Building GraphQL servers in Go

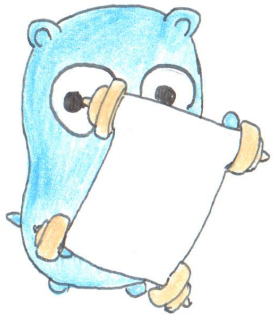
(using gqlgen)

Filip Wojciechowski
23 Mar 2020

Hi, I'm Filip!

- Twitter: @filipcodes (https://twitter.com/filipcodes)
- Blog: https://w11i.me (https://w11i.me)
- Github: https://github.com/fwojciec (https://github.com/fwojciec)

Today's plan



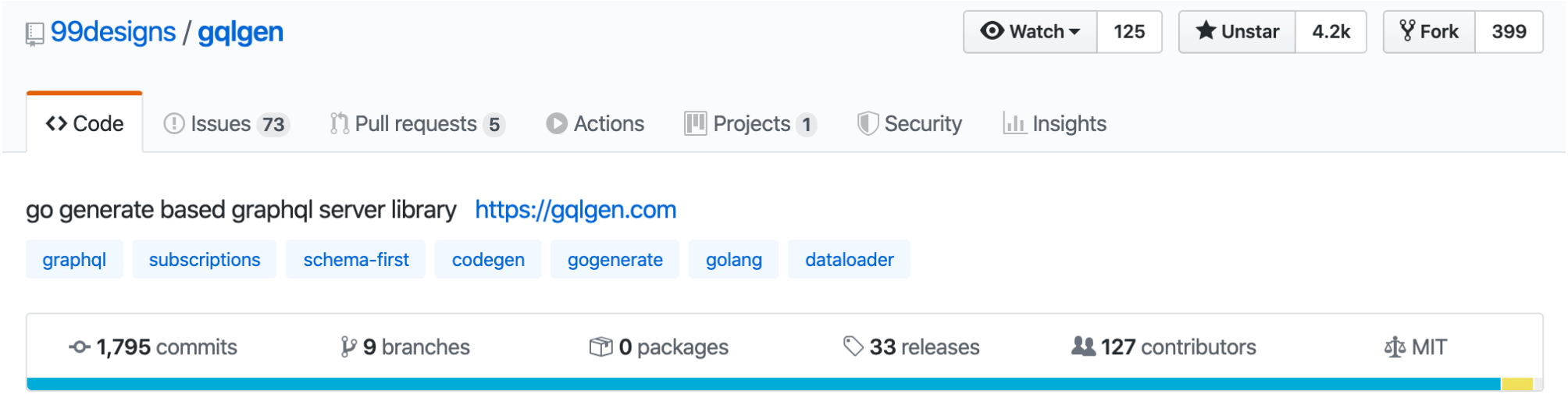
- Why build GraphQL servers (with `gqlgen`)?
- Intro to gqlgen
- Architecture and project organization
- Datalayer considerations
- Authentication
- Authorization
- Testing strategies and examples

Why build GraphQL servers (with `gqlgen`)?

Why GraphQL?

- emerging standard in the JS frontend community (Gatsby, RedwoodJS)
- empowers frontend developers...
- and makes them more productive
- single backend for multiple frontends (web and mobile)
- strongly typed! 😻
- REST is not dead

Why gqlgen?



- popular and actively maintained
- schema-first approach
- type safety (no `map[string]interface{}`)

- feature-full

	gqlgen	gophers	graphql-go	thunder
Kind	schema first	schema first	run time types	struct first
Boilerplate	less	more	more	some
Docs	docs & examples	examples	examples	examples
Query	👍	👍	👍	👍
Mutation	👍	🚧 pr	👍	👍
Subscription	👍	🚧 pr	👍	👍
Type Safety	👍	👍	⛔	👍
Type Binding	👍	👍	⛔	👍
Embedding	👍	⛔	🚧 pr	⛔
Interfaces	👍	👍	👍	⛔ is
Generated Enums	👍	⛔	⛔	⛔
Generated Inputs	👍	⛔	⛔	⛔
Stitching gql	🕐 is	⛔	⛔	⛔
Opentracing	👍	👍	⛔	✂️ pr
Hooks for error logging	👍	⛔	⛔	⛔
Dataloading	👍	👍	👍	⚠️
Concurrency	👍	👍	👍	👍
Custom errors & error.path	👍	⛔ is	⛔	⛔
Query complexity	👍	⛔ is	⛔	⛔

Source: https://gqlgen.com/feature-comparison (https://gqlgen.com/feature-comparison)

- performs well in benchmarks (for what it's worth...)

