Struct Embedding, Instrumentation, and Code Generation

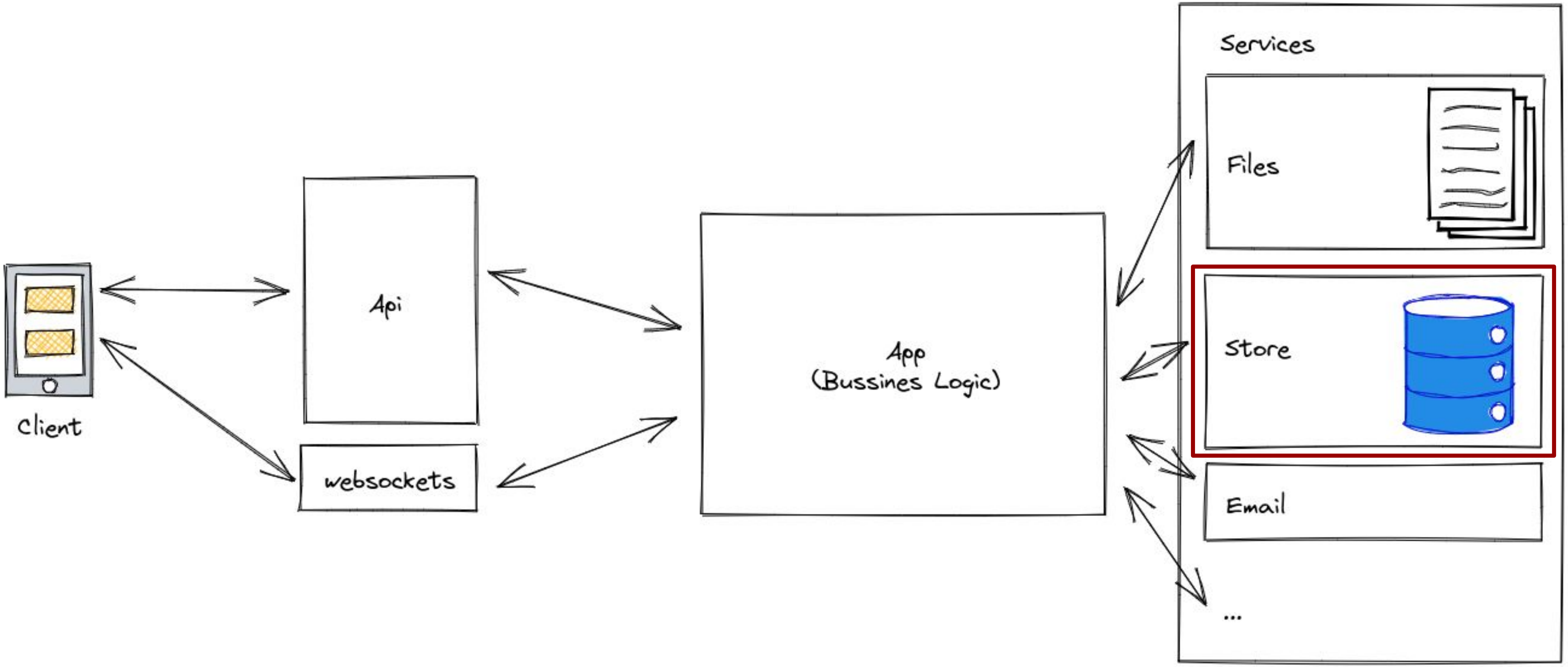
// Jesús Espino (Software Engineer at Mattermost)

