intro

So let's remind ourselves where do the agent "make contact" with server?

In `server/server.go` it is of course our RootHandler that is called when the endpoint is hit

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

    log.Printf("Endpoint %s has been hit by agent\n", r.URL.Path)

    // Set content type to JSON
    w.Header().Set("Content-Type", "application/json")

    response := "You have hit the server's root endpoint"

    // Encode and send the response
    if err := json.NewEncoder(w).Encode(response); err != nil {
        log.Printf("Error encoding response: %v\n", err)
        http.Error(w, "Internal Server Error",
http.StatusInternalServerError)
        return
    }

}
```

- So very simple, nothing really happens, we get a message on the server side, and on the client side just to help us confirm it worked - that's it
- But what we want to do now is when this endpoint is hit:
    - Is there a command in index 0 in queue?
    - If no - respond saying false
    - If yes - retrieve it from queue, REMOVE it from queue, and send to the agent

The first thing to note however now before we even get to that is we need another custom type

Right now we have of course the command and (processed) arguments we received from client

```
type CommandClient struct {
    Command   string          `json:"command"`
    Arguments json.RawMessage `json:"data,omitempty"`
}
```

But to respond to the agent, this is not enough, we want 2 extra fields

- A simple field just indicating - yes, there is a command, no there is no command
- JobID -
- In the scope of our project not really necessary, but simple enough to implement and won't hurt
- Just want to generate a random job number that will also be sent to agent
- So for example let's say something is placed in queue it's called job_1234
- Then when agent returns results, it can say "here are the results for job_1234"
- Should be obvs that, when things become more complex, multiple clients, agents etc this becomes critical for traceablity

So in models/types.go let's add the following

```
// ServerResponse represents a response from the server to the agent
type ServerResponse struct {
    Job       bool            `json:"job"`
    JobID     string          `json:"job_id,omitempty"`
    Command   string          `json:"command,omitempty"`
    Arguments json.RawMessage `json:"data,omitempty"`
}
```

So first 2 fields are new

- Job false if no job true if there is a job
- If there is no job, none of the other fields will be included
- If there is a job we will generate an ID (in Root Handler), and simple copy the other 2 fields from CommandClient

Now we can jump into RootHandler, but first, we need a new function

- This function will be called from RootHandler
- It will check if there is a command in index 0 in queue
- If no - return false
- If yes - return true, return contents of index, remove that from the queue

Let's add this to command_api.go`

```
// GetCommand retrieves and removes the next command from queue
func (cq *CommandQueue) GetCommand() (models.CommandClient, bool) {
    cq.mu.Lock()
    defer cq.mu.Unlock()

    if len(cq.PendingCommands) == 0 {
        return models.CommandClient{}, false
    }

    cmd := cq.PendingCommands[0]
    cq.PendingCommands = cq.PendingCommands[1:]

    log.Printf("DEQUEUED: Command '%s'", cmd.Command)

    return cmd, true
}
```

- SO first it check if its empty (ie there is no command), then it returns empty struct, and false
- But, if true, then cmd is assigned to CommandType at index 0 (ie at the front of the queue)
- and then this is an idiomatic way to essentially remove what is at the front of the queue, and move everything 1 up `cq.PendingCommands = cq.PendingCommands[1:]`

Great, so now we have the function to interact with the queue, we can now jump back into server.go and implement our real RootHandler

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

    log.Printf("Endpoint %s has been hit by agent\n", r.URL.Path)

    var response models.ServerResponse

    // Check for pending commands
    cmd, exists := control.AgentCommands.GetCommand()
    if exists {
        log.Printf("Sending command to agent: %s\n", cmd.Command)
        response.Job = true
        response.Command = cmd.Command
        response.Arguments = cmd.Arguments
        response.JobID = fmt.Sprintf("job_%06d", rand.Intn(1000000))
        log.Printf("Job ID: %s\n", response.JobID)
    } else {
        log.Printf("No commands in queue")
    }

    // Set content type to JSON
    w.Header().Set("Content-Type", "application/json")

    // Encode and send the response
    if err := json.NewEncoder(w).Encode(response); err != nil {
        log.Printf("Error encoding response: %v\n", err)
        http.Error(w, "Internal Server Error",
http.StatusInternalServerError)
        return
    }

}
```

Before we can test it there is one other small change we should make in RunLoop() we have the following line

```
log.Printf("Response from server: %s", response)
```

Now this was fine when we received a single string, but right now it will print the entire JSON, including the base64 string, which will just fill up lines and lines of terminal. So instead let's reconfigure this to be a bit more legible.