	Requests/sec
graphql-go	19004.92
graph-gophers	44308.44
thunder	40994.33
gqlgen	**49925.73**

Without graphql (only gin render json output)

	Requests/sec
json without graphql	**55334.07**

Source: https://github.com/appleboy/golang-graphql-benchmark (https://github.com/appleboy/golang-graphql-benchmark)

Intro to gqlgen

Schema-first workflow

1. Create `gqlgen.yml` config file
2. Write a graphql schema
3. Write domain models that correspond to the schema types
4. Generate the server by running `gqlgen`
5. Provide custom implementations of the missing resolvers
6. On schema change: regenerate the server and update the resolvers

[demo]

Architecture and project organization

Example app overview

Repository link: https://github.com/fwojciec/gqlmeetup (https://github.com/fwojciec/gqlmeetup)

Built using:

- gqlgen (https://github.com/99designs/gqlgen)
- dataloaden (https://github.com/vektah/dataloaden)
- sqlx (https://github.com/jmoiron/sqlx)
- the standard library

"Standard Package Layout"

A better approach

The package strategy that I use for my projects involves 4 simple tenets:

1. Root package is for domain types
2. Group subpackages by dependency
3. Use a shared *mock* subpackage
4. *Main* package ties together dependencies

These rules help isolate our packages and define a clear domain language across the entire application. Let's look at how each one of these rules works in practice.

Source: "Standard Package Layout" by Ben Johnson (https://medium.com/@benbjohnson/standard-package-layout-7cdbc8391fc1)

Directory layout

```
├── bcrypt
├── cmd
│   ├── createuser
│   └── server
├── dataloaden
├── gqlgen
├── http
├── mocks
├── postgres
├── scs
├── errors.go       // root dir files
├── gqlgen.yml
├── gqlmeetup.go
├── schema.graphql
├── schema.sql
└── tools.go
```

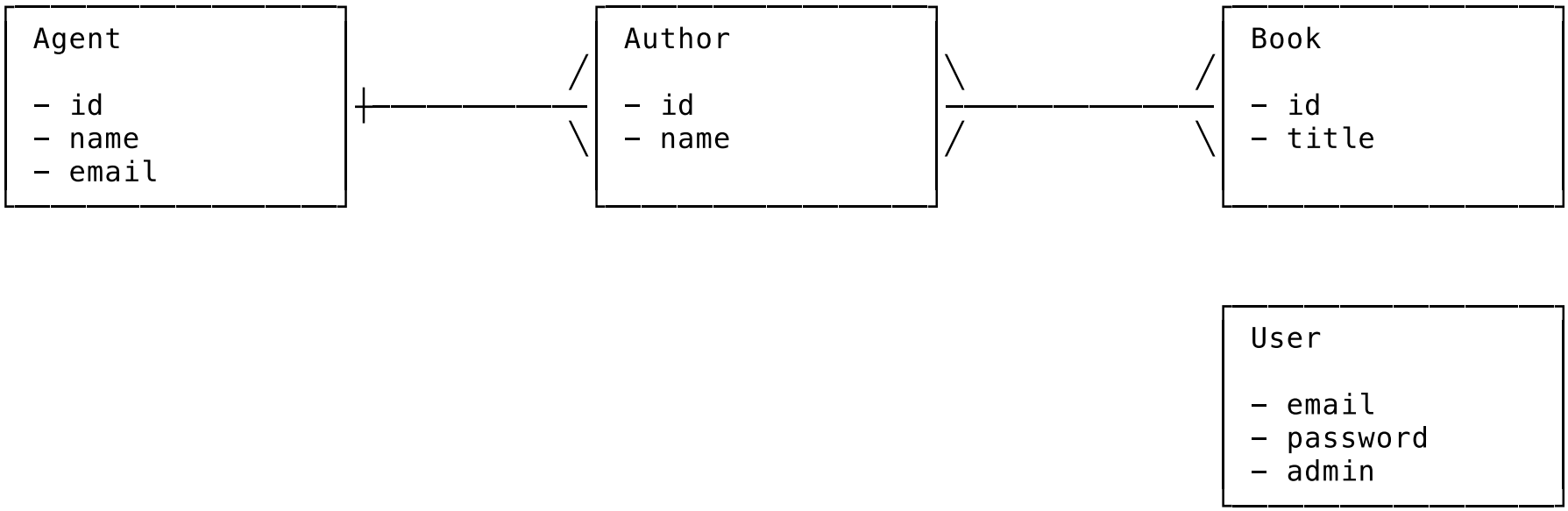
Architecture and project organization: Final thoughts

- prevents the circular dependency problem
- modular design: isolates dependencies, prevents coupling
- makes mocking and testing easy

Datalayer considerations

Data schema

- four types: agent, author, book, and user
- agent has many authors (one-to-many)
- author has many books and book can have many authors (many-to-many)
- user for authentication/authorization purposes

