Best Practices

JWKS Services

Anthony Lee

Code Review

Z's code

```
func main() {

    const certPEM = `
-----BEGIN CERTIFICATE-----
MIIIojCCBoqgAwIBAgITQwAAVbBhhkMoUyGs4AAAAABVsDANBgkqhkiG9w0BAQsFADBxMRMwEQYKCZImiZPyLGQBGRYDY29tMRYwFAYK
-----END CERTIFICATE-----`

    block, _ := pem.Decode([]byte(certPEM))
    var cert *x509.Certificate
    cert, err := x509.ParseCertificate(block.Bytes)
    if err != nil {
        logger.Fatal("There was a problem parsing cert", zap.Any("error", err))
    }
    rsaPublicKey := cert.PublicKey.(*rsa.PublicKey)
    fmt.Println(PrintStruct(*rsaPublicKey, true))

    //id := uuid.New()

    jwksSet := &jose.JSONWebKeySet{
        Keys: []jose.JSONWebKey{
            {
                Algorithm:    "RS256",
                Key:          rsaPublicKey,
                Use:          "sig",
                KeyID:        "S.STG.01C3GJFQYR8E97",
                Certificates: []*x509.Certificate{cert},
            },
```

```
    },
}

HandleJwks := func(w http.ResponseWriter, r *http.Request) {
    fmt.Println("r = " + PrintStruct(*r, true))
    json.NewEncoder(w).Encode(jwksSet)
}

// Initialize web http listener
r := mux.NewRouter()
r.HandleFunc("/openid/connect/jwks.json", http.HandlerFunc(HandleJwks)).Methods("GET")

// Configure http listener to use the mux router as the handler for all requests.
http.Handle("/", r)

serverCert, err := filepath.Abs("./cert.pem")
if err != nil {
    logger.Fatal("There was a problem finding server cert", zap.Any("error", err))
}

serverKey, err := filepath.Abs("./key.pem")
if err != nil {
    logger.Fatal("There was a problem finding server key", zap.Any("error", err))
}

err = http.ListenAndServeTLS(":8999", serverCert, serverKey, nil)
//err = http.ListenAndServe(":8999", nil)

if err != nil {
    logger.Fatal("There was a problem setting up the http listener", zap.Any("error", err))
```

```
    }

}
```

On to the review

internal/z/z.go (internal/z/z.go)

Review:

- Great job fulfilling functionality before introducing complexity.
- From here, separate concerns and develop in "layers."
- Errors should be handled or deferred to client code.
- Avoiding the "Simpsons already did it."
- *Generally* avoid critiquing performance unless its critical software.
- Hot take: the only software of value is running in production.

Project Structure

Root Directory

```
|-cmd - packages that are compilable bins
|-pkg - source code that is out in the wild
|-internal - "magic" dir that cannot be used outside of this Project
|
|-go.mod - dependency resolution
|-go.sum - checksum of dependencies
|
| ** Optional junk **
|-Makefile - a common interface
```

Examples:

- Athens (https://github.com/gomods/athens)
- Imunicorn (https://git.target.com/ae-authentication/go-imunicorn)

Our Application, Reified

```
|-cmd
| |-juke - HTTP binary
|
|-pkg
| |-juke - JWKS manager
| |-jwks - JWKS parser/generator
|
|-internal
| |-http - HTTP server
| |-resolver - Dependency resolver
... Other stuff
```

Light Domain Driven Design.

Perhaps a bit overkill, but easy to extend.

Code Structure

pkg/jwks - Domain

Simple interfaces, deep functionality:

- Keep signatures easy to reason about.
- Shield consumers from implementation details.

```
// JWKS is the default cert to JWKS generator.
// UUIDs are used as `kid`s.
func JWKS(pemCerts ...[]byte) (*jose.JSONWebKeySet, error) {
    return basicJWKer.JWKS(pemCerts...)
}
```

The Garden vs...

