Go Context Library

By:

Smita Vijayakumar

Agenda

1 Introduction
- *Context Management in Distributed Systems*

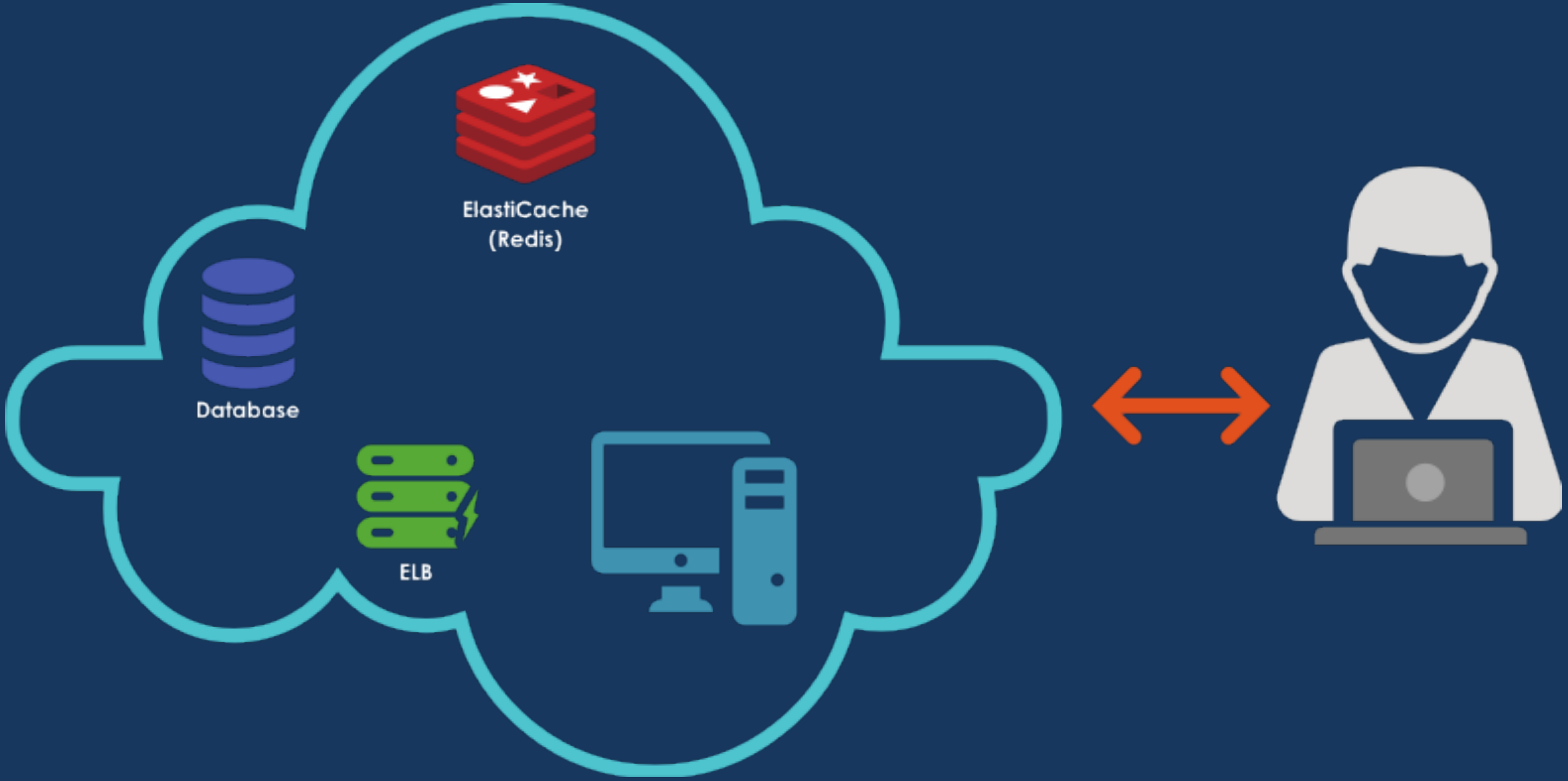
2 Go Context Library

3 Example Use case

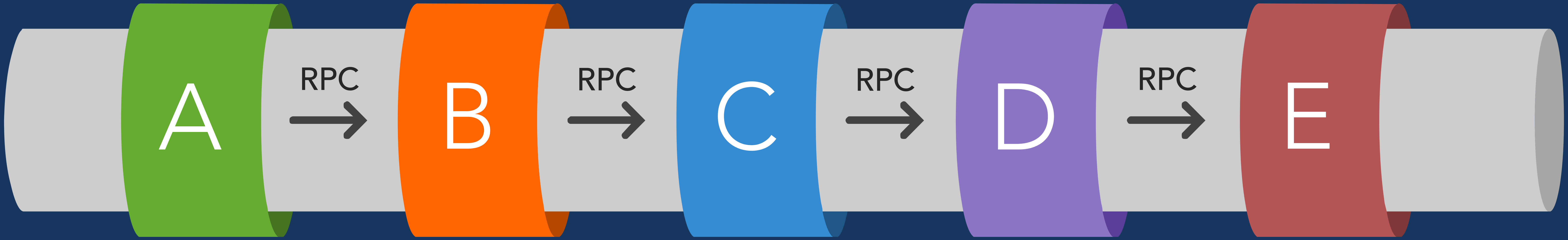
4 Potential Pitfalls

Introduction

Distributed Data Flow



Pipelined Processing



Go Context

- Defines *Context* type
- Carries request-scoped values:
  - Deadlines
  - Cancellation signals
  - Others
- Works across API boundaries
  - Also between processes

Details

```go
type Context interface {
    // Done returns a channel that is closed when this Context is canceled
    // or times out.
    Done() <-chan struct{}