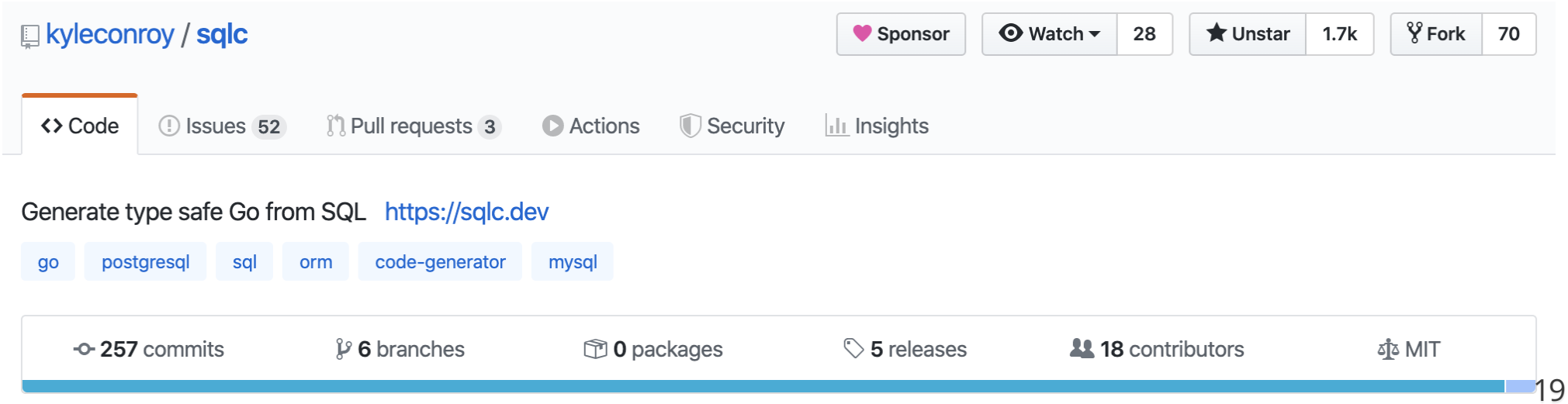


GraphQL, SQL and domain types

```
// schema.graphql
type Author {
  id: ID!
  name: String!
  agent: Agent!
}

// schema.sql
CREATE TABLE IF NOT EXISTS authors (
    id bigserial PRIMARY KEY,
    name text NOT NULL,
    agent_id bigint NOT NULL,
    FOREIGN KEY (agent_id) REFERENCES agents (id) ON DELETE RESTRICT
);

// gqlmeetup.go
type Author struct {
    ID      int64
    Name    string
    AgentID int64
}
```

sqlc

- it's possible to streamline the workflow using sqlc
- very promising, but not a perfect fit yet...
  - values instead of pointers
  - sql.Null types instead of pointers
  - generates its own models



GraphQL ID type

- according to the spec ID can be of any type
- in practice usually string is assumed by tools
- possible to override the default ID type in gqlgen
- a good idea?

How GraphQL resolves queries

- graph of nodes
- node: fields and references to other nodes
- multiple points of entry
- unlimited depth

Example

```
query {
  agents {
    id
    name
    authors {
      id
      name
    }
  }
}
```

```
1. Call the agents method on the queryResolver
2. Call the id method on the agentResolver for every result of (1)
3. Call the name method on the agentResolver for every result of (1)
4. Call the authors method on the agentResolver for every result of (1)
5. Call the id method on the authorResolver for every result of (4)
6. Call the name method on the authorResolver for every result of (4)
```

- resolvers are executed concurrently when possible
- in some cases resolving a field is just looking up a field on a struct
- in other cases, however, it involves a DB or API call
- implications for performance

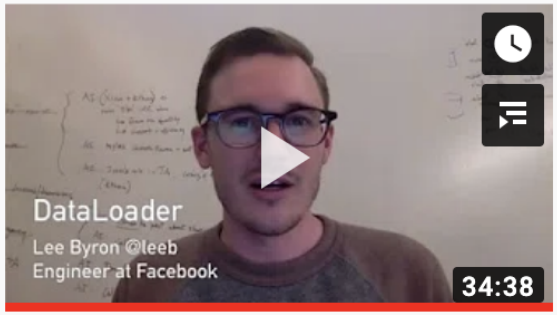
N+1 query problem

- 1 -> The initial query to get the list of agents
- N -> The individual queries to get a list of authors for each agent
- with SQL databases each query uses a DB connection
- luckily there is a solution to this problem: DataLoaders

[demo]

DataLoaders

- "request-scoped method of caching and batching data-fetching queries"
- "request-scoped": a DataLoader instance lives only for the duration of a request
- "caching": each piece of data is fetched only once
- "batching": data is fetched from the datalayer in batches
- great introduction and history of the pattern by Lee Byron