...The Weeds

```
// JWKS creates a JSON Web Key Set from a number of PEM certificates.
// See: https://tools.ietf.org/html/rfc7517
func (j *JWKer) JWKS(pemCerts ...[]byte) (*jose.JSONWebKeySet, error) {
    if j.ID == nil {
        j.ID = UUID()
    }
    jwks := &jose.JSONWebKeySet{
        Keys: []jose.JSONWebKey{},
    }
    for _, pc := range pemCerts {
        pk, cert, err := ParsePEM(pc)
        if err != nil {
            return nil, errors.Wrap(err, "public key parsing failed")
        }
        jwk := jose.JSONWebKey{
            Algorithm:    "RS356",
            Key:          pk,
            Use:          "sig",
            KeyID:        j.ID(),
            Certificates: []*x509.Certificate{cert},
        }
        jwks.Keys = append(jwks.Keys, jwk)
    }

    return jwks, nil
}
```

pkg/juke - Service

Juke manages the storage, retrieval and validation of JWKS.

```
// JWKS returns a snapshot of the current JWKS.
func (s *Server) JWKS() (*jose.JSONWebKeySet, error) {
    // Check validity of current JWKS and update if necessary.
    if !VerifyJWKS(s.jwks) {
        if err := s.SetJWKS(); err != nil {
            return nil, err
        }
    }
    return s.jwks, nil
}
```

Signatures remain similar throughout the app stack.

internal/http - Encapsulation

```
// Server to handle HTTP interactions with Juke.
type Server struct {
    Router chi.Router
    JWKSer JWKSer
    Logger *zap.SugaredLogger
    Config Config
}
```

```
// NewServer instantiates a server.
func NewServer(jwkser JWKSer, router chi.Router, logger *zap.SugaredLogger, cfg Config) *Server {
    s := &Server{
        JWKSer: jwkser,
        Router: router,
        Logger: logger,
        Config: cfg,
    }
    s.Routes()
    return s
}
```

internal/http - Defining dependencies

```
type JWKSer interface {
    JWKS() (*jose.JSONWebKeySet, error)
}
```

Benfits of abstracting dependencies:

- Create mocks for testing.
- Keep implementation details out of mind.
- Swap out implementations.

Interfaces are defined by consumers!

internal/http

Minimal application logic.

A single handler retrieves a JWKS and responds with it.

```
// GetJWKS returns a raw JSON Web Key Set.
func (s *Server) GetJWKS() http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        enc := json.NewEncoder(w)
        enc.SetIndent("", "    ")
        jwks, err := s.JWKSer.JWKS()
        if err != nil {
            http.Error(w, "jwks retrieval failed", http.StatusInternalServerError)
            return
        }
        if err := enc.Encode(jwks); err != nil {
            s.Logger.Errorw("jwks encoding failed", "err", err)
            http.Error(w, "jwks encoding failed", http.StatusInternalServerError)
            return
        }
    }
}
```

cmd

Resolve dependencies and start application.

```
func main() {
    path := "configs/dev.json"
    bb, err := ioutil.ReadFile(path)
    if err != nil {
        log.Fatalf("could not load config at %s", path)
    }
    var cfg resolver.Config
    if err := json.Unmarshal(bb, &cfg); err != nil {
        log.Fatalf("unable to unmarshal config: %s", err)
    }
    r := resolver.NewResolver(cfg)

    s := r.ResolveHTTP()
    r.Logger.Infof("serving on %s", ":"+r.Config.HTTP.Port)
    if err := http.ListenAndServe(":"+r.Config.HTTP.Port, s.Router); err != nil {
        r.Logger.Fatal(err)
    }
}
```

Run

TODO:

- Eliminate use of `square/jose` types in API.

Future:

- Error Handling
- Configuration Management
- Dependency Resolution
- Unit Testing
- Building an HTTP Server
- Suggested Third Party Packages

Helpful Resources:

- https://github.com/golang/go/wiki/CodeReviewComments
- https://golang.org/doc/effective_go.html

Thank you

Anthony Lee