But before we even do that, let's make some changes to Send().

```
// Send implements Communicator.Send for HTTPS
func (agent *Agent) Send(ctx context.Context) ([]byte, error) {
    // Construct the URL
    url := fmt.Sprintf("https://%s/", agent.serverAddr)

    // Create GET request
    req, err := http.NewRequestWithContext(ctx, "GET", url, nil)
    if err != nil {
        return nil, fmt.Errorf("creating request: %w", err)
    }

    // Send request
    resp, err := agent.client.Do(req)
    if err != nil {
        return nil, fmt.Errorf("sending request: %w", err)
    }
    defer resp.Body.Close()

    // Check status code
    if resp.StatusCode != http.StatusOK {
        body, _ := io.ReadAll(resp.Body)
        return nil, fmt.Errorf("server returned status %d: %s",
resp.StatusCode, body)
    }

    // Read response body
    body, err := io.ReadAll(resp.Body)
    if err != nil {
        return nil, fmt.Errorf("reading response: %w", err)
    }

    // Return the raw JSON as message data
    return body, nil
}
```

Right now it just returns a byte slice containing our raw JSON response body
But we of course now have a type specifically for this - `models.ServerResponse`
So let's now rather unmarshall response into this, and then return that

```
// Send implements Communicator.Send for HTTPS
func (agent *Agent) Send(ctx context.Context) (*models.ServerResponse,
error) {
    // Construct the URL
    url := fmt.Sprintf("https://%s/", agent.serverAddr)

    // Create GET request
    req, err := http.NewRequestWithContext(ctx, "GET", url, nil)
    if err != nil {
        return nil, fmt.Errorf("creating request: %w", err)
    }

    // Send request
    resp, err := agent.client.Do(req)
    if err != nil {
        return nil, fmt.Errorf("sending request: %w", err)
    }
    defer resp.Body.Close()

    // Check status code
    if resp.StatusCode != http.StatusOK {
        body, _ := io.ReadAll(resp.Body)
        return nil, fmt.Errorf("server returned status %d: %s",
resp.StatusCode, body)
    }

    // Read response body
    body, err := io.ReadAll(resp.Body)
    if err != nil {
        return nil, fmt.Errorf("reading response: %w", err)
    }

    // Unmarshal into ServerResponse
    var serverResp models.ServerResponse
    if err := json.Unmarshal(body, &serverResp); err != nil {
        return nil, fmt.Errorf("unmarshaling response: %w", err)
```

```
    }

    // Return the parsed response
    return &serverResp, nil
}
```

So we change signature to now return this `*models.ServerResponse`

And add the following code at bottom

```
    // Unmarshal into ServerResponse
    var serverResp models.ServerResponse
    if err := json.Unmarshal(body, &serverResp); err != nil {
        return nil, fmt.Errorf("unmarshaling response: %w", err)
    }

    // Return the parsed response
    return &serverResp, nil
```

OK so back in RunLoop(), when we call Send(), we now of course get this custom type back, so we can now print individual fields.

So after we receive response, we now have this new code to simply print if there is a job or not, and if there is, what is the command and ID

```
    response, err := agent.Send(ctx)
        if err != nil {
            log.Printf("Error sending request: %v", err)
            // Don't exit - just sleep and try again
            time.Sleep(cfg.Timing.Delay)
            continue // Skip to next iteration
        }

        // REMOVE THIS
        log.Printf("Response from server: %s", response)

        // NEW CODE
        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")
        }
```

test

CLIENT

```
❯ curl -X POST http://localhost:8080/command \
  -d '{
    "command": "shellcode",
    "data": {
      "file_path": "./payloads/calc.dll",
      "export_name": "LaunchCalc"
    }
  }'
"Received command: shellcode"
```

SERVER

```
❯ go run ./cmd/server
2025/11/06 15:37:44 Starting Control API on :8080
2025/11/06 15:37:44 Starting  server on 0.0.0.0:8443
2025/11/06 15:37:49 Endpoint / has been hit by agent
2025/11/06 15:37:49 No commands in queue
2025/11/06 15:37:54 Endpoint / has been hit by agent
2025/11/06 15:37:54 No commands in queue
2025/11/06 15:38:01 Endpoint / has been hit by agent
2025/11/06 15:38:01 No commands in queue
2025/11/06 15:38:03 Received command: shellcode
2025/11/06 15:38:03 Validation passed: file_path=./payloads/calc.dll,
export_name=LaunchCalc
2025/11/06 15:38:03 Processed file: ./payloads/calc.dll (111493 bytes) ->
base64 (148660 chars)
2025/11/06 15:38:03 Processed command arguments: shellcode
2025/11/06 15:38:03 QUEUED: shellcode
2025/11/06 15:38:04 Endpoint / has been hit by agent
2025/11/06 15:38:04 DEQUEUED: Command 'shellcode'
```

```
2025/11/06 15:38:04 Sending command to agent: shellcode
2025/11/06 15:38:04 Job ID: job_411895
2025/11/06 15:38:08 Endpoint / has been hit by agent
2025/11/06 15:38:08 No commands in queue
```

CLIENT

```
❯ go run ./cmd/agent
2025/11/06 15:37:49 Starting Agent Run Loop
2025/11/06 15:37:49 Delay: 5s, Jitter: 50%
2025/11/06 15:37:49 No job from Server
2025/11/06 15:37:49 Sleeping for 5.22541057s
2025/11/06 15:37:54 No job from Server
2025/11/06 15:37:54 Sleeping for 6.748574669s
2025/11/06 15:38:01 No job from Server
2025/11/06 15:38:01 Sleeping for 3.650038675s
2025/11/06 15:38:04 Job received from Server
-> Command: shellcode
-> JobID: job_411895
2025/11/06 15:38:04 Sleeping for 3.454947595s
2025/11/06 15:38:08 No job from Server
2025/11/06 15:38:08 Sleeping for 6.82275109s
```

Perfect!

Great so that means for now we are done with the server

- The agent is now receiving this new response type
- It now needs to be able to process it, and then perform correct actions

Let's go!
