Cloud Provider

How we implemented an offline first synchronised store at Fyne Labs

What we wanted from a cloud provider

- Sync data between all our devices efficiently
- Great offline
- We wanted it to fit in the Go ecosystem - boltDB
- Conflict free with CRDTs and a last write win algorithm

Our own preferences

```go
type lalPreferences struct {
    lock            sync.RWMutex
    changeListeners []func()
    wg              *sync.WaitGroup
    db              *lal.DB
}

// Ensure conformity with fyne preferences
var _ (fyne.Preferences) = (*lalPreferences)(nil)

// CloudPreferences sets app to use lal preferences
func (d *lalStore) CloudPreferences(a fyne.App) fyne.Preferences {
    p := newLalPreferences(d.db)
    d.prefs = p
    return p
}
```

```
// Preferences describes the ways that an app can save and load user preferences
type Preferences interface {
    // Bool looks up a boolean value for the key
    Bool(key string) bool
    // BoolWithFallback looks up a boolean value and returns the given fallback if not found
    BoolWithFallback(key string, fallback bool) bool
    // SetBool saves a boolean value for the given key
    SetBool(key string, value bool)

    // Float looks up a float64 value for the key
    Float(key string) float64
    // FloatWithFallback looks up a float64 value and returns the given fallback if not found
    FloatWithFallback(key string, fallback float64) float64
    // SetFloat saves a float64 value for the given key
    SetFloat(key string, value float64)

    // Int looks up an integer value for the key
    Int(key string) int
    // IntWithFallback looks up an integer value and returns the given fallback if not found
    IntWithFallback(key string, fallback int) int
    // SetInt saves an integer value for the given key
    SetInt(key string, value int)

    // String looks up a string value for the key
    String(key string) string
    // StringWithFallback looks up a string value and returns the given fallback if not found
    StringWithFallback(key, fallback string) string
    // SetString saves a string value for the given key
    SetString(key string, value string)

    // RemoveValue removes a value for the given key (not currently supported on iOS)
    RemoveValue(key string)

    // AddChangeListener allows code to be notified when some preferences change. This will fire on any update.
    AddChangeListener(func())

    // ChangeListeners returns a list of the known change listeners for this preference set.
    //
    // Since: 2.3
    ChangeListeners() []func()
}
```

Setters - Conforming to the interface

```go
// SetBool saves a boolean value for the given key
func (p *lalPreferences) SetBool(key string, value bool) {
    p.set(key, value)
}


// SetFloat saves a float64 value for the given key
func (p *lalPreferences) SetFloat(key string, value float64) {
    p.set(key, value)
}


// SetInt saves an integer value for the given key
func (p *lalPreferences) SetInt(key string, value int) {
    p.set(key, value)
}
```

Set

```
func (p *lalPreferences) set(key string, value interface{}) {
    if p.db == nil {
        return
    }
    if err := p.db.Update(func(tx *lal.Tx) error {
        b := tx.Bucket([]byte("preferences"))

        var buf bytes.Buffer
        enc := gob.NewEncoder(&buf)
        err := enc.Encode(value)
        if err != nil {
            return err
        }
        return b.Put([]byte(key), buf.Bytes())
    }); err != nil {
        fyne.LogError("Can't update bucket", err)
    }

    p.fireChange()
}
```

Fire Change

```
func (p *lalPreferences) fireChange() {
    p.lock.RLock()
    defer p.lock.RUnlock()

    for _, l := range p.changeListeners {
        p.wg.Add(1)
        go func(listener func()) {
            defer p.wg.Done()
            listener()
        }(l)
    }

    p.wg.Wait()
}
```

Getter - String

```go
// String looks up a string value for the key
func (p *lalPreferences) String(key string) string {
    return p.StringWithFallback(key, "")
}

// StringWithFallback looks up a string value and returns the given fallback if not found
func (p *lalPreferences) StringWithFallback(key, fallback string) string {
    var val string
    err := p.get(key, &val)
    if err != nil {
        return fallback
    }
    return val
}
```

Get

```go
func (p *lalPreferences) get(key string, target interface{}) error {
    return p.db.View(func(tx *lal.Tx) error {
        b := tx.Bucket([]byte("preferences"))
        if b == nil {
            return errors.New("bucket does not exist")
        }
        val := b.Get([]byte(key))
        if len(val) == 0 {
            return errors.New("value length is 0")
        }
        buf := bytes.NewBuffer(val)
        dec := gob.NewDecoder(buf)
        return dec.Decode(target)
    })
}
```

Remove

```
func (p *lalPreferences) remove(key string) {
    if p.db == nil {
        return
    }
    if err := p.db.Update(func(tx *lal.Tx) error {
        b := tx.Bucket([]byte("preferences"))
        return b.Delete([]byte(key))
    }); err != nil {
        fyne.LogError("Can't delete bucket", err)
    }

    p.fireChange()
}
```

Network Synchronisation

```go
// newLalPreferences creates a new preferences implementation
func newLalPreferences(db *lal.DB) *lalPreferences {
    p := &lalPreferences{
        wg: &sync.WaitGroup{},
        db: db,
    }

    p.db.RegisterSynchronizationListener(lal.NewSynchronizationListener(func(newData bool, err
error) {
        if newData {
            p.fireChange()
        }
        if err != nil {
            fyne.LogError("Could not synchronize with lal service", err)
        }
    }))

    return p
}
```