intro

- There is a lot now we need to do
- First, validate command - does it exist?
- If not - reject
- If yes, process command, process arguments for that command, is that valid?
- If not - reject
- If yes, prepare another struct that will be server instructions for the agent, which then goes in a queue to wait for when agent checks in again

In this section now we'll simply make sure the command actually exists

Basic idea

- Somewhere we store all our valid commands
- When we receive a command it looks there if found - good - if not - rjects

we want to validate command

- We want to queue command for agent to receive BUT ONLY if it's a real command! (FAIL FAST PRINCIPLE SHORT EXPLANATION)
- SO here we should validate immediately before anything

We will create a `map` as a "set" to store our valid commands, as it's the most efficient way to check if a command exists

Why a Map and Not a List?

- **If you used a list (slice):** `[]string{"load", "unload", "start"}` To check if a command exists, you would have to loop through the entire list. This is slow, especially if you have 1,000 commands. (This is an **O(n)** operation).
- **By using a map:** `map[string]...` Checking if a key exists (`if _, ok := myMap["load"]`) is almost instant, no matter how many commands you add. (This is an **O(1)** operation).

The most memory-efficient and idiomatic way in Go to create a set is to use an **empty struct** (`struct{}`) as the value, because it takes up **zero memory**.

Let's create a new file - `command_api.go`, also inside of `control`

```
// Registry of valid commands with their validators and processors
var validCommands = map[string]struct {
}{
    "shellcode": {},
}
```

- So key is string (name of command) and for now struct is empty
- This is the idiomatic way for "lookups" - using emptry struct (as explained above)
- however, we will use this later and add to the value struct
    - argument validator: based on the command, validate arguments
    - processor: some arguments need to be processed/prepared before being queued
- Now we have something we can validate against in our handler

handleCommand

- First "normalize" - all lower case - to ensure for example if the wrote Shellcode or SHELLCODE etc its still accepted

```
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)

    // Check if command exists
    _, exists := validCommands[cmdClient.Command]
    if !exists {
        var commandInvalid = fmt.Sprintf("ERROR: Unknown command: %s",
cmdClient.Command)
        log.Printf(commandInvalid)
        w.WriteHeader(http.StatusBadRequest)
        json.NewEncoder(w).Encode(commandInvalid)
        return
    }

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

}
```

So our new code is on lines 18 to 26

- explain it briefly
- explain why checking against a map gives us 2 values `_, exists := validCommands[cmdClient.Command]`

test

- We can now test
- Let's run our server
- Let's call first with a made-up command, and then with load

```
❯ curl -X POST http://localhost:8080/command -d '{"command": "derp"}'
"ERROR: Unknown command: derp"
❯ curl -X POST http://localhost:8080/command -d '{"command":
"shellcode"}'
"Received command: shellcode"
```

```
❯ go run ./cmd/server
2025/11/04 14:44:42 Starting Control API on :8080
2025/11/04 14:44:42 Starting  server on 127.0.0.1:8443
2025/11/04 14:44:46 Received command: derp
2025/11/04 14:44:46 ERROR: Unknown command: derp
2025/11/04 14:44:50 Received command: shellcode
```

COPY CODE TO FOLDER