Go Beyond Globals: A Practical Guide to Dependency Injection

Язык оригинала: en

# Оригинал

When building an application in Go, it's tempting to reach for global variables to share resources like configuration or a logger. It feels easy, but it's a path laden with technical debt.  
In this article we will try and investigate the best approach to make our dependencies available to all our handlers, its an approach that i have fallen in love with and use now in almost all my projects.  
First let's look at this common but problematic approach which entails assigning our dependencies as global variables:  
//using global variables  
var logger \*slog.Logger  
var appConfig config  
  
func main() {  
 // initialize logger and config...  
 // register handlers...  
}  
  
func HealthcheckHandler(w http.ResponseWriter, r \*http.Request) {  
 // This handler secretly depends on global state.  
 // It's not clear from its signature what it needs to run.  
 data := map[string]string{  
 "status": "available",  
 "environment": appConfig.env, // using global config  
 }  
 // ... write response ...  
}  
Enter fullscreen mode  
Exit fullscreen mode  
This code works, but it hides a time bomb. It creates implicit, "magical" dependencies that makes our application:  
Hard to Test  
: How do you test a handler that relies on global state? You have to manipulate those globals, which can lead to flaky tests that interfere with each other.  
Hard to Reason About  
: Looking at HealthcheckHandler's signature, you have no idea it needs appConfig. The function isn't honest about its requirements.  
Less Reusable  
: The handler is now tightly coupled to the main package and cannot be easily used elsewhere.  
A Cleaner Architecture: Dependency Injection  
Let's build a simple API to see it in action, focusing on three common dependencies:  
A config struct for application settings.  
A structured slog logger for consistent logging.  
A set of helper methods for centralized JSON error handling.  
Dependency #1: Application Configuration  
Almost every application needs configuration (port, environment, version, etc.). We'll define a struct to hold this data.  
// In main.go  
type config struct {  
 port int  
 env string  
}  
Enter fullscreen mode  
Exit fullscreen mode  
Dependency #2: The application Struct  
This struct is our central dependency container. We'll start by adding fields for our config and a structured logger.  
// In main.go  
import (  
 "log/slog"  
)  
  
type application struct {  
 config config  
 logger \*slog.Logger  
}  
Enter fullscreen mode  
Exit fullscreen mode  
Dependency #3: Centralized Helper Methods  
Dependency injection isn't just for data; it's also for behavior. We can define helper methods directly on our application struct. This is perfect for tasks like sending consistent JSON error responses.  
These helpers will have access to the other dependencies, like our logger!  
// In helpers.go  
package main  
  
import (  
 "encoding/json"  
 "net/http"  
)  
  
// The errorResponse method sends a JSON-formatted error message to the client  
// with a given status code. It uses our application logger!  
func (app \*application) errorResponse(w http.ResponseWriter, r \*http.Request, status int, message any) {  
 env := map[string]any{"error": message}  
  
 js, err := json.Marshal(env)  
 if err != nil {  
 app.logger.Error("failed to marshal error response", "error", err)  
 w.WriteHeader(http.StatusInternalServerError)  
 return  
 }  
  
 w.Header().Set("Content-Type", "application/json; charset=utf-8")  
 w.WriteHeader(status)  
 w.Write(js)  
}  
  
// A specific helper for server errors (500).  
func (app \*application) serverErrorResponse(w http.ResponseWriter, r \*http.Request, err error) {  
 app.logger.Error("server error", "request\_method", r.Method, "request\_url", r.URL.String(), "error", err)  
 message := "the server encountered a problem and could not process your request"  
 app.errorResponse(w, r, http.StatusInternalServerError, message)  
}  
Enter fullscreen mode  
Exit fullscreen mode  
By centralizing this, we ensure all our error responses look the same and are properly logged. Our handlers stay clean and DRY (Don't Repeat Yourself).  
Bringing it Together: Writing Handlers as Methods  
Now, our handlers become methods on \*application, giving them access to everything we've set up.  
// In handlers.go  
package main  
  
import (  
 "encoding/json"  
 "net/http"  
)  
  