DataLoader – Source code walkthrough

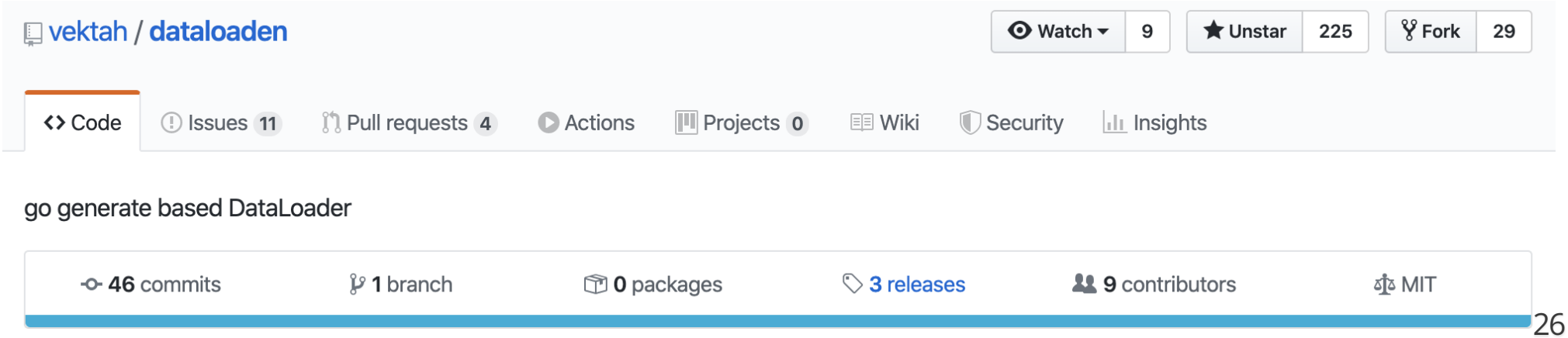
Lee Byron • 26K views • 4 years ago

DataLoader is a utility for batching and caching loaded data in your web service. Learn how it works with a walkthrough of it ...

DataLoader lifecycle

1. a DataLoader instance is initialized by middleware when a request comes in
2. it blocks for a set duration of time and waits to receive arguments for its data query
3. it retrieves the data in one or more batches
4. it returns results corresponding to the respective arguments back to the resolvers
5. it is garbage-collected when the processing of the request finishes

Implementing DataLoaders in Go using dataloaden



Step 1: generate a DataLoader type for a given combination on input and output types

```
// dataloaden/dataloaden.go

package dataloaden

//go:generate dataloaden AuthorSliceLoader int64 []*github.com/fwojciec/gqlmeetup.Author

type loaders struct {
    AuthorsByAgentID *AuthorSliceLoader
    // (...)
}
```

- the input type in this case is `int64` values
- the output value is a slice of `Author` pointers
- `loaders` is the value we will be storing in the context

Step 2: create a concrete implementation...

```
// dataloaden/dataloaden.go

func newAuthorsByAgentID(ctx context.Context, repo gqlmeetup.Repository) *AuthorSliceLoader {
    return NewAuthorSliceLoader(AuthorSliceLoaderConfig{
        MaxBatch: 100,
        Wait:     5 * time.Millisecond,
        Fetch: func(agentIDs []int64) ([][]*gqlmeetup.Author, []error) {
            res, err := repo.AuthorListByAgentIDs(ctx, agentIDs)
            if err != nil {
                return nil, []error{err}
            }
            groupByAgentID := make(map[int64][]*gqlmeetup.Author, len(agentIDs))
            for _, r := range res {
                groupByAgentID[r.AgentID] = append(groupByAgentID[r.AgentID], r)
            }
            result := make([][]*gqlmeetup.Author, len(agentIDs))
            for i, agentID := range agentIDs {
                result[i] = groupByAgentID[agentID]
            }
            return result, nil
        },
    })
}
```

... using a ANY() SQL query

```
// postgres/repository.go

const authorListByAgentIDsQuery = `
SELECT * FROM authors
WHERE agent_id = ANY($1::bigint[]);
`

func (r *Repository) AuthorListByAgentIDs(
    ctx context.Context,
    agentIDs []int64,
) ([]*gqlmeetup.Author, error) {
    res := make([]*gqlmeetup.Author, 0)
    if err := r.DB.SelectContext(
        ctx,
        &res,
        authorListByAgentIDsQuery,
        pq.Array(agentIDs),
    ); err != nil {
        return nil, err
    }
    return res, nil
}
```

Step 3: store the DataLoaders in the request context using HTTP middleware

```
// dataloaden/dataloaden.go

type contextKey string

const key = contextKey("dataloaders")

func (s *DataLoaderService) Initialize(ctx context.Context) context.Context {
    return context.WithValue(ctx, key, &loaders{
        AuthorsByAgentID: newAuthorsByAgentID(ctx, s.Repository),
        // (...)
    })
}

func (s *DataLoaderService) Middleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        ctx := r.Context()
        nextCtx := s.Initialize(ctx)
        r = r.WithContext(nextCtx)
        next.ServeHTTP(w, r)
    })
}
```

Provide a method to retrieve the DataLoader from the context (for use in the resolvers)

```
// dataloaden/dataloaden.go

func (s *DataLoaderService) AuthorListByAgentID(
    ctx context.Context,
    agentID int64,
) ([]*gqlmeetup.Author, error) {
    l, err := s.retrieve(ctx)
    if err != nil {
        return nil, err
    }
    return l.AuthorsByAgentID.Load(agentID)
}

func (s *DataLoaderService) retrieve(ctx context.Context) (*loaders, error) {
    l, ok := ctx.Value(key).(*loaders)
    if !ok {
        return nil, gqlmeetup.ErrInvalid
    }
    return l, nil
}
```

DataLoaders: Final thoughts

- they come at a cost so it might be worth it to consider alternative implementations
- fine-tune performance using the batch/wait variables
- data-source agnostic, so they could be used to talk to APIs rather than DBs, for example

Authentication

- authentication: determining the identity of the user making the request
- many ways to approach this:
  - OAuth2 / Login Form / Both?
  - Login Form implemnented using HTTP handler or as a GraphQL mutation?
  - JWT Tokens?
  - Cookie-based sessions?