Mattermost

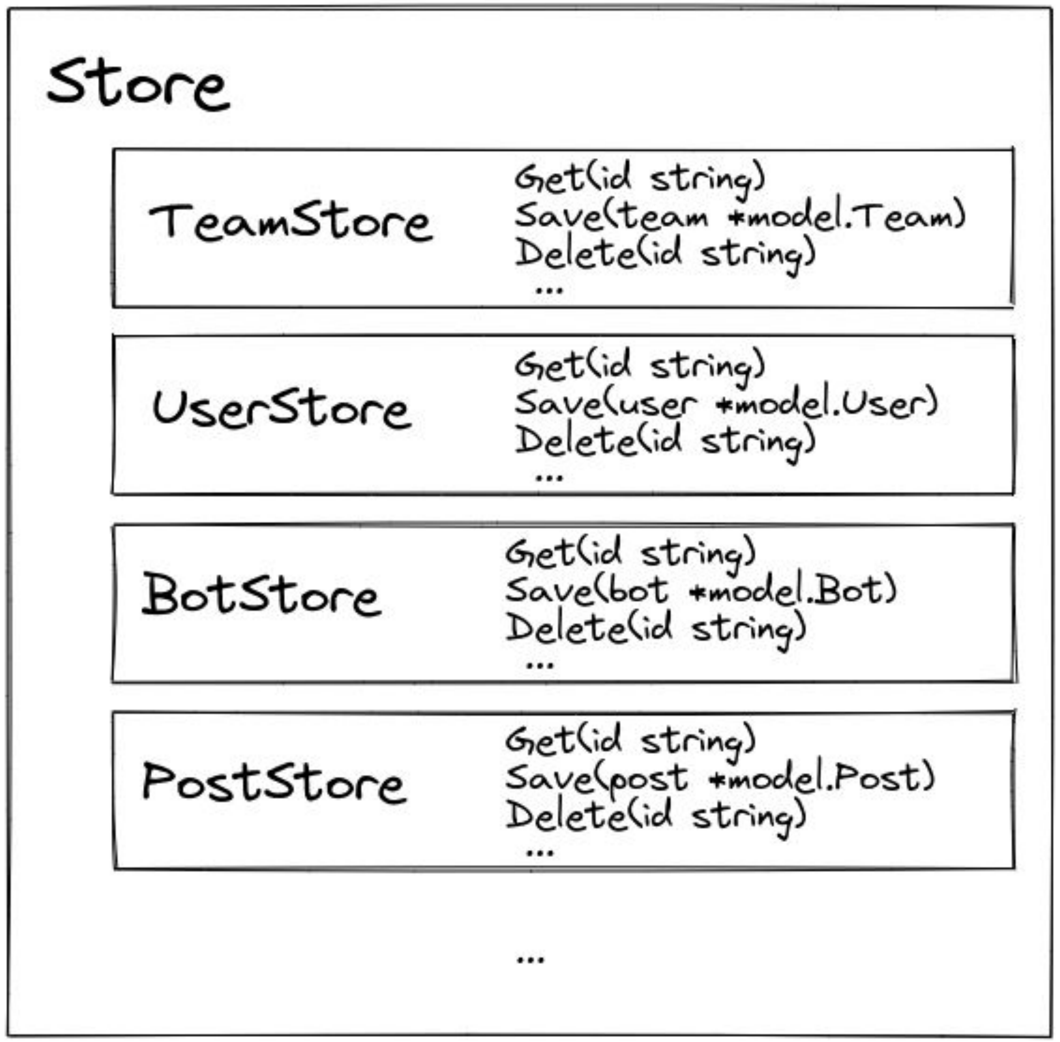
What is Mattermost?

- A communication platform.
- Backend written in Go 1.15.
- Frontend written in Typescript/Javascript/React.
- Focused on security and performance.
- Open Source/Open Core.
- Self Hosted and Cloud-based deployments.

What are the main pieces?



What does our store look like?



What does our store look like?

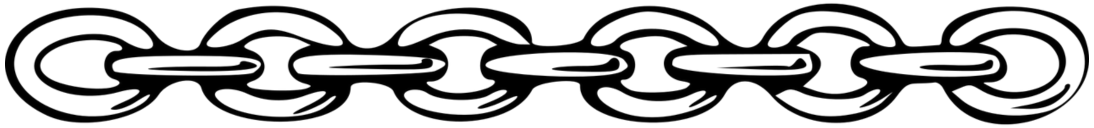
```
type Store interface {
    Team() TeamStore
    Channel() ChannelStore
    Post() PostStore
    Thread() ThreadStore
    User() UserStore
    Bot() BotStore
    Audit() AuditStore
    ...
}
```

```
type TeamStore interface {
    Save(team *model.Team) (*model.Team, error)
    Update(team *model.Team) (*model.Team, error)
    Get(id string) (*model.Team, error)
    GetByName(name string) (*model.Team, error)
    GetByNames(name []string) ([]*model.Team, error)
    SearchAll(opts *model.TeamSearch) ([]*model.Team, error)
    ...
}
```

What problem were we trying to solve?

