perplexity

Building a Temporal Workflow in Go for RSS Feed Processing and Bluesky Integration

Before diving into the code, let's understand what we're building: a Golang application that fetches RSS feeds defined in an OPML file, filters for yesterday's posts, and distributes them evenly throughout the day on Bluesky. This tutorial will introduce you to Temporal workflows and the Bluesky AtProtocol.

Introduction to Temporal and Bluesky

Temporal is a workflow engine that helps build reliable applications by providing fault tolerance, automatic retries, and state tracking[1]. Bluesky is a decentralized social network that uses the AT Protocol, with an API called XRPC for interactions[2].

Why Use Temporal for This Task?

Our workflow needs to:

- Process multiple feeds reliably
- Handle network failures when fetching feeds
- Schedule posts throughout the day
- Maintain state if the process crashes

Temporal excels at these requirements, offering durability and automatic recovery from failures[1] [3].

Project Structure

Let's organize our project:

```
/rss-to-bluesky
  - main.go           # Entry point and worker setup
  - workflow.go       # Temporal workflow definition
  - activities.go     # Activities for OPML parsing, feed fetching, and posting
  - bluesky.go        # Bluesky API client
  - go.mod            # Module dependencies
```

Setting Up Dependencies

Create a new Go module:

```
go mod init rss-to-bluesky
go get github.com/gilliek/go-opml/opml
```

```
go get github.com/mmcdole/gofeed
go get go.temporal.io/sdk
```

Defining Our Data Models

First, let's create our basic data structures:

```
// workflow.go
package main

import (
        "time"
)

// FeedItem represents a single post from an RSS feed
type FeedItem struct {
        Title     string
        Content   string
        URL       string
        Published time.Time
}

// PostSchedule contains info about when to post an item
type PostSchedule struct {
        Item      FeedItem
        PostTime  time.Time
}
```

Creating the Activities

Activities in Temporal are individual units of work that can fail and be retried independently[1]. Let's define our activities:

1. Fetching and Parsing OPML

```
// activities.go
package main

import (
        "context"
        "fmt"
        "io/ioutil"
        "net/http"

        "github.com/gilliek/go-opml/opml"
)

// FetchAndParseOPMLActivity retrieves an OPML file from GitHub and parses it
func FetchAndParseOPMLActivity(ctx context.Context, githubURL string) ([]string, error) {
        // Fetch OPML file
        resp, err := http.Get(githubURL)
        if err != nil {
```

```
                return nil, fmt.Errorf("failed to fetch OPML: %w", err)
        }
        defer resp.Body.Close()

        // Read the content
        content, err := ioutil.ReadAll(resp.Body)
        if err != nil {
                return nil, fmt.Errorf("failed to read OPML content: %w", err)
        }

        // Parse the OPML
        doc, err := opml.NewOPMLFromBytes(content)
        if err != nil {
                return nil, fmt.Errorf("failed to parse OPML: %w", err)
        }

        // Extract feed URLs
        var feedURLs []string
        for _, outline := range doc.Outlines() {
                if outline.XMLURL != "" {
                        feedURLs = append(feedURLs, outline.XMLURL)
                }

                // Check nested outlines
                for _, nestedOutline := range outline.Outlines {
                        if nestedOutline.XMLURL != "" {
                                feedURLs = append(feedURLs, nestedOutline.XMLURL)
                        }
                }
        }

        return feedURLs, nil
}
```

This activity fetches an OPML file from a GitHub URL and parses it using the go-opml library[4]. It returns a list of feed URLs defined in the OPML.

2. Fetching and Filtering RSS Feeds

```
// activities.go (continued)
import (
        "github.com/mmcdole/gofeed"
)

// FetchAndFilterFeedsActivity retrieves content from RSS feeds and filters for yesterday
func FetchAndFilterFeedsActivity(ctx context.Context, feedURLs []string) ([]FeedItem, err
        parser := gofeed.NewParser()
        var yesterdayItems []FeedItem

        // Calculate yesterday's date range
        now := time.Now()
        yesterdayStart := time.Date(now.Year(), now.Month(), now.Day()-1, 0, 0, 0, 0, now
        yesterdayEnd := yesterdayStart.Add(24 * time.Hour)
```

```
        for _, url := range feedURLs {
                feed, err := parser.ParseURL(url)
                if err != nil {
                        // Log error but continue with other feeds
                        fmt.Printf("Error parsing feed %s: %v\n", url, err)
                        continue
                }

                // Filter for posts from yesterday
                for _, item := range feed.Items {
                        pubDate := item.PublishedParsed
                        if pubDate != nil && pubDate.After(yesterdayStart) && pubDate.Bef
                                yesterdayItems = append(yesterdayItems, FeedItem{
                                        Title:     item.Title,
                                        Content:   item.Description,
                                        URL:       item.Link,
                                        Published: *pubDate,
                                })
                        }
                }
        }

        return yesterdayItems, nil
}
```

This activity uses the gofeed library[5] to parse RSS and Atom feeds. It filters the posts to only include those published yesterday.