    // Err indicates why this context was canceled, after the Done channel
    // is closed.
    Err() error

    // Deadline returns the time when this Context will be canceled, if any.
    Deadline() (deadline time.Time, ok bool)

    // Value returns the value associated with key or nil if none.
    Value(key interface{}) interface{}
}
```

Primary Use Cases

1. Distributed Tracing
2. Request Cancellation

Example 1 - Distributed Tracing

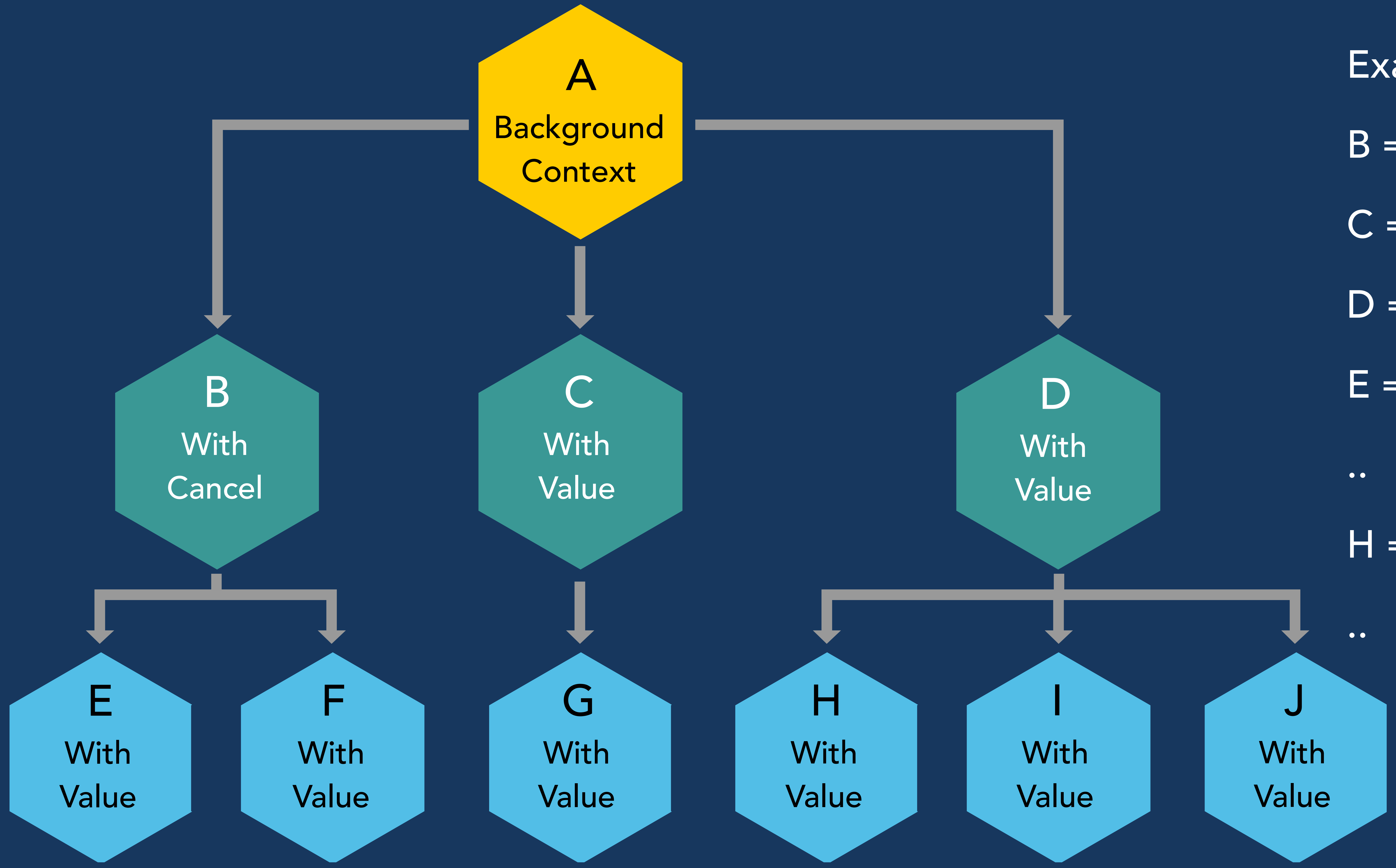
```
shared.thrift

struct ContextValue {
  /* RequestID is the request ID for that particular request.
   * Eg: This can be a UUID of the form: c66e30e7-5a91-435c-bf18-e63469d472b3
   */
  1: required string requestID
  2: required string GUID
}
```