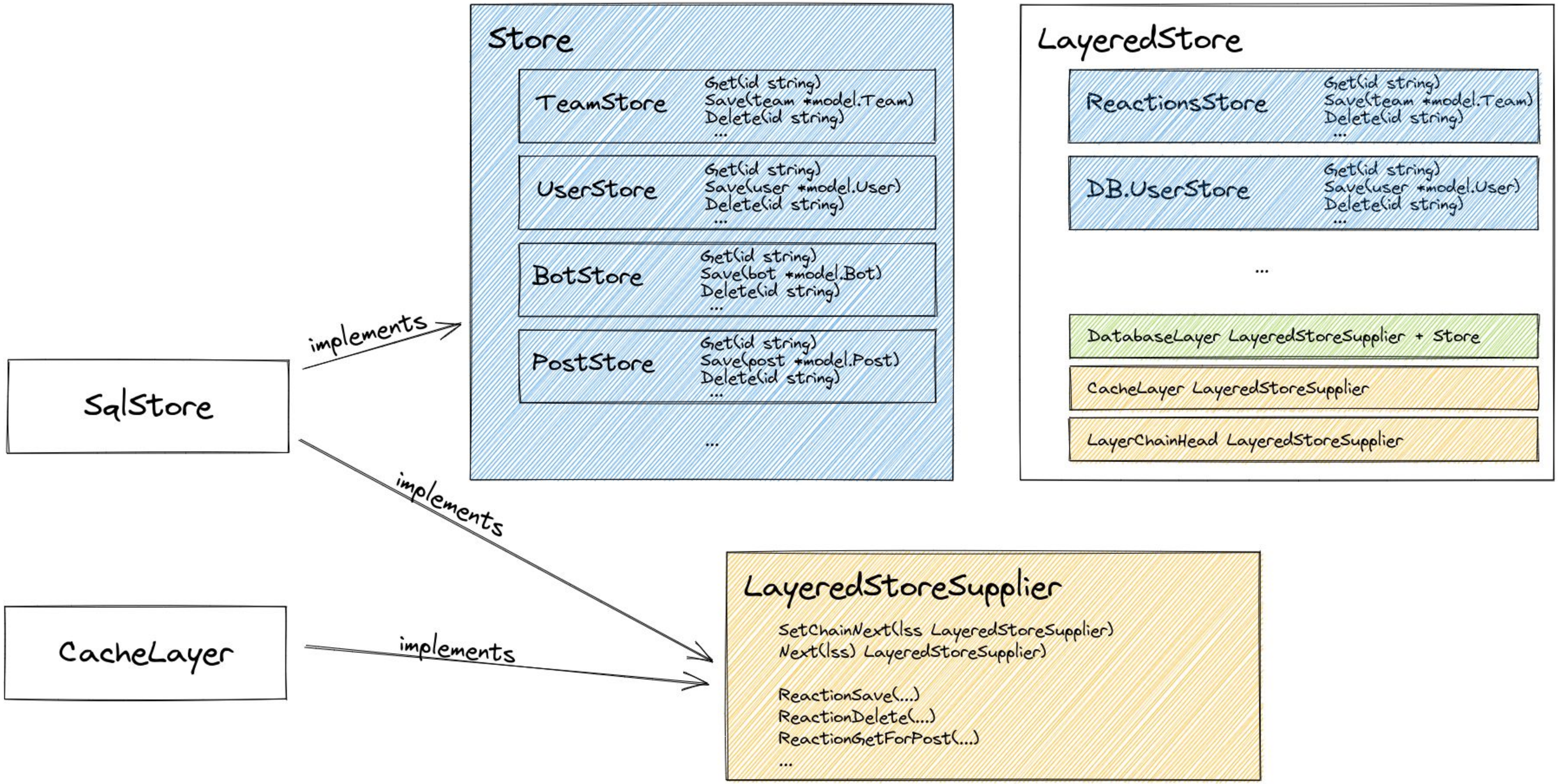
- We wanted add a cache on our store.
- We didn’t want to mix responsibilities in the store code.

Our initial approach



- Apply a middleware pattern.
- We create a wrapper struct with its own interface.
- Some cache methods, a `next` method.
- A set of “layered interfaces”.
- We needed some time to understand what’s going on under the hood.

Our initial approach



Store

TeamStore
Get(id string)
Save(team *model.Team)
Delete(id string)
...

UserStore
Get(id string)
Save(user *model.User)
Delete(id string)
...

BotStore
Get(id string)
Save(bot *model.Bot)
Delete(id string)
...

PostStore
Get(id string)
Save(post *model.Post)
Delete(id string)
...

