Lab2 - Concurrency and RPCs in Go

0. Go Concurrency

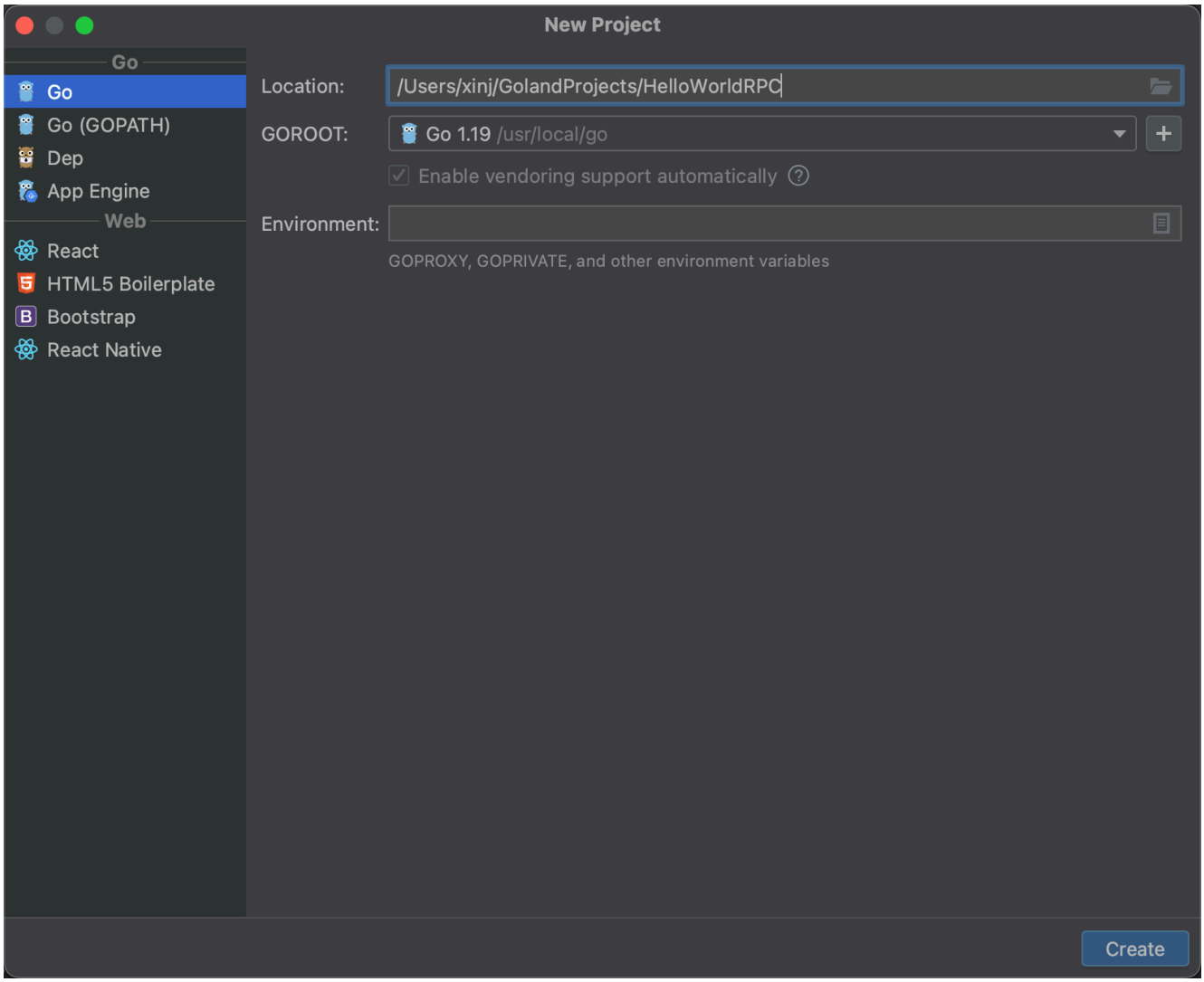
As mentioned in last lab, you should already learned through the A Tour of Go - Concurrency.

If not, do it now.