Example implementation

- a login mutation which saves the session in the HttpOnly cookie
- caveat - this approach is incompatible with subscriptions/websockets

(https://github.com/alexedwards/scs/issues/82)

It's tricky to implement cookie-based authentication in gqlgen

- no access to the `http.ResponseWriter` and `*http.Request` in the resolvers
- as such no easy way to set/access cookies

Question: Setting cookies #567

Closed DouglasHoang opened this issue on Mar 1, 2019 · 24 comments

DouglasHoang commented on Mar 1, 2019

I was looking at #98 and was wondering if I should set cookies through a resolver or a handler.

mathewbyrne commented on Mar 3, 2019 Contributor

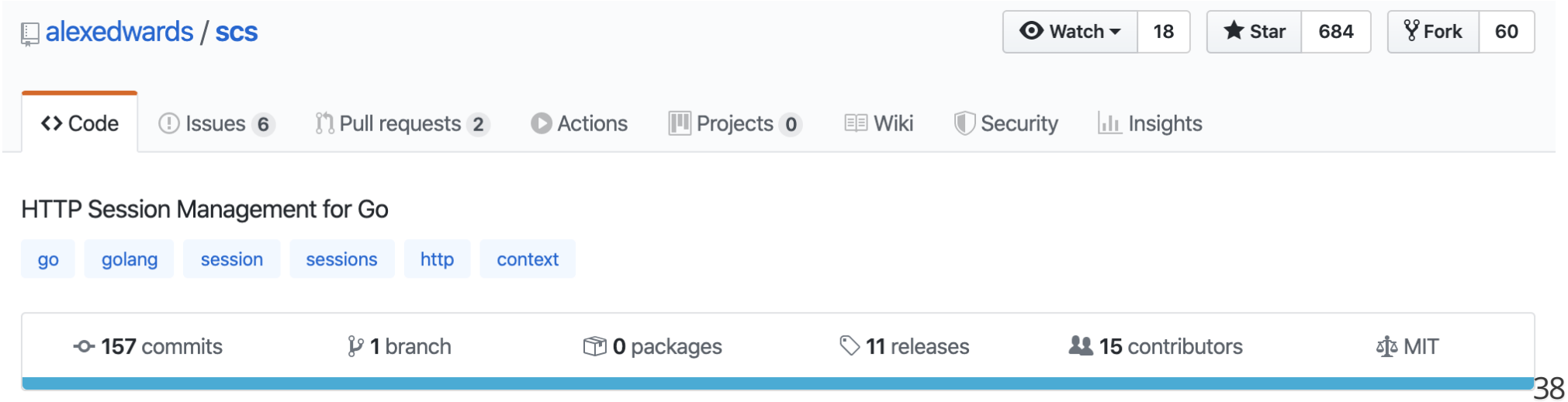
If you want to access cookie information from a resolver, then follow the advice given in #98 and use a mmiddleware to inject a value into context.

Source: https://github.com/99designs/gqlgen/issues/567 (https://github.com/99designs/gqlgen/issues/567)

Step 1: Define SessionService interface

```
// gqlmeetup.go

type SessionService interface {
    Login(ctx context.Context, user *User) error
    Logout(ctx context.Context) error
    GetUser(ctx context.Context) *User
    Middleware(http.Handler) http.Handler
}
```

Step 2: Implement the functionality using the scs library...



.. Login method ...

```
// scs/scs.go

type SessionService struct {
    sm *scs.SessionManager
}

func (s *SessionService) Login(ctx context.Context, user *gqlmeetup.User) error {
    if err := s.sm.RenewToken(ctx); err != nil {
        return err
    }
    // marshal to JSON, without pwd hash
    b, err := s.marshalUser(user)
    if err != nil {
        return err
    }
    s.sm.Put(ctx, "user", b)
    return nil
}
```

.. Logout method ...

```
// scs/scs.go

func (s *SessionService) Logout(ctx context.Context) error {
    return s.sm.Destroy(ctx)
}
```

.. GetUser method ...

```
// scs/scs.go

func (s *SessionService) GetUser(ctx context.Context) *gqlmeetup.User {
    b := s.sm.GetBytes(ctx, "user")
    if b == nil {
        return nil
    }
    // unmarshal from JSON
    user, err := s.unmarshalUser(b)
    if err != nil {
        return nil
    }
    return user
}
```

.. and provide a HTTP Middleware to wrap the GraphQL handler

```
// scs/scs.go

func (s *SessionService) Middleware(next http.Handler) http.Handler {
    return s.sm.LoadAndSave(next)
}
```

Authenitcation: Final thoughts

- if you end choosing JWT tokens: don't store them in local storage
- use an established OpenID Connect Service like (Auth0, AWS Cognito, etc.)