...

LayeredStore

ReactionsStore
Get(id string)
Save(team *model.Team)
Delete(id string)
...

DB.UserStore
Get(id string)
Save(user *model.User)
Delete(id string)
...

...

DatabaseLayer LayeredStoreSupplier + Store

CacheLayer LayeredStoreSupplier

LayerChainHead LayeredStoreSupplier

SqlStore

CacheLayer

implements

implements

implements

LayeredStoreSupplier

SetChainNext(lss LayeredStoreSupplier)
Next(lss) LayeredStoreSupplier)

ReactionSave(...)
ReactionDelete(...)
ReactionGetForPost(...)
...

How we did it

```
type LayeredStoreDatabaseLayer interface {
    LayeredStoreSupplier
    Store
}

type LayeredStore struct {
    TmpContext      context.Context
    ReactionStore   ReactionStore
    RoleStore       RoleStore
    DatabaseLayer   LayeredStoreDatabaseLayer
    LocalCacheLayer *LocalCacheSupplier
    RedisLayer      *RedisSupplier
    LayerChainHead  LayeredStoreSupplier
}

type LayeredStoreSupplier interface {
    SetChainNext(LayeredStoreSupplier)
    Next() LayeredStoreSupplier
    ReactionSave(ctx context.Context, reaction *model.Reaction, hints ...LayeredStoreHint) *LayeredStoreSupplierResult
    ReactionDelete(ctx context.Context, reaction *model.Reaction, hints ...LayeredStoreHint) *LayeredStoreSupplierResult
    ReactionGetForPost(ctx context.Context, postId string, hints ...LayeredStoreHint) *LayeredStoreSupplierResult
    ...
}
```

What went well? What didn’t work?

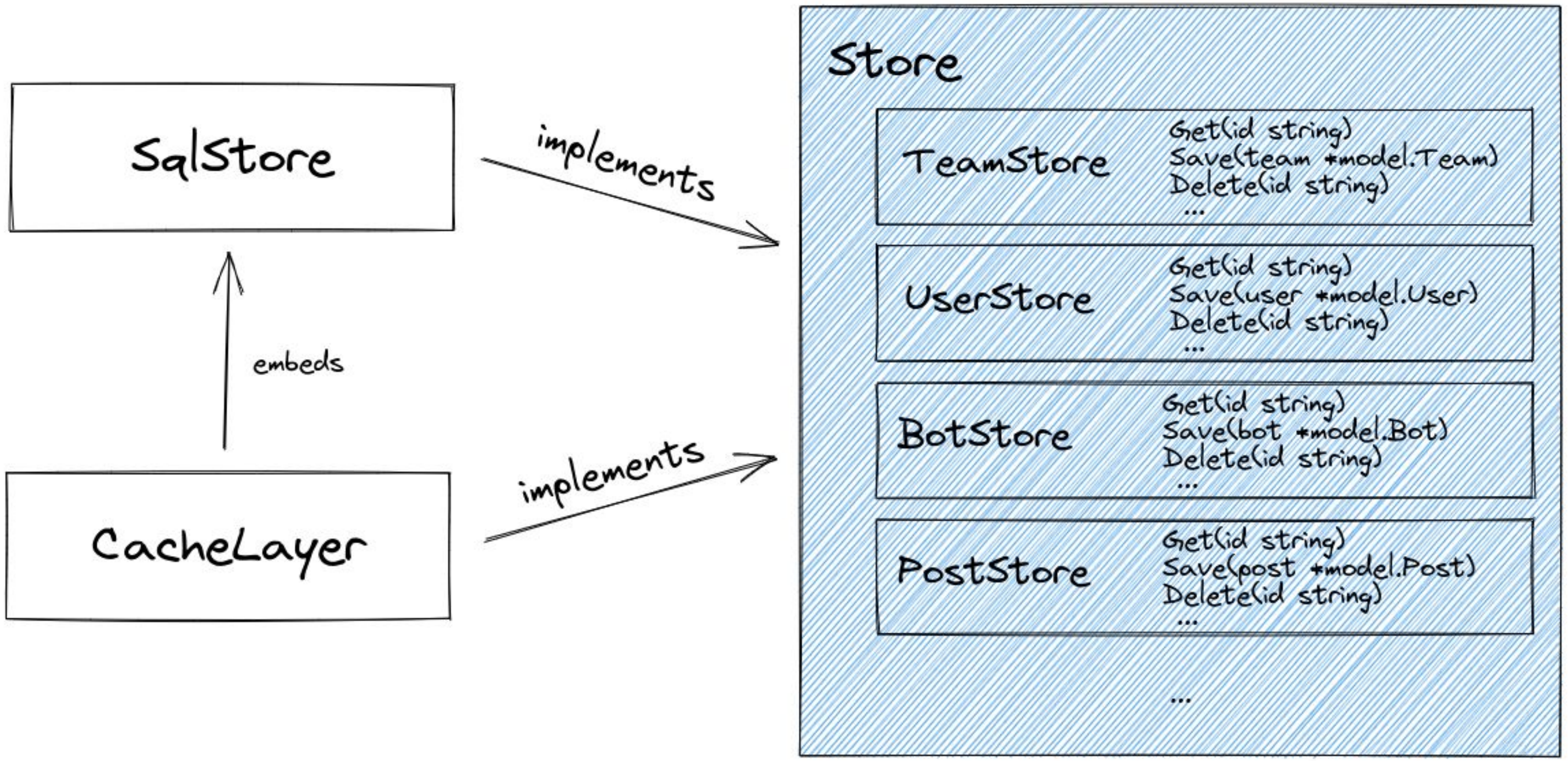
- What went well?
  - The middleware pattern is really common and well known.
  - We had the opportunity to provide extra information without affecting the layers beneath (but we didn’t end use it).

