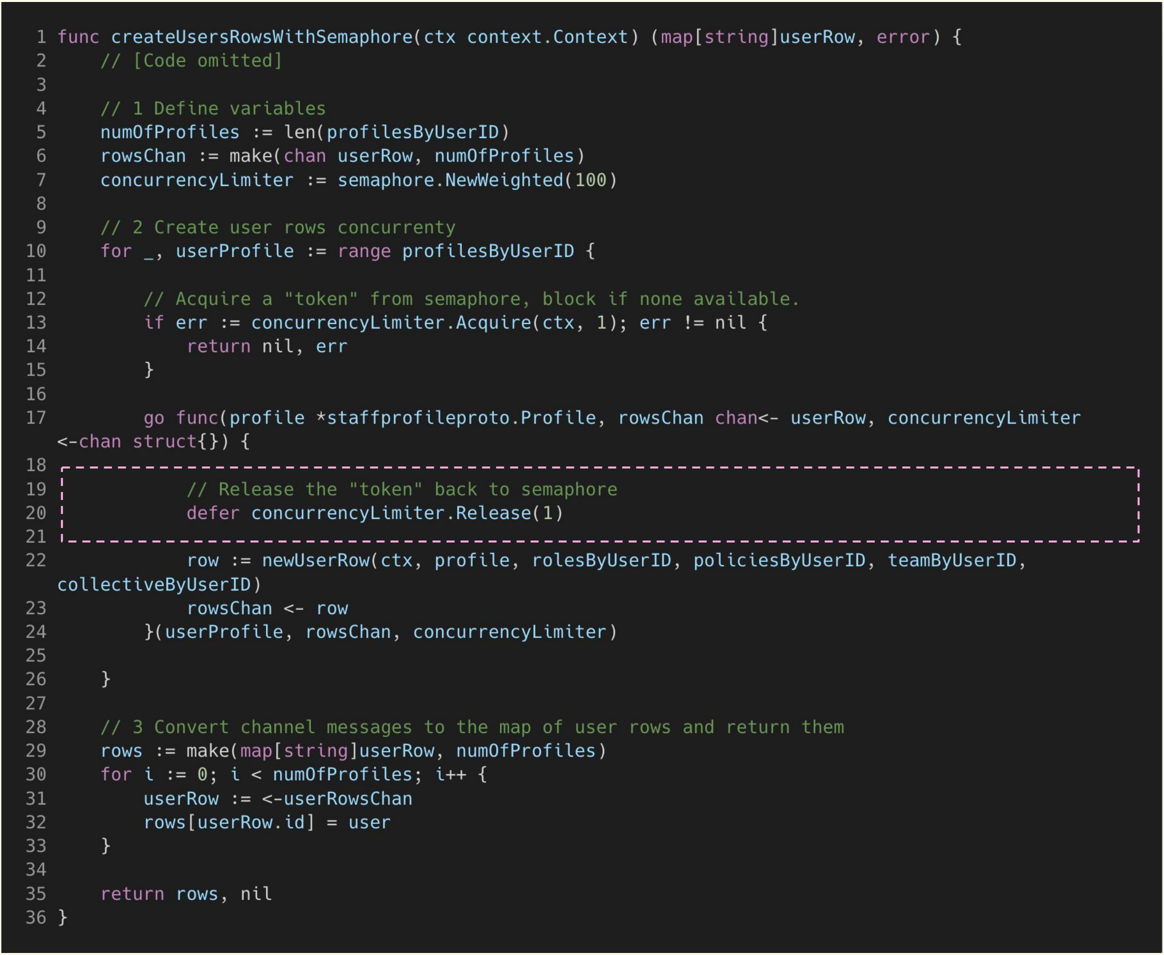


```go
func createUsersRowsWithSemaphore(ctx context.Context) (map[string]userRow, error) {
    // [Code omitted]

    // 1 Define variables
    numOfProfiles := len(profilesByUserID)
    rowsChan := make(chan userRow, numOfProfiles)
    concurrencyLimiter := semaphore.NewWeighted(100)

    // 2 Create user rows concurrenty
    for _, userProfile := range profilesByUserID {

        // Acquire a "token" from semaphore, block if none available.
        if err := concurrencyLimiter.Acquire(ctx, 1); err != nil {
            return nil, err
        }

        go func(profile *staffprofileproto.Profile, rowsChan chan<- userRow, concurrencyLimiter <-chan struct{}) {

            // Release the "token" back to semaphore
            defer concurrencyLimiter.Release(1)

            row := newUserRow(ctx, profile, rolesByUserID, policiesByUserID, teamByUserID, collectiveByUserID)
            rowsChan <- row
        }(userProfile, rowsChan, concurrencyLimiter)

    }

    // 3 Convert channel messages to the map of user rows and return them
    rows := make(map[string]userRow, numOfProfiles)
    for i := 0; i < numOfProfiles; i++ {
        userRow := <-userRowsChan
        rows[userRow.id] = user
    }

    return rows, nil
}
```

Old Age

Create report rows with Buffered channel and Semaphore

The function `createUsersRows` uses a buffered channel and a semaphore concurrency primitives, to generate reports rows.