Authorization

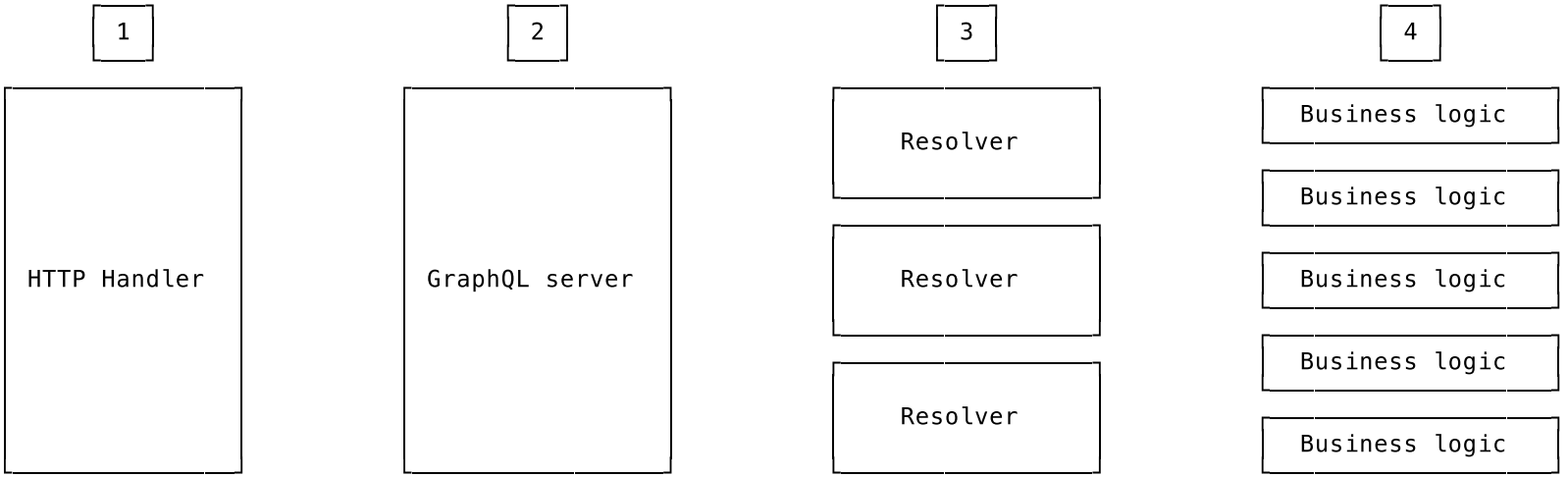
- authorization: should this user be able to do this
- a very complex topic, for a deeper dive see:

GraphQL Berlin Meetup #15: Learnings from implementing authentication and authorization with GraphQL

Prisma • 776 views • 5 months ago

Key learnings from implementing authentication and **authorization** using **GraphQL** — Christoph Hartmann Using **GraphQL** in ...

Where to put authorization?



Source: Christoph Hartmann (https://www.youtube.com/watch?v=pF6SH7Tm6Pc)

1. HTTP server / implemented using HTTP middleware
2. GraphQL server / implemented using directives
3. Resolvers / custom logic in the resolvers
4. Authorization deferred to some outside logic called by the resolvers

Schema directives

Field-level authorization implemented using the GraphQL schema:

```
// schema.graphql

directive @hasRole(role: Role!) on FIELD_DEFINITION

enum Role {
  ADMIN
  USER
}

type Query {
  me: User! @hasRole(role: USER)
  // (...)
}

type Mutation {
  agentCreate(data: AgentInput!): Agent! @hasRole(role: ADMIN)
  authorCreate(data: AuthorInput!): Author! @hasRole(role: USER)
  // (...)
}
```

Implementation of the hasRole directive

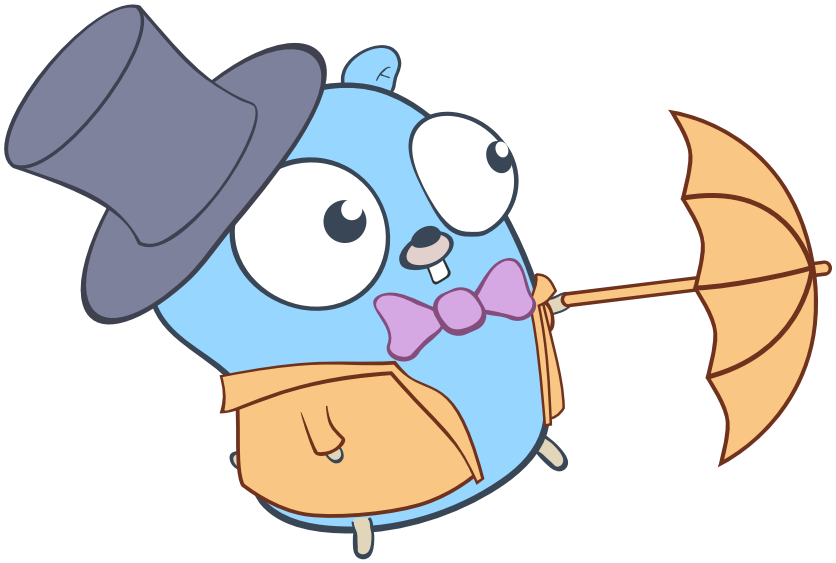
```
// gqlgen/gqlgen.go

type Directive func(context.Context, interface{}, graphql.Resolver, Role) (interface{}, error)

func HasRole(session gqlmeetup.SessionService) Directive {
    return func(
        ctx context.Context,
        obj interface{},
        next graphql.Resolver,
        role Role,
    ) (interface{}, error) {
        user := session.GetUser(ctx)
        if user == nil {
            return nil, gqlmeetup.ErrUnauthorized
        }
        if role == RoleAdmin && !user.Admin {
            return nil, gqlmeetup.ErrUnauthorized
        }
        return next(ctx)
    }
}
```

Resolver-level authorization

- use the SessionService to retrieve the user
- grant/restrict access based on who the user is
- run different queries based on who the user is
- etc.