// This healthcheck handler can now access the config and logger from app.  
func (app \*application) healthcheckHandler(w http.ResponseWriter, r \*http.Request) {  
 data := map[string]string{  
 "status": "available",  
 "environment": app.config.env,  
 "port": fmt.Sprintf("%d", app.config.port),  
 }  
  
 w.Header().Set("Content-Type", "application/json")  
 w.WriteHeader(http.StatusOK)  
 json.NewEncoder(w).Encode(data)  
}  
  
func (app \*application) createUserHandler(w http.ResponseWriter, r \*http.Request) {  
 var input struct {  
 Name string `json:"name"`  
 Email string `json:"email"`  
 }  
  
 err := json.NewDecoder(r.Body).Decode(&input)  
 if err != nil {  
 // Here we use our new helper!  
 app.errorResponse(w, r, http.StatusBadRequest, "invalid request body")  
 return  
 }  
  
 // ... validation logic for name and email ...  
  
 app.logger.Info("Creating new user", "name", input.Name, "email", input.Email)  
  
 // In a real application, we would pass this data to a `service` layer  
 // to handle database interactions.  
  
 w.WriteHeader(http.StatusCreated)  
 // ... send success response ...  
}  
Enter fullscreen mode  
Exit fullscreen mode  
The Final main.go  
Finally, we initialize our dependencies, "inject" them into an application instance, and register our methods with the router.  
// In main.go  
package main  
  
import (  
 "flag"  
 "fmt"  
 "log/slog"  
 "net/http"  
 "os"  
)  
  
// ... config struct definition ...  
// ... application struct definition ...  
  
func main() {  
 var cfg config  
  
 // 1. Initialize Dependencies  
 flag.IntVar(&cfg.port, "port", 4000, "API server port")  
 flag.StringVar(&cfg.env, "env", "development", "Environment (development|staging|production)")  
 flag.Parse()  
  
 logger := slog.New(slog.NewJSONHandler(os.Stdout, nil))  
  
 // 2. Create and Inject into the application struct  
 app := &application{  
 config: cfg,  
 logger: logger,  
 }  
  
 // 3. Register Method Handlers  
 mux := http.NewServeMux()  
 mux.HandleFunc("GET /v1/healthcheck", app.healthcheckHandler)  
 mux.HandleFunc("POST /v1/users", app.createUserHandler)  
  
 logger.Info("starting server", "addr", fmt.Sprintf(":%d", cfg.port), "env", cfg.env)  
  
 err := http.ListenAndServe(fmt.Sprintf(":%d", cfg.port), mux)  
 logger.Error("server failed", "error", err)  
 os.Exit(1)  
}  
Enter fullscreen mode  
Exit fullscreen mode  
The Payoff  
This dependency injection pattern is foundational for professional Go development. It gives you:  
Explicit Code  
: Your function signatures are honest. It's clear that your handlers require an \*application context to work.  
Superior Testability  
: You can easily create a mock application instance in your tests with a mock logger and fixed config, allowing you to test handler logic in perfect isolation.  
Maintainable Scale  
: Adding a new shared dependency (like a rate limiter or mail client) is as simple as adding a field to the application struct. No messy refactoring required.  
By moving away from global variables and embracing the application struct pattern for dependency injection, we are doing more than just cleaning up our code—we are laying a robust foundation for the future of our application.  
The beauty of this pattern lies in its extensibility. While we focused on configuration, logging, and helpers, you can see how easily this could expand to include other common dependencies:  
A \*sql.DB database connection pool.  
A pre-compiled \*template.Template cache.  
A client for an external service (like Stripe or an email provider).  
This approach provides a fantastic starting point for small to medium-sized applications.  
Give this pattern a try in your next Go project. It will lead to code that is cleaner, more professional, and a pleasure to maintain as it grows.  
THANK YOU AND HAPPY CODING