---

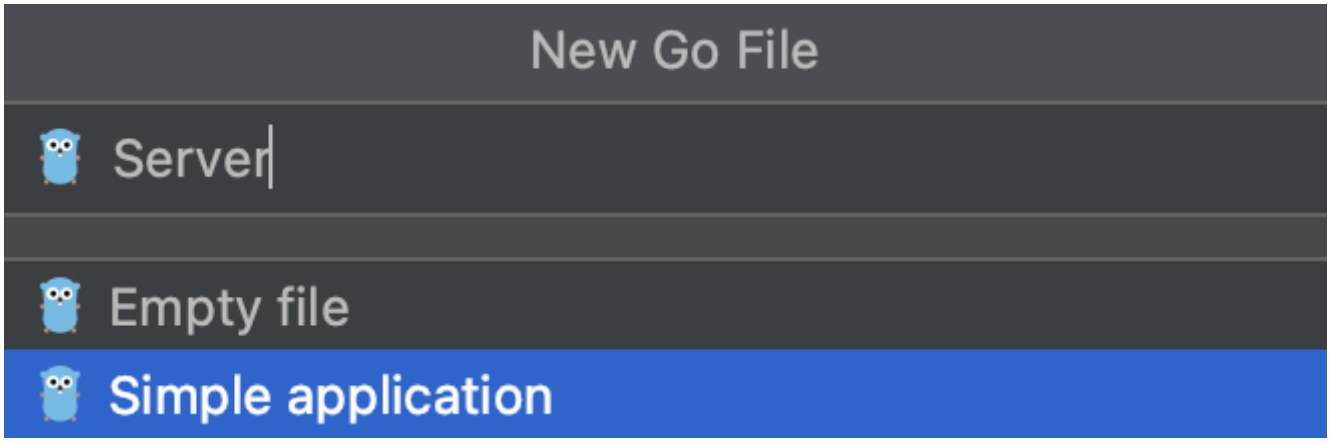
1. RPC Hello World

- Create a **Go Module** Project `HelloWorldRPC`.



- Create `server.go`

  By choosing **Simple application**, the created file is under the **main** package with a **main** function.



- Copy code to `server.go` , please read comments and understand the code.

```
package main

import (
  "log"
  "net"
  "net/rpc" // Go's RPC package
)

/*
Go does not support 'class'.
Define an empty struct as RPC service
*/
type HelloService struct{}

/*
Function will be called by RPC.
(p *HelloService) specifies the receiver, so Hello is a HelloService's 'method'.
To be an RPC function, there are three rules :
1. Only have two serializable parameters, and the second is pointer
2. With one return value - error
3. Public method
*/
func (p *HelloService) Hello(request string, reply *string) error {
  *reply = "hello:" + request
  return nil
}

func main() {
  rpc.RegisterName("HelloService", new(HelloService)) // Register all methods of
HelloService who fit three rules as an RPC function
  Listener, err := net.Listen("tcp", ":1234")         // Listen to a TCP port, and
return a listener
  if err != nil {
    log.Fatal("ListenTCP error:", err)
  }
```

```
  conn, err := Listener.Accept() //The listener will block the code until a TCP
connection built from the port.
  if err != nil {
    log.Fatal("Accept error:", err)
  }
  rpc.ServeConn(conn) // Provide RPC service on our TCP connection
}
```

- Create `client.go`.
- Copy code to `client.go`, please read comments and understand the code.

```
package main

import (
  "fmt"
  "log"
  "net/rpc"
)

func main() {
  client, err := rpc.Dial("tcp", "localhost:1234") // Dial to RPC destination
  if err != nil {
    log.Fatal("dialing:", err)
  }

  var reply string
  err = client.Call("HelloService.Hello", "hello", &reply) // Call RPC function
  if err != nil {
    log.Fatal(err)
  }

  fmt.Println(reply)
}
```

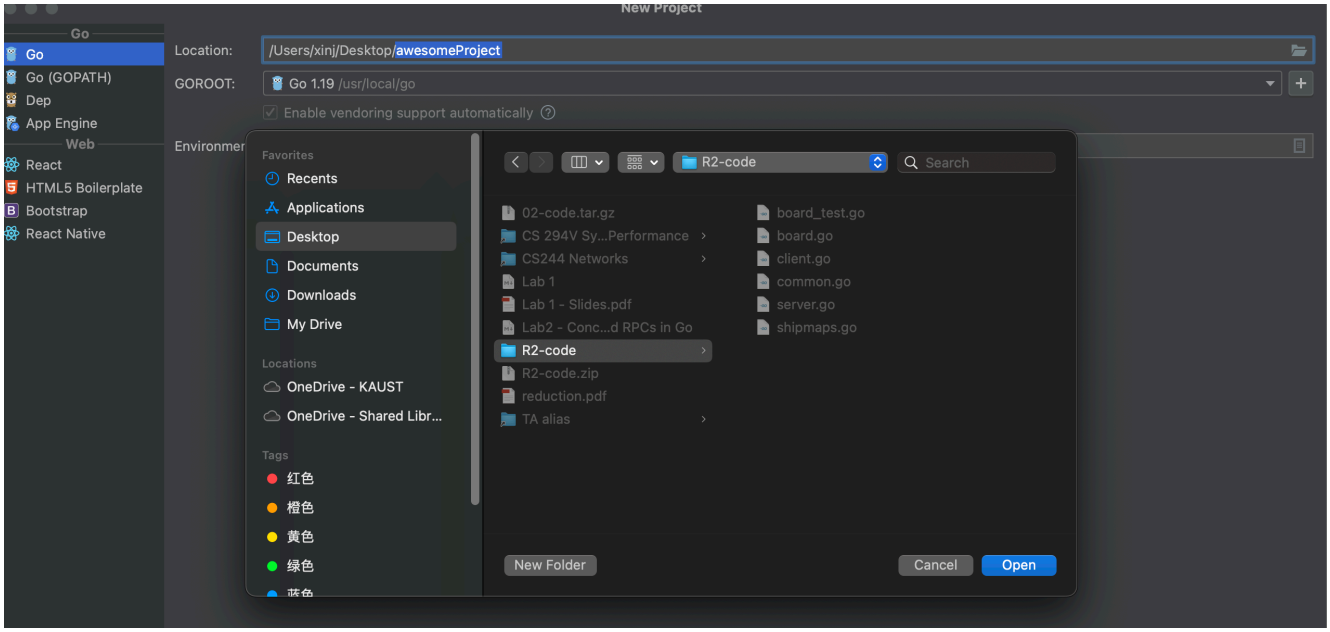
- Run `server.go`
- Run `client.go` and you should get the output

