First let's just understand process

DRAW THIS!

SERVER
-> RUNLOOP
-> ExecuteTask
-> Orchestrator
-> Doer

Runloop - this is of course the agent's contact point with the server, the response we receive from the server,

We just saw it now has access to response in form of models.ServerResponse, which will indicate whether there is a job or not, and if there is it has the command, arguments, and jobID.

Now we don't want to do ANY processing inside of the runloop, instead if there is a job we hand it off to another function which we'll call ExecuteTask, this will do all the high-level processing. if detected that something is present it is immediately sent to another function `ExecuteTask` for processing since that is not the domain of runloop.

`ExecuteTask` can be thought of as a Command Router, it receives the response from the Server, all it knows that "there is a command here", but nothing else. It is thus responsible then for checking what is the command, and calling that command's Orchestrator.

Note that `ExecuteTask` will also receive the result from the Orchestrator after the command is done, and is responsible for then sending result back to the server.

Every command has 2 types of functions - Orchestrator and Doer.

The Orchestrator receives command struct from ExecuteTask, unpacks, validates the arguments, process the arguments if required for the Doer, and then calls the doer. It will then also after the doer is done receive result, and roll the command-specific results back into a main/generic response struct before returning it to `ExecuteTask`

The Doer is the function that actually performs the action. So it receives arguments in the form that is required, and then executes. Something to note however is that, for many types of commands, the Doer is expressed as an interface, allowing OS-specific version. With for example load, since reflective loading shellcode is highly contingent upon the OS API, the Orchestrator calls the interface, and the there is a Win-specific implementation thereof. Though we will not create a Linux and Darwin-specific implementation thereof in this course, it is set up to do just that.

FOR EXAMPLE

So for example if the command is `download` in the ServerTaskResponse received in `executeTask`, it calls

- `orchestrateDownload`, which will pull the specific download struct from `ServerTaskResponse` called `DownloadArgs` and then pass this as argument to
- `doDownload`, which will then work with `DownloadArgs` to actually perform the action, it returns `AgentTaskResult` (partially completed) to
- `orchestrateDownload`, which then completes `AgentTaskResult` and passes this back to
- `executeTask` which will finally call SendResult to send `AgentTaskResult` back to the server

now in this specific lesson

We'll now create ExecuteTask.

So in RunLoop we just added these lines at the end of the previous lesson

```
if response.Job {
    log.Printf("Job received from Server\n-> Command: %s\n-> JobID: %s",
response.Command, response.JobID)
} else {
    log.Printf("No job from Server")
}
```

So if `response.Job`, let's then also send response to `executeTask`, which will be a method on agent.

```
        if response.Job {
            log.Printf("Job received from Server\n-> Command: %s\n->
JobID: %s", response.Command, response.JobID)
            agent.ExecuteTask(response)
        } else {
            log.Printf("No job from Server")
        }
```

This will of course error out atm since we have not yet created it.

So we need that method, but before we even get to it let's talk about how commands will be registered and located on agent's side since in ExecuteTask it has to match the command keyword, say `shellcode`, with its orchestrator.

So how will we do it? Well there are a few ways we could do it really.

- a switch, and that switch then in turn calls the correct function for say `shellcode`
- This simple system works, and in a simple application such as ours, totally fine
- But I want to teach you a better system
- This is not only more maintainable, but ESP later when we want to do "selective compilation" (i.e. don't just assume each agent will always be compiled with every single agent, but we can choose which to do, ie command abilities are completely modular can be added and removed dynamically, we will NEED this type of design
- So what we're instead going to create is a **CommandOrchestrator Map**

Orchestrator Function Signature

- To create the map we want, where we map
  - Key: string/keyword of the command (for example `shellcode`)
  - Value: The orchestrator function (NOT the call (), but the ACTUAL FUNCTION ie without ())
- We need a function signature, we need to ensure ALL our orchestrators conform to same pattern, we can't make a function as the value if they have different shapes!

We need to know this for now since we need to know what signature will look like

So let's add the following to **internal/agent/commands.go**

```
type OrchestratorFunc func(agent *Agent, job *models.ServerResponse)
models.AgentTaskResult
```

So we'll notice models.AgentTaskResult errors out, we need to create it.
This is the result sent from the agent to the server after it has executed.

It will once again be similar than before in that we will have a top-level generic struct in this case AgentTaskResult, but inside of it again we need something to have specific results for specific commands. For exmaple the results for Load, Shell-command, upload, download etc will all be different, we need to account for that.

So now in models/types.go

```
type AgentTaskResult struct {
    JobID  string `json:"job_id"`
    Success bool `json:"success"`
    CommandResult json.RawMessage `json:"command_result,omitempty"`
    Error   error `json:"error,omitempty"`
}
```

JobID obvs the same one we received - this allows the server to reconcile the result with the specific command that was dispatched.

Success true if succeeded, false if failed

CommandResult will contain the results specific to a command

Error - if it failed, the specific error message

Now back in `commands.go` this should no longer throw error

```
type OrchestratorFunc func(agent *Agent, job *models.ServerResponse)
models.AgentTaskResult
```

WHAT ARE WE DOING HERE...