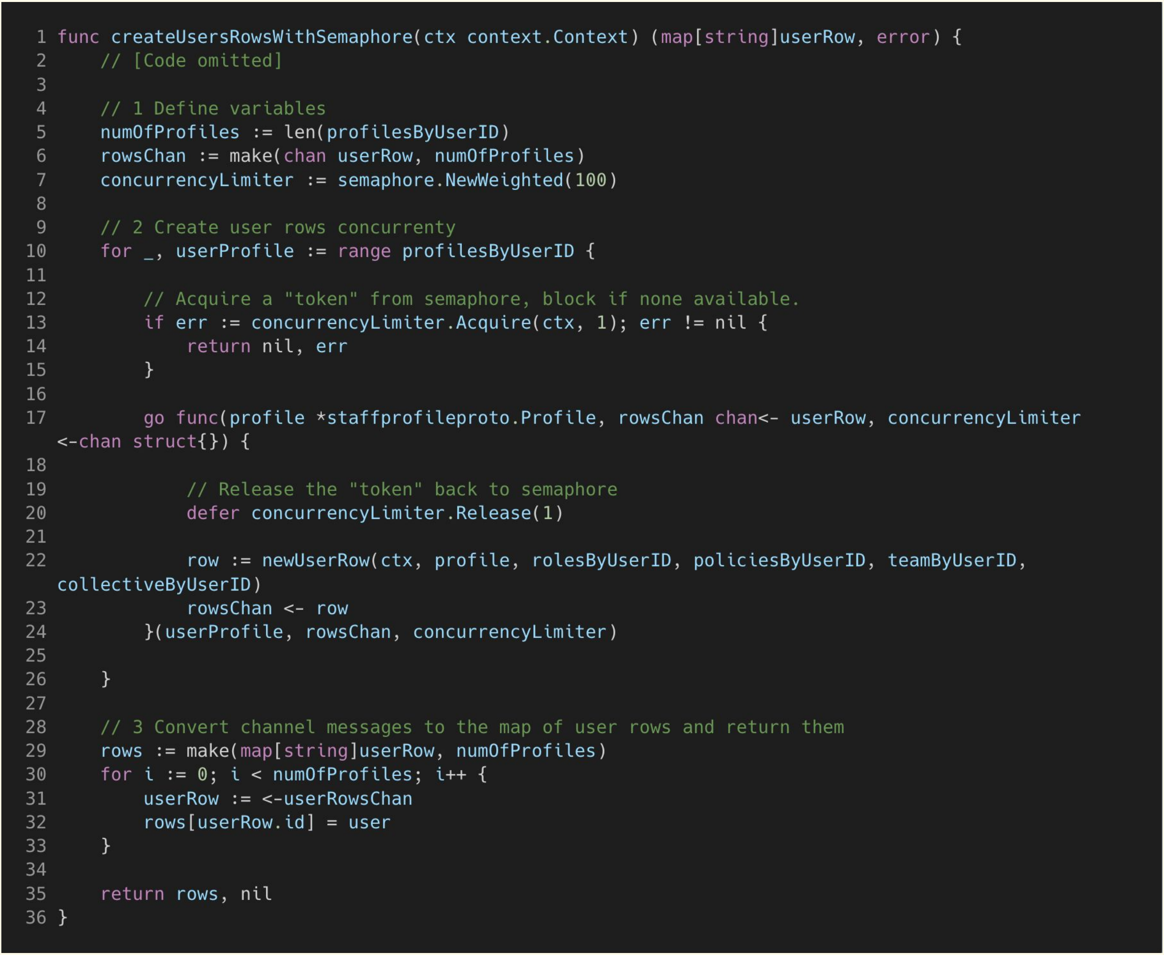


```go
func createUsersRowsWithSemaphore(ctx context.Context) (map[string]userRow, error) {
    // [Code omitted]

    // 1 Define variables
    numOfProfiles := len(profilesByUserID)
    rowsChan := make(chan userRow, numOfProfiles)
    concurrencyLimiter := semaphore.NewWeighted(100)

    // 2 Create user rows concurrenty
    for _, userProfile := range profilesByUserID {

        // Acquire a "token" from semaphore, block if none available.
        if err := concurrencyLimiter.Acquire(ctx, 1); err != nil {
            return nil, err
        }

        go func(profile *staffprofileproto.Profile, rowsChan chan<- userRow, concurrencyLimiter <-chan struct{}) {

            // Release the "token" back to semaphore
            defer concurrencyLimiter.Release(1)

            row := newUserRow(ctx, profile, rolesByUserID, policiesByUserID, teamByUserID, collectiveByUserID)
            rowsChan <- row
        }(userProfile, rowsChan, concurrencyLimiter)

    }

    // 3 Convert channel messages to the map of user rows and return them
    rows := make(map[string]userRow, numOfProfiles)
    for i := 0; i < numOfProfiles; i++ {
        userRow := <-userRowsChan
        rows[userRow.id] = user
    }

    return rows, nil
}
```