# Перевод на русский

When building an application in Go, it's tempting to reach for global variables to share resources like configuration or a logger. It feels easy, but it's a path laden with technical debt.  
In this article we will try and investigate the best approach to make our dependencies available to all our handlers, its an approach that i have fallen in love with and use now in almost all my projects.  
First let's look at this common but problematic approach which entails assigning our dependencies as global variables:  
//using global variables  
var logger \*slog.Logger  
var appConfig config  
  
func main() {  
 // initialize logger and config...  
 // register handlers...  
}  
  
func HealthcheckHandler(w http.ResponseWriter, r \*http.Request) {  
 // This handler secretly depends on global state.  
 // It's not clear from its signature what it needs to run.  
 data := map[string]string{  
 "status": "available",  
 "environment": appConfig.env, // using global config  
 }  
 // ... write response ...  
}  
Enter fullscreen mode  
Exit fullscreen mode  
This code works, but it hides a time bomb. It creates implicit, "magical" dependencies that makes our application:  
Hard to Test  
: How do you test a handler that relies on global state? You have to manipulate those globals, which can lead to flaky tests that interfere with each other.  
Hard to Reason About  
: Looking at HealthcheckHandler's signature, you have no idea it needs appConfig. The function isn't honest about its requirements.  
Less Reusable  
: The handler is now tightly coupled to the main package and cannot be easily used elsewhere.  
A Cleaner Architecture: Dependency Injection  
Let's build a simple API to see it in action, focusing on three common dependencies:  
A config struct for application settings.  
A structured slog logger for consistent logging.  
A set of helper methods for centralized JSON error handling.  
Dependency #1: Application Configuration  
Almost every application needs configuration (port, environment, version, etc.). We'll define a struct to hold this data.  
// In main.go  
type config struct {  
 port int  
 env string  
}  
Enter fullscreen mode  
Exit fullscreen mode  
Dependency #2: The application Struct  
This struct is our central dependency container. We'll start by adding fields for our config and a structured logger.  
// In main.go  
import (  
 "log/slog"  
)  
  
type application struct {  
 config config  
 logger \*slog.Logger  
}  
Enter fullscreen mode  
Exit fullscreen mode  
Dependency #3: Centralized Helper Methods  
Dependency injection isn't just for data; it's also for behavior. We can define helper methods directly on our application struct. This is perfect for tasks like sending consistent JSON error responses.  
These helpers will have access to the other dependencies, like our logger!  
// In helpers.go  
package main  
  
import (  
 "encoding/json"  
 "net/http"  
)  
  
// The errorResponse method sends a JSON-formatted error message to the client  
// with a given status code. It uses our application logger!  
func (app \*application) errorResponse(w http.ResponseWriter, r \*http.Request, status int, message any) {  
 env := map[string]any{"error": message}  
  
 js, err := json.Marshal(env)  
 if err != nil {  
 app.logger.Error("failed to marshal error response", "error", err)  
 w.WriteHeader(http.StatusInternalServerError)  
 return  
 }  
  
 w.Header().Set("Content-Type", "application/json; charset=utf-8")  
 w.WriteHeader(status)  
 w.Write(js)  
}  
  
// A specific helper for server errors (500).  
func (app \*application) serverErrorResponse(w http.ResponseWriter, r \*http.Request, err error) {  
 app.logger.Error("server error", "request\_method", r.Method, "request\_url", r.URL.String(), "error", err)  
 message := "the server encountered a problem and could not process your request"  
 app.errorResponse(w, r, http.StatusInternalServerError, message)  
}  
Enter fullscreen mode  
Exit fullscreen mode  
By centralizing this, we ensure all our error responses look the same and are properly logged. Our handlers stay clean and DRY (Don't Repeat Yourself).  
Bringing it Together: Writing Handlers as Methods  
Now, our handlers become methods on \*application, giving them access to everything we've set up.  
// In handlers.go  
package main  
  
import (  
 "encoding/json"  
 "net/http"  
)  
  