Types of Context Nodes

1. Background Node
2. Value Node
3. Cancellable Node
4. TODO() Node

Types - A Context Tree



Example - A = context.Background()

B = context.WithCancel(A)

C = context.WithValue(A, “Value”, “C”)

D = context.WithValue(A, “Value”, “D”)

E = context.WithValue(B, “B Key”, “E”)

..

H = context.WithValue(D, “D Key”, “H”)

..

Example 2 - UTIL Package - Define set and get Context

```
package util

const uuidKey string = “NewID”

func SetContextValue(ctx context.Context, u uuid.UUID) context.Context {
    return context.WithValue(ctx, uuidKey, u)
}

func GetContextValue(ctx context.Context) (uuid.UUID, bool) {
    // ctx.Value returns nil if ctx has no value for the key;
    // the uuid.UUID type assertion returns ok=false for nil.
    u, ok := ctx.Value(uuidKey).(uuid.UUID)
    return u, ok
}
```

Example 2 - Cancellable Request - Set the Context

```go
package request

func startRequest(event *event.Event, timeout time.Duration) {
    var (
        ctx    context.Context
        cancel context.CancelFunc
    )
    ctx, cancel = context.WithTimeout(context.Background(), timeout)
    defer cancel()

    // Extract UUID from the event
    id := getIDFromEvent(event)

    // Store the id inside the context tree value node
    ctx = util.SetContextValue(ctx, id)

    // Send the event out to the upstream server
    status, err := processor.Server(ctx, event)
}
```

Example 2 - Cancellable Request – Get and Handle Context

```
package processor

func Server(ctx *context.Context, event *event.Event) error {
    //Get Event fields and process
    evName := event.GetEventName()
    if uuid, ok := util.FromContext(ctx); !ok {
        return false, errors.New(“Not a valid UUID to process”)
    }

    var evId string
    switch evName {
    case DownstreamEvent1:
        evId = event.DE1.GetEventId()
    case DownstreamEvent2:
        evId = event.DE2.GetEventId()
    }
```

Example 2 - Cancellable Request – Get and Handle Context

```
    // Create processEvent
    p := make(chan error)

    // This routine does highly latent event processing operation
    go func(ctx *context.Context, evId string, event *event.Event) error {
        p <- processEvent(ctx, evId, event)
    }()

    // Wait till event is either processed or times out
    var err error
    select {
    case <-ctx.Done():
    //go cleanup the event processing
    cancelRequestAndCleanup(event)
    <-p
    return ctx.Err()
    case err = <-p:
        break
    }
    return err
}
```