Authorization: Final thoughts

- authorization code should be fast (similar considerations as in the case of DataLoaders)
- consider authorization when designing the schema
- fine-grained authorization for large GraphQL APIs is a difficult engineering challenge



Testing strategies and examples

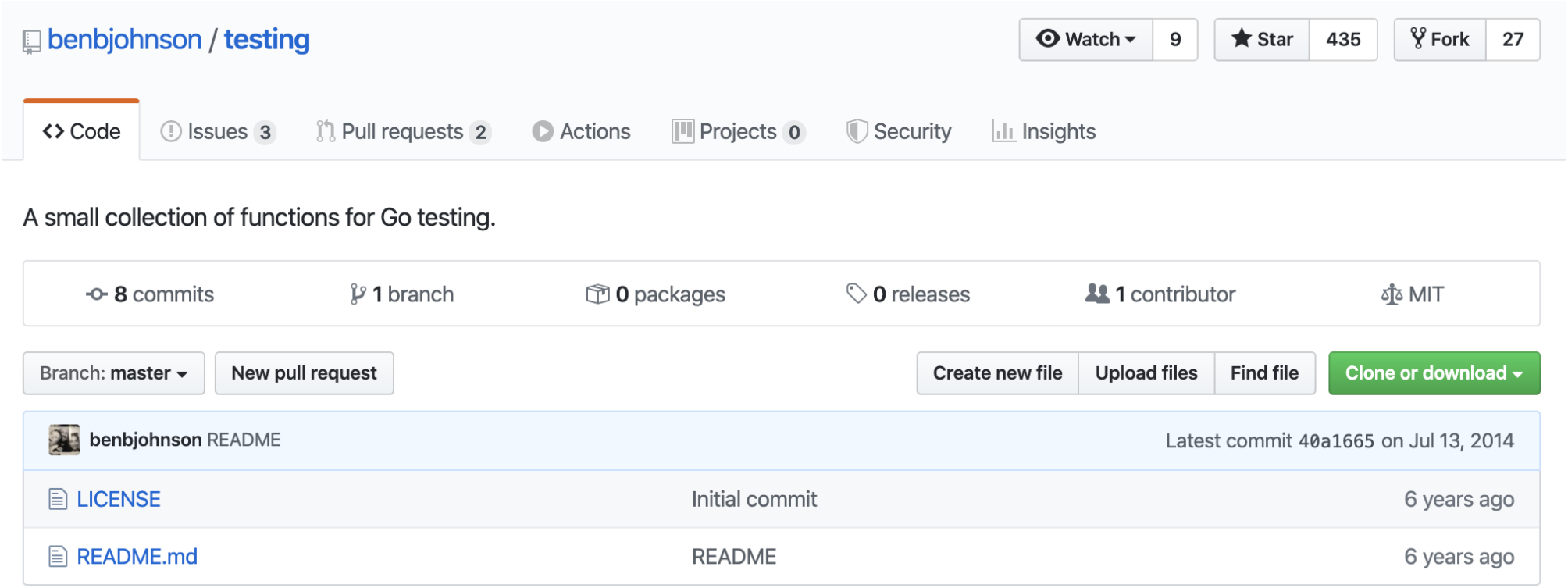
"Standard Package Layout" makes testing manageable

- all dependencies are isolated in separate subpackages
- their functionality is defined by interfaces
- which makes it easy to use mocks

t.Parallel() everything

Use this approach, for example, to run DB tests in parallel: https://conroy.org/per-test-database-isolation-in-postgres (https://conroy.org/per-test-database-isolation-in-postgres)

ok, equals, assert



https://github.com/benbjohnson/testing (https://github.com/benbjohnson/testing)

DB Testing: don't mock the database

```
// postgres/postgres_test.go

testAgent1 = gqlmeetup.Agent{
    ID:    1,
    Name:  "Test Agent 1",
    Email: "agent1@test.com",
}

// postgres/repository_test.go

func TestAgentGetByID(t *testing.T) {
    t.Parallel()
    pgt.Runner(t, []string{"agents"}, func(t *testing.T, sdb *sqlx.DB) {
        repo := &postgres.Repository{DB: sdb}
        res, err := repo.AgentGetByID(context.Background(), 1)
        ok(t, err)
        equals(t, &testAgent1, res)
    })
}
```

pgt: https://github.com/fwojciec/pgtester (https://github.com/fwojciec/pgtester)