3. Authenticating with Bluesky

```
// bluesky.go
package main

import (
        "bytes"
        "context"
        "encoding/json"
        "fmt"
        "net/http"
)

type BlueskyClient struct {
        Host       string
        AccessJWT  string
        RefreshJWT string
        DID        string
        Handle     string
}

type AuthResponse struct {
        Did        string `json:"did"`
        Handle     string `json:"handle"`
        AccessJwt  string `json:"accessJwt"`
        RefreshJwt string `json:"refreshJwt"`
```

```
}

// AuthenticateWithBlueskyActivity authenticates with Bluesky and returns a client
func AuthenticateWithBlueskyActivity(ctx context.Context, identifier, password string) (*
        host := "https://bsky.social"
        authURL := fmt.Sprintf("%s/xrpc/com.atproto.server.createSession", host)

        // Create auth request
        authReq := map[string]string{
                "identifier": identifier,
                "password":   password,
        }

        // Convert to JSON
        authJSON, err := json.Marshal(authReq)
        if err != nil {
                return nil, err
        }

        // Make the request
        resp, err := http.Post(authURL, "application/json", bytes.NewBuffer(authJSON))
        if err != nil {
                return nil, err
        }
        defer resp.Body.Close()

        // Parse response
        var authResp AuthResponse
        if err := json.NewDecoder(resp.Body).Decode(&authResp); err != nil {
                return nil, err
        }

        return &BlueskyClient{
                Host:       host,
                AccessJWT:  authResp.AccessJwt,
                RefreshJWT: authResp.RefreshJwt,
                DID:        authResp.Did,
                Handle:     authResp.Handle,
        }, nil
}
```

This activity implements the Bluesky authentication flow using the XRPC protocol[2]. It returns a client with the necessary tokens for future API calls.

4. Posting to Bluesky

```
// bluesky.go (continued)

// PostToBlueskyActivity posts content to Bluesky
func PostToBlueskyActivity(ctx context.Context, client *BlueskyClient, item FeedItem) err
        postURL := fmt.Sprintf("%s/xrpc/com.atproto.repo.createRecord", client.Host)

        // Create the post content with attribution and link
        text := fmt.Sprintf("%s\n\n%s", item.Title, item.URL)
```

```
        // Current time in RFC3339 format
        now := time.Now().Format(time.RFC3339)

        // Create the record
        postRecord := map[string]interface{}{
                "$type":     "app.bsky.feed.post",
                "text":      text,
                "createdAt": now,
                "langs":     []string{"en"},
        }

        // Create the post request
        postReq := map[string]interface{}{
                "repo":       client.DID,
                "collection": "app.bsky.feed.post",
                "record":     postRecord,
        }

        // Convert to JSON
        postJSON, err := json.Marshal(postReq)
        if err != nil {
                return err
        }

        // Create HTTP request
        req, err := http.NewRequest("POST", postURL, bytes.NewBuffer(postJSON))
        if err != nil {
                return err
        }

        // Add auth header
        req.Header.Set("Content-Type", "application/json")
        req.Header.Set("Authorization", "Bearer "+client.AccessJWT)

        // Make the request
        resp, err := http.DefaultClient.Do(req)
        if err != nil {
                return err
        }
        defer resp.Body.Close()

        // Check response
        if resp.StatusCode != http.StatusOK {
                return fmt.Errorf("post failed with status: %d", resp.StatusCode)
        }

        return nil
}
```

This activity creates a post on Bluesky using the XRPC Protocol[6]. It formats the post with the feed item's title and URL.