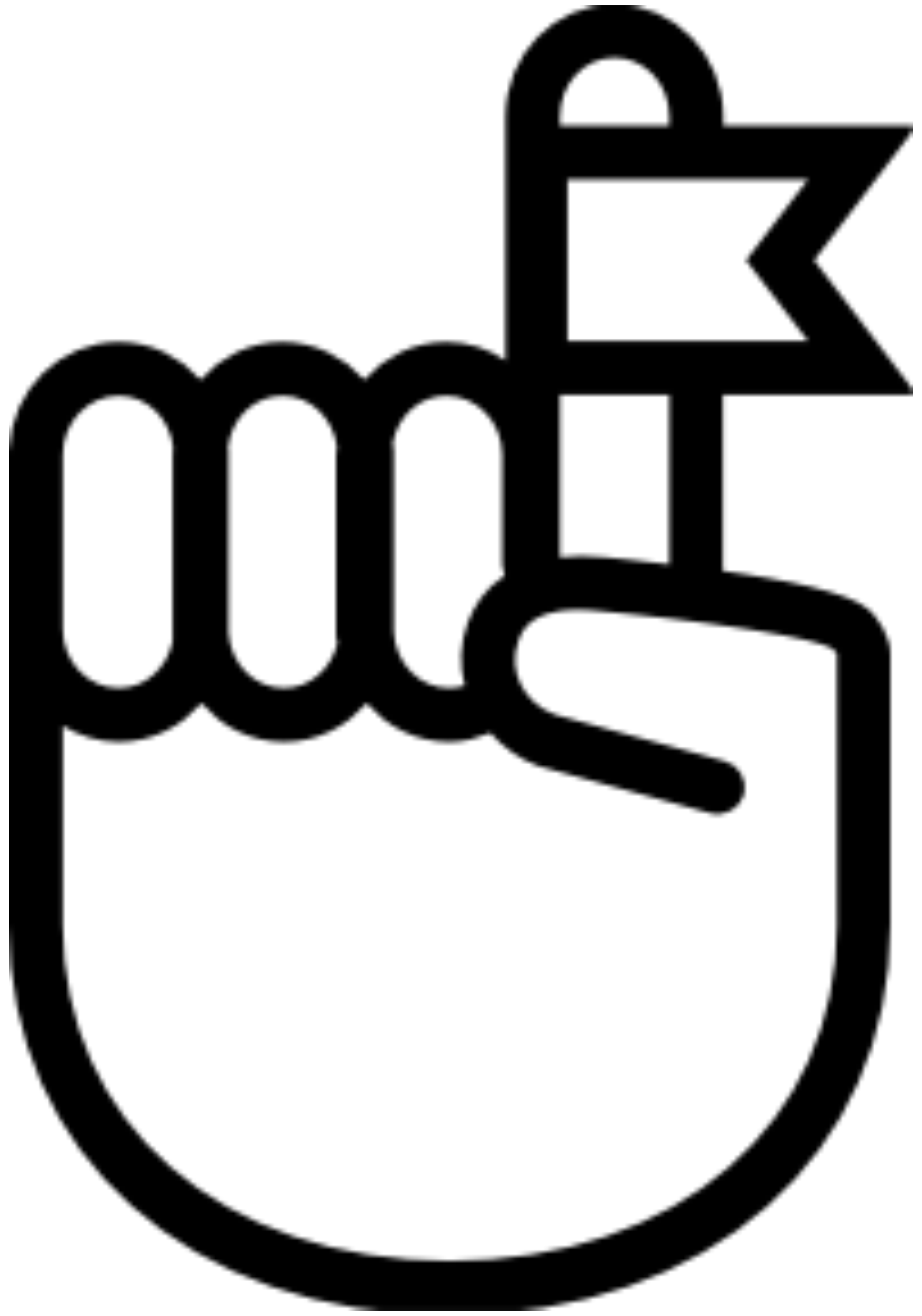
Summary - Use Cases In Distributed System

1. Ease of handling multiple, concurrent requests

2. Flow Traceability and Fingerprinting

3. Time Sensitive and Cancellable Request Processing

Remember!



Pitfalls…

Code Complexity:

For larger systems, complexity is the downside

Pitfalls…

Inter-Process Boundaries:

Difficult to actually implement passing cancellable signals downstream

Pitfalls…

Garbage Collection:

Don’t store context variables inside structures

Pitfalls…

Querying:

Holding the right context node

Thank you!

For any queries:

Smita Vijayakumar | smita@exotel.in