DB Testing: table-driven tests

```
func TestAuthorListByAgentIDs(t *testing.T) {
    t.Parallel()
    tests := []struct {
        agentIDs []int64
        exp      []*gqlmeetup.Author
    }{
        {[]int64{1}, []*gqlmeetup.Author{&testAuthor1}},
        {[]int64{1, 2}, []*gqlmeetup.Author{&testAuthor1, &testAuthor2}},
    }
    for _, tc := range tests {
        tc := tc
        t.Run(fmt.Sprintf("%v", tc.agentIDs), func(t *testing.T) {
            t.Parallel()
            pgt.Runner(t, []string{"authors"}, func(t *testing.T, sdb *sqlx.DB) {
                repo := &postgres.Repository{DB: sdb}
                res, err := repo.AuthorListByAgentIDs(context.Background(), tc.agentIDs)
                ok(t, err)
                equals(t, tc.exp, res)
            })
        })
    }
}
```

DB Testing: create/update/delete

```
func TestAgentCreate(t *testing.T) {
    t.Parallel()
    pgt.Runner(t, []string{"agents"}, func(t *testing.T, sdb *sqlx.DB) {
        ctx := context.Background()
        repo := &postgres.Repository{DB: sdb}
        t.Run("create", func(t *testing.T) {
            res, err := repo.AgentCreate(ctx, testAgentCreate)
            ok(t, err)
            equals(t, &testAgentCreate, res)
            t.Run("assert agent was created", func(t *testing.T) {
                t.Parallel()
                res, _ := repo.AgentGetByID(ctx, testAgentCreate.ID)
                equals(t, &testAgentCreate, res)
            })
        })
    })
}
```

Subpackages

- test the individual subpackages
- use mocks elsewhere, where these subpackages are used
- the DB tests discussed previously follow this pattern
- use something like moq (https://github.com/matryer/moq) to automatically generate mocks from interfaces

Resolver testing

This is how the resolver is defined:

```
// gqlgen/resolver.go

type Resolver struct {
    Repository  gqlmeetup.Repository
    DataLoaders gqlmeetup.DataLoaderService
    Password    gqlmeetup.PasswordService
    Session     gqlmeetup.SessionService
}
```

Testing individual resolver methods typically involves:

1. using mock implementations of the services used by a particular method
2. asserting that these services were called correctly

Resolver testing: Example 1

```
// gqlgen/resolvers_test.go

func TestMutationResolverAuthorCreate(t *testing.T) {
    t.Parallel()
    repoMock := &mocks.RepositoryMock{
        AuthorCreateFunc: func(c context.Context, a gqlmeetup.Author) (*gqlmeetup.Author, error) {
            return nil, nil
        },
    }
    r := &gqlgen.Resolver{Repository: repoMock}
    _, _ = r.Mutation().AuthorCreate(context.Background(), gqlgen.AuthorInput{
        Name:    "Test Name",
        AgentID: "12",
    })
    equals(t, repoMock.AuthorCreateCalls()[0].Data, gqlmeetup.Author{
        Name:    "Test Name",
        AgentID: 12,
    })
}
```

Resolver testing: Example 2

```
// glqgen/resolvers_test.go

func TestQueryResolverBooks(t *testing.T) {
    t.Parallel()
    repoMock := &mocks.RepositoryMock{
        BookListFunc: func(ctx context.Context, limit, offset *int) ([]*gqlmeetup.Book, error) {
            return nil, nil
        },
    }
    r := &gqlgen.Resolver{Repository: repoMock}
    _, err := r.Query().Books(context.Background(), intToPtr(2), nil)
    ok(t, err)
    equals(t, repoMock.BookListCalls()[0].Limit, intToPtr(2))
    equals(t, repoMock.BookListCalls()[0].Offset, (*int)(nil))
}
```

DataLoader testing

- to test dataloaders you should test how they behave under conditions of concurrency
- use the t.Parallel() to simulate concurrent processing of a request

DataLoader testing: Example

```
func TestAuthorListByAgentID(t *testing.T) {
	t.Parallel()
	mock := &mocks.RepositoryMock{
		AuthorListByAgentIDsFunc: func(ctx context.Context, agentIDs []int64) ([]*gqlmeetup.Author, error) {
			return []*gqlmeetup.Author{&testAuthor1, &testAuthor2, &testAuthor3}, nil
		},
	}
	dls := &dataloaden.DataLoaderService{Repository: mock}
	ctx := dls.Initialize(context.Background())
	t.Run("concurrent requests", func(t *testing.T) {
		tests := []struct {
			agentID int64
			exp     []*gqlmeetup.Author
		}{
			{2, []*gqlmeetup.Author{&testAuthor2, &testAuthor3}},
			{1, []*gqlmeetup.Author{&testAuthor1}},
		}
		for _, tc := range tests {
			tc := tc
			t.Run(fmt.Sprintf("Agent ID: %d", tc.agentID), func(t *testing.T) {
				t.Parallel()
				res, err := dls.AuthorListByAgentID(ctx, tc.agentID)
				ok(t, err)
				equals(t, tc.exp, res)
			})
		}
	})
	equals(t, 1, len(mock.AuthorListByAgentIDsCalls()))
}
```

Thank you

Filip Wojciechowski
23 Mar 2020
fwojciec@gmail.com (mailto:fwojciec@gmail.com)
@filipcodes (http://twitter.com/filipcodes)