intro

- Right now we just have this dummy endpoint that client can hit
- Does not really do anything, nothing happens, just prints message on server side and returns message
- We will now transform this into the endpoint that will receive our commands

control_api.go

Right now this contains 2 functions

- StartControlAPI() is called from our server's main, starts the control API listener, we can also see makes one endpoint available "`/dummy`"
- `dummyHandler` is the function that is called when the endpoint is hit

```go
func StartControlAPI() {
    // Create Chi router
    r := chi.NewRouter()

    // Define the POST endpoint
    r.Get("/dummy", dummyHandler)

    log.Println("Starting Control API on :8080")
    go func() {
        if err := http.ListenAndServe(":8080", r); err != nil {
            log.Printf("Control API error: %v", err)
        }
    }()
}

func dummyHandler(w http.ResponseWriter, r *http.Request) {

    log.Println("dummyHandler called")

    response := "Dummy endpoint triggered"

    json.NewEncoder(w).Encode(response)
}
```

```
```

StartControlAPI()

- First off let's just change the name of endpoint to something more apt like /command
- Then, let's also change from GET to POST since we will send a POST body with instructions, arguments etc

```go
func StartControlAPI() {
    // Create Chi router
    r := chi.NewRouter()

    // Define the POST endpoint
    r.Post("/command", commandHandler) // CHANGE THIS

    log.Println("Starting Control API on :8080")
    go func() {
        if err := http.ListenAndServe(":8080", r); err != nil {
            log.Printf("Control API error: %v", err)
        }
    }()
}
```

- We want to now of course change dummyHandler, but before we do we need a new type to represent the command (keywork + arguments) received by the client
- Now we will have many custom types, so good convention is to group them all together

internal/models/types.go

- We add the following

```go
// CommandClient represents a command with its arguments as sent by
Client
type CommandClient struct {
    Command   string          `json:"command"`
    Arguments json.RawMessage `json:"data,omitempty"`
}
```

- This represent the command received by our server from the client, in the case of this workshop we'll just use `curl`
- Arguments is another json - this is the command-specific arguments
- So for example load will have specific arguments, and then if we say Download, that will have other arguments, Upload yet others etc
- So this is essentially just where the command-specific arguments will go of the command that is called, `json.RawMessage` so it is "flexible" to accommodate all the different options
- Now, while we are here, we might as well add our specific argument struct for shellcode loader

```go
// ShellcodeArgsClient contains the command-specific arguments for
Shellcode Loader as sent by Client
type ShellcodeArgsClient struct {
    FilePath   string `json:"file_path"`
    ExportName string `json:"export_name"`
}
```

We have 2 arguments

- FilePath is where the DLL containing the shellcode resides on the server
- ExportName is the name of the exported function in the DLL that should be called

We'll discuss these in more detail when we discuss their validation, for now I just wanted you to be aware of them

Further, it's worth noting that this is specifically called `ShellcodeArgsClient` and not just `ShellcodeArgs` more broadly. That's because, in this case, the arguments as we receive them from the client will not be exactly the same when we send them to Agent following processing, so that will require its own custom type. More on this later.

So let's get back to `control_api.go`

- Now we can actually instantiate an empty `CommandType` struct, and deserialize the message we received from the client into it

```go
func commandHandler(w http.ResponseWriter, r *http.Request) {

    // Instantiate custom type to receive command from client
    var cmdClient models.CommandClient

    // The first thing we need to do is unmarshall the request body into the custom type
    if err := json.NewDecoder(r.Body).Decode(&cmdClient); err != nil {
        log.Printf("ERROR: Failed to decode JSON: %v", err)
        w.WriteHeader(http.StatusBadRequest)
        json.NewEncoder(w).Encode("error decoding JSON")
        return
    }

    // Visually confirm we get the command we expected
    var commandReceived = fmt.Sprintf("Received command: %s", cmdClient.Command)
    log.Printf(commandReceived)

    // Confirm on the client side command was received
    w.WriteHeader(http.StatusOK)
    json.NewEncoder(w).Encode(commandReceived)

}
```

test

- Run server with go run `./cmd/server`
- Now run the following `curl` command in another terminal window

```
curl -X POST http://localhost:8080/command -d '{"command": "load"}'
```

- As we can see on the server side we received the command

```
❯ go run ./cmd/server
2025/11/04 13:44:38 Starting Control API on :8080
2025/11/04 13:44:38 Starting  server on 127.0.0.1:8443
2025/11/04 13:44:56 Received command: load
```

- And indeed on the client side too we can see we received the specific command

```
❯ curl -X POST http://localhost:8080/command -d '{"command": "load"}'
"Received command: load"
```

SHORT BRIDGE TO NEXT LESSON