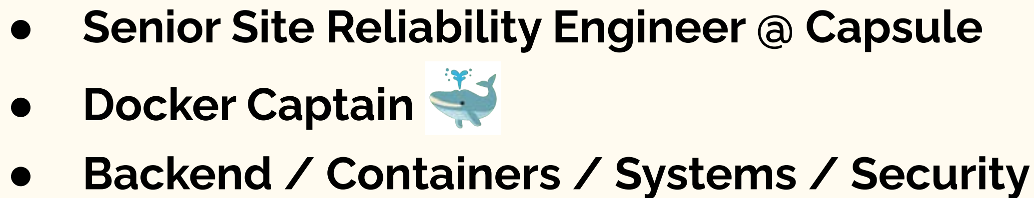
Optimizing Performance and Security: Crafting Robust API with GO

Prabesh Thapa

$ whoami



- Senior Site Reliability Engineer @ Capsule
- Docker Captain 🐳
- Backend / Containers / Systems / Security

Things we want our API to be

- Secure
- Fast
- Robust

Secure

Authentication / authorization

- Token based authentication
    - Know the purpose of your token (access_token, id_token)
    - Signed JWT: algo, strong secret,
        - PASETO
    - OAuth 2.0 with appropriate flow
    - Using access and refresh token (YMMV)
- Proper CORS configuration
- Cookie
    - Secure and HttpOnly
    - Encrypted (HMAC with salt)
    - SameSite Policy (strict, lax)
- Least privilege principle

Authentication / authorization

- Re-authenticate user before critical change (email change, password reset, profile update) even if they are already logged in
  - Check IsAuthenticated
  - Check IsAuthorized

```
func IsAuthenticated(authPayload *token.Payload) bool {
// Check if user is authenticated before any critical operations
}

func IsAuthorised(userID string, store *repository.Store) bool {
// Check ownership and authorization
}
```

Validate inputs

- Validate and sanitize all inputs received from clients to prevent from injection attacks like SQL Injection, NoSQL Injection and XSS
- Leverage battle tested libraries
  - Validator: github.com/go-playground/validator/v10
  - Bluemonday: https://github.com/microcosm-cc/bluemonday

```
type User struct {
    Username        string      `json:"username" validate:"required,min=3,max=20"`
    Email           string      `json:"email" validate:"required,email"`
    Password        string      `json:"password" validate:"gte=18"`
}
```

log user activity (Audit trails)

- Log user activity
- First,
  - Define necessary and sufficient permissions
- Then
  - Define baseline activity, alert anything that is off the baseline
  - User who is a customer should be trying to log into admin portal

Use appropriate response

- Use appropriate response message so that attacker cannot run enumeration attacks

```go
if !userExists {
	w.WriteHeader(http.StatusNotFound)
	response := ErrorResponse{Message: "User does not exist"}
	json.NewEncoder(w).Encode(response)
	return
}
```

```go
if !userExists {
	w.WriteHeader(http.StatusBadRequest)
	response := ErrorResponse{Message: "Could not process request"}
	json.NewEncoder(w).Encode(response)
	return
}
```

Password management

- Use one-way hash while storing password (bcrypt)
- Do not allow user to use consecutive chars from their email or username in password
- Do not allow password from most common list
  - https://github.com/danielmiessler/SecLists/blob/master/Passwords/Common-Credentials/10k-most-common.txt
  -
- Other
  - Check for complexity, length,

Password management

```go
if validators.NotIn(password.CommonPasswords, request.Password) {
	errorsListslice := []model.Error{
		{
			Code:    int(errors.ErrPasswordTooCommon),
			Message: errors.ErrPasswordTooCommon.Message(),
		},
	}

	log.Error().
		Str("request-id", requestid.Get(ctx)).
		Msg("password too common")

	result := model.Result{}
	response := model.HTTPResponse{
		Errors:     “Password too common”,
		Result:     &result,
		StatusCode: http.StatusBadRequest,
	}

	ctx.JSON(http.StatusBadRequest, response)
	return
}
```

Use config file vs env vars

- Problem: Env vars can be accessed from any part of the program
- Passing values using config file
  - Singleton
  - Use appropriate filesystem permission to protect file

```go
type Server struct {
	conf *configs.configuration
	logger *zerolog.Logger
	dbClient *mongo.Client
	redisClient *redis.Client
}
```

Mask sensitive data

- Make sure you are masking any kind of information which might reveal user info for any data in transit or at rest
- Using techniques like:
  - Data masking
  - Encryption
  - Tokenization

```
func MaskData(data string) string {
	masked := ""
	for range data {
		masked += "*"
	}
	return masked
}
```

```
func Tokenize Data(data string, secretKey string) (string, error) {
	token := jwt.NewWithClaims(jwt.SigningMethodHS256, jwt.MapClaims{
		"data": data,
	})
	tokenString, err := token.SignedString([]byte(secretKey))
	if err != nil {
		return "", err
	}
	return tokenString, nil
}
```