Old Age

Create report rows with Buffered channel and Semaphore

The function `createUsersRows` uses a buffered channel and a semaphore concurrency primitives, to generate reports rows.



```go
func createUsersRowsWithSemaphore(ctx context.Context) (map[string]userRow, error) {
    // [Code omitted]

    // 1 Define variables
    numOfProfiles := len(profilesByUserID)
    rowsChan := make(chan userRow, numOfProfiles)
    concurrencyLimiter := semaphore.NewWeighted(100)

    // 2 Create user rows concurrenty
    for _, userProfile := range profilesByUserID {

        // Acquire a "token" from semaphore, block if none available.
        if err := concurrencyLimiter.Acquire(ctx, 1); err != nil {
            return nil, err
        }

        go func(profile *staffprofileproto.Profile, rowsChan chan<- userRow, concurrencyLimiter <-chan struct{}) {

            // Release the "token" back to semaphore
            defer concurrencyLimiter.Release(1)

            row := newUserRow(ctx, profile, rolesByUserID, policiesByUserID, teamByUserID, collectiveByUserID)
            rowsChan <- row
        }(userProfile, rowsChan, concurrencyLimiter)

    }

    // 3 Convert channel messages to the map of user rows and return them
    rows := make(map[string]userRow, numOfProfiles)
    for i := 0; i < numOfProfiles; i++ {
        userRow := <-userRowsChan
        rows[userRow.id] = user
    }

    return rows, nil
}
```

Learning concurrency

Cultivating Knowledge and letting it blossom

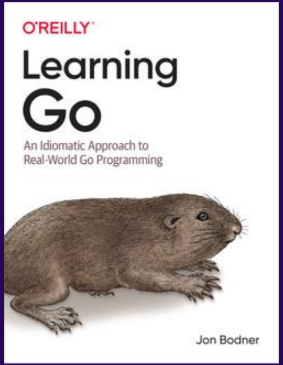
Learning

Learning resources

1

Learning Go: An Idiomatic Approach to Real-World Go Programming

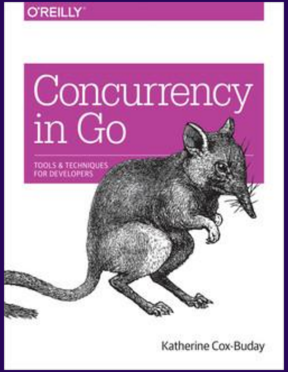
A book by Jon Bodner, published by O'Reilly. Chapter 13 is dedicated to Concurrency.



2

Concurrency in Go

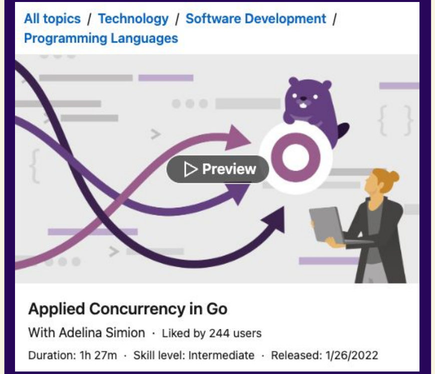
A book by Katherine Cox-Buday, published by O'Reilly.



3

Applied Concurrency in Go

A video course by Adelina Simion, released by LinkedIn learning.



All topics / Technology / Software Development / Programming Languages

Applied Concurrency in Go

With Adelina Simion · Liked by 244 users

Duration: 1h 27m · Skill level: Intermediate · Released: 1/26/2022