Creating the Workflow

Now, let's create our Temporal workflow that orchestrates these activities:

```
// workflow.go (continued)
package main

import (
        "time"

        "go.temporal.io/sdk/workflow"
)

// RSSToBlueskyWorkflow is the main workflow that orchestrates the entire process
func RSSToBlueskyWorkflow(ctx workflow.Context, githubURL, blueskyIdentifier, blueskyPass
        // Activity options
        activityOpts := workflow.ActivityOptions{
                StartToCloseTimeout: 10 * time.Minute,
                RetryPolicy: &temporal.RetryPolicy{
                        InitialInterval:    time.Second,
                        BackoffCoefficient: 2.0,
                        MaximumInterval:    time.Minute,
                        MaximumAttempts:    5,
                },
        }
        ctx = workflow.WithActivityOptions(ctx, activityOpts)

        // Step 1: Fetch and parse OPML
        var feedURLs []string
        err := workflow.ExecuteActivity(ctx, FetchAndParseOPMLActivity, githubURL).Get(ct
        if err != nil {
                return err
        }

        // Step 2: Fetch and filter feeds
        var feedItems []FeedItem
        err = workflow.ExecuteActivity(ctx, FetchAndFilterFeedsActivity, feedURLs).Get(ct
        if err != nil {
                return err
        }

        // Exit early if no items found
        if len(feedItems) == 0 {
                workflow.GetLogger(ctx).Info("No feed items from yesterday found")
                return nil
        }

        // Step 3: Authenticate with Bluesky
        var client *BlueskyClient
        err = workflow.ExecuteActivity(ctx, AuthenticateWithBlueskyActivity, blueskyIdent
        if err != nil {
                return err
        }

        // Step 4: Schedule posts evenly throughout the day
        now := workflow.Now(ctx)
```

```
        startTime := time.Date(now.Year(), now.Month(), now.Day(), 9, 0, 0, 0, now.Locati
        endTime := time.Date(now.Year(), now.Month(), now.Day(), 17, 0, 0, 0, now.Locatio

        // Calculate time between posts
        totalDuration := endTime.Sub(startTime)
        interval := totalDuration / time.Duration(len(feedItems))

        // Schedule and post each item
        for i, item := range feedItems {
                postTime := startTime.Add(interval * time.Duration(i))

                // Sleep until the scheduled post time
                sleepDuration := postTime.Sub(workflow.Now(ctx))
                if sleepDuration > 0 {
                        workflow.Sleep(ctx, sleepDuration)
                }

                // Post to Bluesky
                err := workflow.ExecuteActivity(ctx, PostToBlueskyActivity, client, item)
                if err != nil {
                        workflow.GetLogger(ctx).Error("Failed to post item", "error", err
                        continue
                }

                workflow.GetLogger(ctx).Info("Posted item to Bluesky", "title", item.Titl
        }

        return nil
}
```

This workflow definition[1] [7] orchestrates our activities:

1. It fetches and parses the OPML file
2. It fetches and filters RSS feeds for yesterday's posts
3. It authenticates with Bluesky
4. It schedules the posts evenly throughout the day (9am-5pm) and posts them at the scheduled times using Temporal's durable timers[8]

Setting Up the Worker

Now, let's set up our Temporal worker:

```
// main.go
package main

import (
        "log"
        "os"
        "os/signal"

        "go.temporal.io/sdk/client"
        "go.temporal.io/sdk/worker"
)
```

```
func main() {
        // Create Temporal client
        c, err := client.Dial(client.Options{})
        if err != nil {
                log.Fatalf("Failed to create Temporal client: %v", err)
        }
        defer c.Close()

        // Create worker
        w := worker.New(c, "rss-bluesky-task-queue", worker.Options{})

        // Register workflow and activities
        w.RegisterWorkflow(RSSToBlueskyWorkflow)
        w.RegisterActivity(FetchAndParseOPMLActivity)
        w.RegisterActivity(FetchAndFilterFeedsActivity)
        w.RegisterActivity(AuthenticateWithBlueskyActivity)
        w.RegisterActivity(PostToBlueskyActivity)

        // Start worker
        err = w.Start()
        if err != nil {
                log.Fatalf("Failed to start worker: %v", err)
        }

        // Wait for interrupt
        sigCh := make(chan os.Signal, 1)
        signal.Notify(sigCh, os.Interrupt)
        <-sigCh

        // Stop worker
        w.Stop()
}
```

This sets up the worker process that will execute our workflow and activities[1].