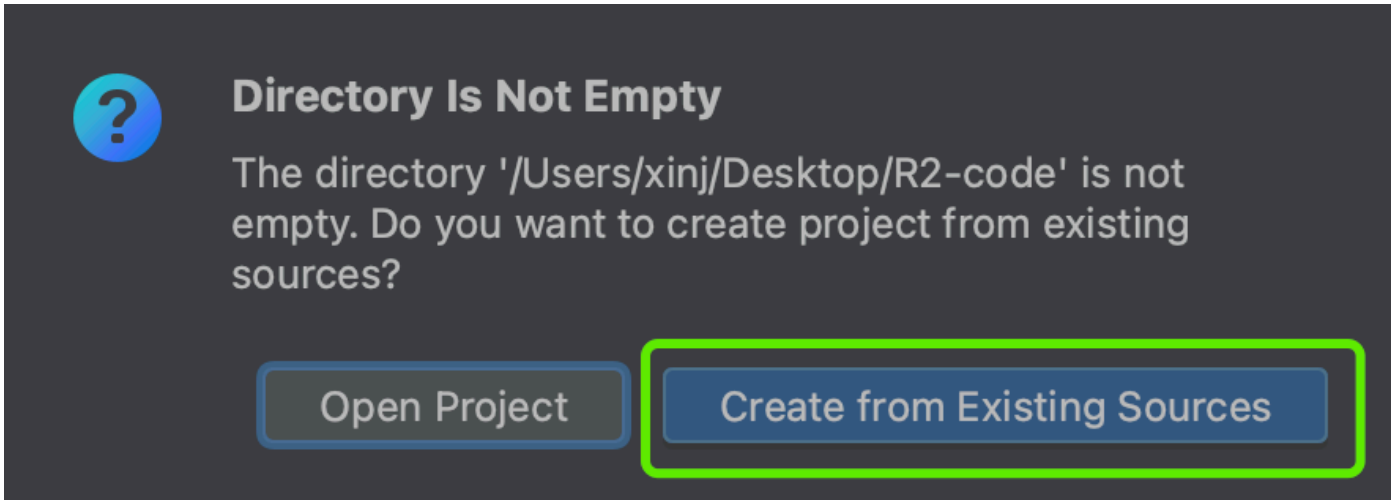
```
hello:hello
```

Battleship

In this section, we will inplement a **Battleship** game by RPC.

- The source code is on course syllabus.
- Create project from unziped folder





- The files are by default in package **main**, they depend on others, we cannot simply run by clicking "Run".

  For simplicity, we will use **CLI** to run code, but it is also encouraged to modify the package structure after class to deal with dependencies.

- Launch the `server`

  ```
  go run server.go common.go shipmaps.go
  ```

  I will launch the server so you actually don't need to do this.

- Launch `client`

```
# go run client.go common.go board.go <game name> <player_name> <server_address>
<server_port> <client_port>
go run client.go common.go board.go JihaoGame Jihao1 localhost 6001 7001
```

Now the code is incomplete, your task is to implement below logics in `client.go` (not necessary to touch other files).

- Taks 1: Establish connection to the server
  - See https://golang.org/pkg/net/rpc/ example “rpc.DialHTTP”
  - Must return a rpc.Client object
- Task 2: Make the JoinGame request
  - You want to call the remote BattleshipsService.JoinGame function
  - Parameters PublicPlayer and JoinGameRequest are defined in common.go
  - See https://golang.org/pkg/net/rpc/ example “client.Call”
- Task 3: Implement the attack server
  - Tasks 1 and 2 were making requests as a client, now must accept requests
  - See https://golang.org/pkg/net/rpc/

    Examples “rpc.Register” and “rpc.HandleHTTP”
  - Create a listener to serve requests on a separate goroutine
- Task 4: Implement the turn logic
  - Hint: The turn logic can be achieved with Channels, Locks or WaitGroups
  - Hint 2: When the other player attacks, you get a “token” to make one attack