- We are creating A TYPE!!!!

- The TYPE is a function!

Also note - above we only said we want one argument, but... Here we can see two!

- ALSO, we want to a pointer to the actual agent, so that it has access to its fields

But to be clear, and you'll notice this shortly when we actually define our individual implementations of this function type:

- `func(agent *Agent, job models.ServerResponse)` makes it seem like both are arguments, THEY ARE NOT!
- `job models.ServerTaskResponse` is indeed an argument but
- `agent *Agent` will actually be the method receiver!

This is possible due to they way we are going to register them below

Called METHOD EXPRESSION

Please give a short quick intro lesson on method expression here so they understand

agent struct

- So we want our agent to have this new ability, this new map connecting keywords to these orchestrator functions, s
- So this map is now added as a field to our Agent struct

```
type Agent struct {
    serverAddr string
    client     *http.Client

    commandOrchestrators map[string]OrchestratorFunc // Maps commands to
their keywords
}
```

constructor

- And of course now that we have this field, we need to instantiate it
- But note there are two seperate, but related

> - We need to FIRST create the actual map - that is call make to give us an actual map we can use
> - And THEN, we need to "register" all our commands. Right now we'll just register every single command we have, but again in the future we can add some logic here to give the user the choice which commands they want to compile with, and then ONLY register those

WHY DO WE NEED TO USE MAKE, WHY DONT WE JUST GET A MAP

Sometimes:

- Go DOES automatically give me a usable object, like with a mutex, if I add it to struct I DONT need to instantiate
- But with channels and maps I do need to actually use make etc

I know Go is really a very thoughtfully designed language, so i am sure there is some design motivaiton or principle underpinning this, do you know?

The core principle in Go is that a type's "zero value" should be useful. This results in two different behaviors. For types like `sync.Mutex` or structs, their zero value is a complete, usable state (e.g., an unlocked mutex), so they work immediately without initialization. In contrast, reference types like maps, slices, and channels have a zero value of `nil`. This `nil` signifies that the variable is a pointer to an underlying data structure that has not been created yet. You cannot use a `nil` map because the memory to store key-value pairs doesn't exist, which is why attempting to do so causes a panic.

Therefore, you must use the `make()` function for these reference types. `make()` is the explicit instruction to allocate the necessary underlying data structure, transforming the `nil` variable into a usable instance. This distinction is a deliberate design choice for safety: simple types just work, while you are forced to be intentional about allocating memory for more complex, dynamic structures.

```
// NewAgent creates a new HTTPS agent
func NewAgent(serverAddr string) *Agent {
    // Create TLS config that accepts self-signed certificates
    tlsConfig := &tls.Config{
        InsecureSkipVerify: true,
```

```
    }

    // Create HTTP client with custom TLS config
    client := &http.Client{
        Transport: &http.Transport{
            TLSClientConfig: tlsConfig,
        },
    }

    agent := &Agent{
        serverAddr:           serverAddr,
        client:               client,
        commandOrchestrators: make(map[string]OrchestratorFunc), // WE
NEED TO INSTANTIATE
    }

    registerCommands(agent) // NOT YET IMPLEMENT - register individual
commands

    return agent
}
```

As you can see we need to first instantiate `agent` so we can can instantiate `commandOrchestrators`, then register, then return.

- So our two changes are

```
commandOrchestrators: make(map[string]OrchestratorFunc), // WE NEED TO
INSTANTIATE

registerCommands(agent) // REGISTER ALL OUR COMMANDS
```

registerCommands

Now let's create `registerCommands`

- Below, yes we could just register them right here, but this would kind of just "bury" them in the constructor, again since in the future differential registering will become more complex I want to carve this out to its own function
- Remember for now we only have 1 command = load

- If we add more, this is where we add them
- Add this to commands.go

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

Remember `orchestrateShellcode` will error out sicne we have not yet created it - do that in next lesson

Because of this, to allow us to test, for now just comment it out, we'll uncomment at start of next lesson

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

finally to close of this lesson let's just create a basic skeleton for ExecuteTask so that we can at least test it

So in agent/commands.go let's add

```
func (agent *Agent) ExecuteTask(job *models.ServerResponse) {
    log.Printf("AGENT IS NOW PROCESSING COMMAND %s with ID %s",
job.Command, job.JobID)
}
```

So this obvs wont do much yet - we'll worry about that later - but for now it will allow us to at least confirm that our command reaches this function

test

- Run server
- Run client

- Then run agent

```
❯ go run ./cmd/agent
2025/11/06 17:53:33 Starting Agent Run Loop
2025/11/06 17:53:33 Delay: 5s, Jitter: 50%
2025/11/06 17:53:33 Job received from Server
-> Command: shellcode
-> JobID: job_506522
2025/11/06 17:53:33 AGENT IS NOW PROCESSING COMMAND shellcode with ID
job_506522
2025/11/06 17:53:33 Sleeping for 6.825524134s
^C2025/11/06 17:53:37 Shutting down client...
```

And we can see the output show us it's reached that function (AGENT IS NOW...)