Implement rate limiting

- Implement rate limiting to prevent abuse and protection against DOS attacks.
- Use appropriate algorithm: FWC, LB, TB
- Things to consider
  - Celebrity problem - plan for burst
  - Communicate rate limits as part of the response
  - Implement endpoint specific rate limit policy

```
type RateLimiter struct {
	mu           sync.Mutex
	windowStart  time.Time
	requestCount int
	limit        int
	interval     time.Duration
}
```

Implement rate limiting

```
{
  // other data,

  "meta": {
    "rate_limit": {
      "limit": 1000,
      "remaining": 750,
      "reset_time": "2024-04-18T12:00:00Z"
    }
  }
}
```

Secure communication

- Ensure communication channel between use and server is encrypted using TLS
- Make sure api server supports TLS natively

```
switch conf.Server.Protocol {
case "https":
	err = srv.RunTLS()
case "http":
	err = srv.Run()
default:
	return fmt.Errorf("server type %s not supported", conf.Server.Protocol)
}
```

Patch your dependencies

- Rolling out your own is not always possible, we need to use external libraries
- Stay on top your patch management
- Tools
  - snyk
  - Github dependabot
  - CISM tools like Wiz

Fast

Use optimized database queries

```
select * from table where user.email = "demo@gmail.com"
```

```
select required_column from table where user.email = "demo@gmail.com"
```

Use optimized database queries

- Use optimized database queries
- Implement indexing (explicit, multi-column, partial)
- Minimize redundant queries

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50),
    email VARCHAR(100)
);
```

```
CREATE INDEX idx_username ON users (username);
```

Implement caching

- Implement caching to store frequently accessed data
  - In-memory or external KV storage (redis)
- Allow clients to cache response using e-tags and cache-control headers
- Leverage CDN is possible / available for static contents (images, binaries etc)

```
w.Header().Set("Cache-Control", "max-age=3600")
```

Pagination

- Helps speed up API by reducing the amount of data to be transferred.
- Reduces data transfer
- Faster query execution
- Reduces latency
- Cursor vs offset based

```
findOptions.SetSkip((int64(page) - 1) * int64(pageSize))
findOptions.SetLimit(int64(pageSize))

// define filter options
total, err := orderCollection.CountDocuments(ctx, filter)
// handle error

cursor, err := orderCollection.Find(ctx, filter, findOptions)
// handle error
```

Use connection pooling

- Improves performance by reusing existing connections rather than establishing new connections for each request.
- Use when there are volume of requests to database and requires new connections to be created each time
- But: If server can handle concurrent connections easily, not worth the complexity and overhead.

```
httpClient := &http.Client{
     Timeout: 10 * time.Second,
     Transport: &http.Transport{
         MaxIdleConns:        100,
         MaxIdleConnsPerHost: 10,
     },
 }
```

Break into smaller service

- Breaking service into microservice helps improve performance by making app
  - Independently scalable
  - Maintainable
  - Resilient
  - Future proof

Break into smaller service

- Micro-service is a double edge-sword.
- Do not go about breaking your app into microservice just because you can.
    - Microservice is only warranted if each domain of business is handled by different teams (payment processing, data, fulfillment etc)

Robust

Fail early, Fail fast

- Early failure helps prevent cascading failure
- Helps quickly identify issues
- Improves stability and robustness

Fail early, Fail fast

```go
func main() {
	if err := runServer(); err != nil {
		log.Fatalf("Error starting server: %v", err)
	}
}

func runServer() error {
	configPath, err := getConfigPath()
	if err != nil {
		return fmt.Errorf("could not get config path: %v", err)
	}

	conf, err := configs.ConfigSetup(configPath, configName, configType)
	if err != nil {
		return fmt.Errorf("could not complete config setup: %v", err)
	}

	srv, err := server.NewServer(conf)
	if err != nil {
		return fmt.Errorf("could not create a new server: %v", err)
	}

	switch conf.Server.Protocol {
	case "https":
		err = srv.RunTLS()
	default:
		return fmt.Errorf("server type %s not supported", conf.Server.Protocol)
	}

	if err != nil {
		return fmt.Errorf("server is not running: %v", err)
	}
	return nil
}
```

Use appropriate status code

- Adds clarity and understandability
- Easier to detect error
- But, need to find that sweet spot between security and robustness

```
GET /api/admin
Response:
Status: 401 Unauthorized
```

```
GET /api/admin
Response:
Status: 403 Forbidden
```

Don't just check errors, handle them gracefully

- Add context to errors
- Only place to panic is in main function
  - Any other place should return error and return them up the call stack.