- What didn’t work?
  - Was a bit hard to understand and follow.
  - Was a bit complex to add new caches.

Our current approach

- Use Struct Embedding create layers.
- We rely on the existing interface.
- Create a CachedStore struct embedding the store.
- We overwrite the methods needed for the cache.
- Everything else was transparent.

Our current approach



How we’re doing it

```
type LocalCacheStore struct {
    store.Store
    team LocalCacheTeamStore // &LocalCacheTeamStore{TeamStore: childTeamStore, rootStore: thisStore}
    ...
}

func (s LocalCacheStore) Team() store.TeamStore {
    return s.team
}

type LocalCacheTeamStore struct {
    store.TeamStore
    rootStore                     *LocalCacheStore
}

func (s *LocalCacheTeamStore) Get(ctx context.Context, id string) (*model.Team, error) {
    ... // Check the cache and return on hit
    team, err := s.TeamStore.Get(ctx, id)
    ... // Store in the cache and return
}
```

What went well? What didn’t work?

- What went well?
  - Simple and straightforward solution.
  - Simple to add new caches.
  - Really general reusable approach for other things.

- What didn’t work?
  - Subtle errors can happen if you don’t know struct embedding.
  - The homogenous interface removes some flexibility.

A lot of new possibilities

New possibilities

- Instrumentation.
- Tracing.
- Logging.
- Storage/Query delegation.
- Extra validations.
- Error handling.

Instrumentation (The Timer layer)

- We need to measure the time of each call.
- So we created layer that wrapped each method to store an histogram in Prometheus.
- But each methods is going to be almost identical.
- Generators to the rescue!

Instrumentation (The Timer layer)

```go
func (s *TimerLayerAuditStore) Save(audit *model.Audit) error {
    start := timemodule.Now()

    err := s.AuditStore.Save(audit)

    elapsed := float64(timemodule.Since(start)) / float64(timemodule.Second)
    if s.Root.Metrics != nil {
        success := "false"
        if err == nil {
            success = "true"
        }
        s.Root.Metrics.ObserveStoreMethodDuration("AuditStore.Save", success, elapsed)
    }
    return err
}
```

Instrumentation (Open Tracing)

- We wanted to add OpenTracing support.
- We added another generator to wrap each method.
- We also replicated this approach on other layers of the application (Creating an interface generator in this case).

Instrumentation (Open Tracing)

```go
func (s *OpenTracingLayerBotStore) Get(userID string, includeDeleted bool) (*model.Bot, error) {
    origCtx := s.Root.Store.Context()
    span, newCtx := tracing.StartSpanWithParentByContext(s.Root.Store.Context(), "BotStore.Get")
    s.Root.Store.SetContext(newCtx)
    defer func() {
        s.Root.Store.SetContext(origCtx)
    }()

    defer span.Finish()
    result, err := s.BotStore.Get(userID, includeDeleted)
    if err != nil {
        span.LogFields(spanlog.Error(err))
        ext.Error.Set(span, true)
    }

    return result, err
}
```

Database Retries

- We wanted to support “Serializable” isolation level in our database.
- That implies to retrying transactions that fail often at that isolation level.
- We autogenerated another layer that retries the transaction if the database failure is something that can be retry.