Scheduling the Workflow

Finally, let's create a client to schedule our workflow to run daily:

```
// scheduler.go
package main

import (
        "context"
        "log"

        "go.temporal.io/sdk/client"
)

func main() {
        // Create Temporal client
        c, err := client.Dial(client.Options{})
        if err != nil {
                log.Fatalf("Failed to create Temporal client: %v", err)
```

```
        }
        defer c.Close()

        // GitHub URL with OPML file, Bluesky credentials
        githubURL := "https://github.com/user/repo/raw/main/feeds.opml"
        blueskyIdentifier := "your-email@example.com"
        blueskyPassword := "your-app-password"

        // Create a schedule
        scheduleHandle, err := c.ScheduleClient().Create(context.Background(), client.Sch
                ID: "rss-to-bluesky-schedule",
                Spec: client.ScheduleSpec{
                        // Run every day at 5 AM
                        CronExpressions: []string{"0 5 * * *"},
                },
                Action: &client.ScheduleWorkflowAction{
                        ID:        "rss-to-bluesky-workflow",
                        Workflow:  RSSToBlueskyWorkflow,
                        Args:      []interface{}{githubURL, blueskyIdentifier, blueskyPas
                        TaskQueue: "rss-bluesky-task-queue",
                },
        })
        if err != nil {
                log.Fatalf("Failed to create schedule: %v", err)
        }

        log.Printf("Created schedule: %s\n", scheduleHandle.GetID())
}
```

This creates a daily schedule[9] [10] [11] [12] that runs our workflow every day at 5 AM. The workflow will then fetch yesterday's posts and schedule them throughout the day.

Understanding Key Temporal Concepts

Let's explore some key Temporal concepts used in this application:

1. Workflows and Activities

Workflows are the core building block in Temporal. They define the overall flow of your application[1]. Activities are individual tasks that can fail and be retried independently[3]. In our case, the workflow orchestrates several activities - fetching OPML, parsing feeds, posting to Bluesky.

2. Durable Timers

One of Temporal's powerful features is durable timers[8]. In our workflow, we use `workflow.Sleep()` to schedule posts throughout the day. Unlike regular sleep functions, Temporal's sleep persists even if your application crashes. When it restarts, the timer continues from where it left off.

3. Scheduling

Temporal provides built-in scheduling functionality[9] [11]. We use it to run our workflow daily at 5 AM. This replaces traditional cron jobs with a more reliable, observable solution[12].

4. Fault Tolerance

If any activity fails (e.g., network issues while fetching feeds), Temporal automatically retries it according to the specified retry policy[1]. This eliminates the need to write complex error handling code.

Understanding Bluesky XRPC Concepts

Let's also review some key Bluesky AtProtocol concepts:

1. Authentication

Bluesky uses JWT tokens for authentication[2]. We obtain these tokens by creating a session with the user's identifier and password.

2. DID (Decentralized Identifier)

Each Bluesky account has a DID, which is a persistent identifier[2]. When creating posts, we specify the user's DID as the repository.

3. Record Creation

Posts in Bluesky are created using the `com.atproto.repo.createRecord` endpoint[6]. We specify the collection as `app.bsky.feed.post` and include the post content in the record.

Conclusion

You've now built a complete Temporal workflow that fetches RSS feeds and posts them to Bluesky on a schedule. This application demonstrates key Temporal concepts like durable workflows, activities, retries, and scheduling. It also shows how to integrate with external services like GitHub and Bluesky.

By using Temporal, we've created an application that's fault-tolerant and resilient to failures. If the process crashes, posts will still be delivered at their scheduled times when the application restarts.

This project serves as a great introduction to both Temporal workflows and the Bluesky AtProtocol, showing how they can be combined to build reliable applications that interact with modern decentralized social platforms.

1. https://learn.temporal.io/getting_started/go/hello_world_in_go/
2. https://www.manton.org/2023/04/29/getting-started-with.html

3. https://docs.temporal.io/develop/go/
4. https://github.com/gilliek/go-opml
5. https://webscraping.fyi/lib/go/gofeed/
6. https://docs.bsky.app/docs/advanced-guides/posts
7. https://learn.temporal.io/getting_started/go/first_program_in_go/
8. https://docs.temporal.io/develop/go/timers
9. https://docs.temporal.io/develop/go/schedules
10. https://docs.temporal.io/cron-job
11. https://brojonat.com/posts/temporal-schedules/
12. https://temporal.io/blog/how-do-i-convert-my-cron-into-a-schedule