```
if err != nil {
    Return err
}
```

```
if err != nil {
    return fmt.Errorf("could not create user: %s", err )
}
```

Context passing

- Passing context or request ID helps in tracing and debugging
- Correlation of logs
- Easier integration with platform like sentry, jaeger etc
- Maintain idempotency

```
{
  "level": "info",
  "time": "2024-04-28T10:15:30.123456789Z",
  "msg": "Received API request",
  "method": "POST",
  "endpoint": "/api/login",
  "requestID": "1622146825789228000",
  "body": {
    "username": "pthapa",
    "password": "2$1************************"
  }
}
```

ACID compliance

- Use contexts and transactions to make operation ACID compliant
- Useful when working with multi-collection updates

```go
// Start session
session, err := client.StartSession()
if err != nil {
    c.JSON(http.StatusInternalServerError, gin.H{"error": "Failed to start session"})
    return
}
defer session.EndSession(context.Background())

// Create a new mongo session context
sessionContext := mongo.NewSessionContext(context.Background(), session)

// Begin transaction
err = session.StartTransaction(
    options.Transaction().
        SetReadConcern(readconcern.Snapshot()).
        SetWriteConcern(writeconcern.Majority()),
)
if err != nil {
    c.JSON(http.StatusInternalServerError, gin.H{"error": "Failed to start transaction"})
    return
}
```

Use structured logs

- Using structured logs helps better maintainability
- Easier to debug and search
- Easier formatting and integration with observability tools
- Two types of logs:
    - Transport logs: logging http requests
    - Application logs: logging application level logs

Use structured logs

```go
func TransportJSONLoggerMiddleware(next http.Handler) http.Handler {
	return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
		startTime := time.Now()
		recorder := &responseWriterWithStatus{w, http.StatusOK}
		next.ServeHTTP(recorder, r)
		logData := map[string]interface{}{
			"status_code":   recorder.statusCode,
			"path":          r.URL.Path,
			"method":        r.Method,
			"start_time":    startTime.Unix(),
			"remote_addr":   r.RemoteAddr,
			"response_time": time.Since(startTime).String(),
			"request_id":    r.Header.Get("X-Request-Id"),
		}
		logLine, err := json.Marshal(logData)
		if err != nil {
			log.Println("Error marshalling log data:", err)
			return
		}
		log.Println(string(logLine))
	})
}
```

Implement health checks

- Helps us know readiness and liveliness of our application
- Very useful when using with service discovery
- Can be used to auto-scale


```json
{
	"errors": null,
	"status_code": 200,
	"result": {
		"message": "running",
		"commit_id": "ed31dc65c1",
		"branch": "main",
		"tag": "v1.0.0"
	}
}
```

```json
{
	"errors": null,
	"status_code": 200,
	"result": {
		"message": "health check completed",
		"status": {
			"database": "OK",
			"redis": "OK"
		}
	}
}
```

Handle retries gracefully

- Borked clients sometimes retry continuously
  - Adds load on server
- Resolution: Exponential backoff
  - Client waits progressively longer each time the backoff limit is reached
- Why
  - Makes API fault tolerant
  - Improved reliability
  - Better resource utilization
  - Enhanced user experience

Handle retries gracefully

- Use appropriate status code
  - 429 Too Many Request or 503 Service Unavailable
  - `Retry-After` header

```
type RetryConfig struct {
	MaxRetries int
	InitialDelay time.Duration
	MaxDelay time.Duration
}
```

Expose metrics

- Exposing metrics helps use put application in a feedback loop of performance monitoring
- Get initial baseline, compare baseline with new requests
- Benefits:
  - Helps detect anomaly
  - Capacity planning
  - RCA
  - Setting up SLAs

Version for maintainability

- Makes API backward compatible
- Flexibility on
    - Stability
    - Maintainability
    - Adaptability

```
v1 := http.NewServeMux()
v1.HandleFunc("/auth/login", handlerV1Login)
v1.HandleFunc("/auth/logout", handlerV1Logout)

v2 := http.NewServeMux()
v2.HandleFunc("/auth/login", handlerV2Login)
v2.HandleFunc("/auth/logout", handlerV2Logout)
```

Conclusion

Building an robust api is like trying to watch movie on this 90’s TV. We twist and turn knobs to find that sweet spot, where we are happy with the brightness, jitter, sound and picture quality.

Thank You

Youtube: @CodeWithPrabesh

Linkedin: @prabeshthapa

Github: @pgaijin66

Blog: 99devops.com

Scan me