Database Retries

```go
func (s *RetryLayerAuditStore) Get(user_id string, offset int, limit int) (model.Audits, error) {
    tries := 0
    for {
        result, err := s.AuditStore.Get(user_id, offset, limit)
        if err == nil {
            return result, nil
        }
        if !isRepeatableError(err) {
            return result, err
        }
        tries++
        if tries >= 3 {
            err = errors.Wrap(err, "giving up after 3 consecutive repeatable transaction failures")
            return result, err
        }
    }
}
```

Layers generator

- We use AST to read the Store interface and subinterfaces.
- We populate a template with that and render the generated code.
- We post process it with go/format package.

Layers generator (timer layer template)

```
{{range $substoreName, $substore := .SubStores}}
{{range $index, $element := $substore.Methods}}
func (s *{{$.Name}}{{$substoreName}}Store) {{$index}}({{$element.Params | joinParamsWithType}}) {{$element.Results | joinResultsForSignature}} {
    start := timemodule.Now()
    {{if $element.Results | len | eq 0}}
    s.{{$substoreName}}Store.{{$index}}({{$element.Params | joinParams}})
    {{else}}
    {{genResultsVars $element.Results false }} := s.{{$substoreName}}Store.{{$index}}({{$element.Params | joinParams}})
    {{end}}
    elapsed := float64(timemodule.Since(start)) / float64(timemodule.Second)
    if s.Root.Metrics != nil {
        success := "false"
        if {{$element.Results | errorToBoolean}} {
            success = "true"
        }
        s.Root.Metrics.ObserveStoreMethodDuration("{{$substoreName}}Store.{{$index}}", success, elapsed)
    {{ with (genResultsVars $element.Results false ) -}}
    }
    return {{ . }}
    {{- else -}}
    }
    {{- end }}
}
{{end}}
{{end}}
```