// This healthcheck handler can now access the config and logger from app.  
func (app \*application) healthcheckHandler(w http.ResponseWriter, r \*http.Request) {  
 data := map[string]string{  
 "status": "available",  
 "environment": app.config.env,  
 "port": fmt.Sprintf("%d", app.config.port),  
 }  
  
 w.Header().Set("Content-Type", "application/json")  
 w.WriteHeader(http.StatusOK)  
 json.NewEncoder(w).Encode(data)  
}  
  
func (app \*application) createUserHandler(w http.ResponseWriter, r \*http.Request) {  
 var input struct {  
 Name string `json:"name"`  
 Email string `json:"email"`  
 }  
  
 err := json.NewDecoder(r.Body).Decode(&input)  
 if err != nil {  
 // Here we use our new helper!  
 app.errorResponse(w, r, http.StatusBadRequest, "invalid request body")  
 return  
 }  
  
 // ... validation logic for name and email ...  
  
 app.logger.Info("Creating new user", "name", input.Name, "email", input.Email)  
  
 // In a real application, we would pass this data to a `service` layer  
 // to handle database interactions.  
  
 w.WriteHeader(http.StatusCreated)  
 // ... send success response ...  
}  
Enter fullscreen mode  
Exit fullscreen mode  
The Final main.go  
Finally, we initialize our dependencies, "inject" them into an application instance, and register our methods with the router.  
// In main.go  
package main  
  
import (  
 "flag"  
 "fmt"  
 "log/slog"  
 "net/http"  
 "os"  
)  
  
// ... config struct definition ...  
// ... application struct definition ...  
  
func main() {  
 var cfg config  
  
 // 1. Initialize Dependencies  
 flag.IntVar(&cfg.port, "port", 4000, "API server port")  
 flag.StringVar(&cfg.env, "env", "development", "Environment (development|staging|production)")  
 flag.Parse()  
  
 logger := slog.New(slog.NewJSONHandler(os.Stdout, nil))  
  
 // 2. Create and Inject into the application struct  
 app := &application{  
 config: cfg,  
 logger: logger,  
 }  
  
 // 3. Register Method Handlers  
 mux := http.NewServeMux()  
 mux.HandleFunc("GET /v1/healthcheck", app.healthcheckHandler)  
 mux.HandleFunc("POST /v1/users", app.createUserHandler)  
  
 logger.Info("starting server", "addr", fmt.Sprintf(":%d", cfg.port), "env", cfg.env)  
  
 err := http.ListenAndServe(fmt.Sprintf(":%d", cfg.port), mux)  
 logger.Error("server failed", "error", err)  
 os.Exit(1)  
}  
Enter fullscreen mode  
Exit fullscreen mode  
The Payoff  
This dependency injection pattern is foundational for professional Go development. It gives you:  
Explicit Code  
: Your function signatures are honest. It's clear that your handlers require an \*application context to work.  
Superior Testability  
: You can easily create a mock application instance in your tests with a mock logger and fixed config, allowing you to test handler logic in perfect isolation.  
Maintainable Scale  
: Adding a new shared dependency (like a rate limiter or mail client) is as simple as adding a field to the application struct. No messy refactoring required.  
By moving away from global variables and embracing the application struct pattern for dependency injection, we are doing more than just cleaning up our code—we are laying a robust foundation for the future of our application.  
The beauty of this pattern lies in its extensibility. While we focused on configuration, logging, and helpers, you can see how easily this could expand to include other common dependencies:  
A \*sql.DB database connection pool.  
A pre-compiled \*template.Template cache.  
A client for an external service (like Stripe or an email provider).  
This approach provides a fantastic starting point for small to medium-sized applications.  
Give this pattern a try in your next Go project. It will lead to code that is cleaner, more professional, and a pleasure to maintain as it grows.  
THANK YOU AND HAPPY CODING