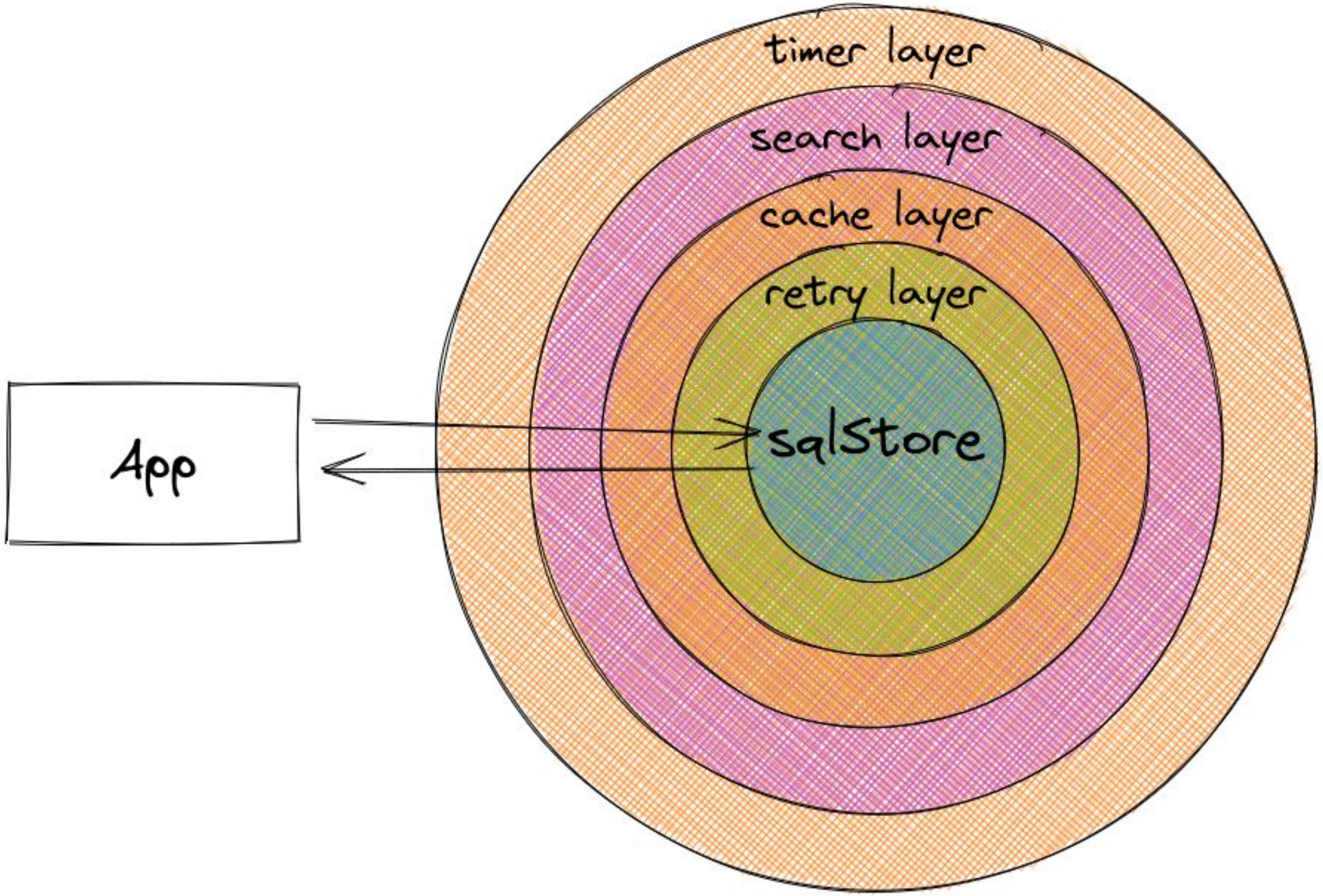
Search wrapper

- We have fulltext search database support but also elasticsearch and bleve.
- We want to make it transparent to the store user.
- We created (writing the code this time) a wrapper of the search methods to call the right search engine (DB, ElasticSearch or Bleve).

Search wrapper

```go
func (s SearchPostStore) Save(post *model.Post) (*model.Post, error) {
    npost, err := s.PostStore.Save(post)

    if err == nil {
        s.indexPost(npost)
    }
    return npost, err
}

func (s SearchPostStore) SearchPostsInTeamForUser(paramsList []*model.SearchParams, userId, teamId string, page, perPage int) (*model.PostSearchResults, error) {
    for _, engine := range s.rootStore.searchEngine.GetActiveEngines() {
        if engine.IsSearchEnabled() {
            results, err := s.searchPostsInTeamForUserByEngine(engine, paramsList, userId, teamId, page, perPage)
            if err != nil {
                mlog.Warn("Encountered error on SearchPostsInTeamForUser.", mlog.String("search_engine", engine.GetName()), mlog.Err(err))
                continue
            }
            mlog.Debug("Using the first available search engine", mlog.String("search_engine", engine.GetName()))
            return results, err
        }
    }

    if *s.rootStore.getConfig().SqlSettings.DisableDatabaseSearch {
        mlog.Debug("Returning empty results for post SearchPostsInTeam as the database search is disabled")
        return &model.PostSearchResults{PostList: model.NewPostList(), Matches: model.PostSearchMatches{}}, nil
    }

    mlog.Debug("Using database search because no other search engine is available")
    return s.PostStore.SearchPostsInTeamForUser(paramsList, userId, teamId, page, perPage)
}
```

The final onion



The final onion

```go
func NewStore(s Server) (store.Store, error) {
    newStore := sqlstore.New(s.Config().SqlSettings, s.Metrics)
    newStore = retrylayer.New(newStore)

    newStore, err2 := localcachelayer.NewLocalCacheLayer(
        newStore,
        s.Metrics,
        s.Cluster,
        s.CacheProvider,
    )
    if err2 != nil {
        return nil, errors.Wrap(err2, "cannot create local cache layer")
    }

    newStore = searchlayer.NewSearchLayer(newStore, s.SearchEngine, s.Config())
    newStore = timerlayer.New(newStore, s.Metrics)

    return newStore, nil
}
```

Drawbacks

- All the layers has to share the same interface.
- Embedding is not Inheritance (Is that bad?)

References

- Mattermost Store code: https://github.com/mattermost/mattermost-server
- Mattermost old layers code: https://github.com/mattermost/mattermost-server/tree/v5.0.0/store
- Talk about struct embedding: https://www.youtube.com/watch?v=-LzYjMzfGDQ
- Talk about code generation: https://www.youtube.com/watch?v=iLk_LnGrst4

Thank you.

Mattermost