RUNTIME GENERATED, TYPESAFE, AND DECLARATIVE

PICK ANY THREE

JON BODNER
CAPITAL ONE
JULY 14TH, 2017

WHEN GEEKS GET TOGETHER

WE BRAG ABOUT LANGUAGE FEATURES

```clojure
(let [my-vector [1 2 3 4]
      my-map {:fred "ethel"}
      my-list (list 4 3 2 1)]
  (list
    (conj my-vector 5)
    (assoc my-map :ricky "lucy")
    (conj my-list 5)
    ;the originals are intact
    my-vector
    my-map
    my-list))
```

WHEN GEEKS GET TOGETHER

WE BRAG ABOUT LANGUAGE FEATURES

```
case class ListNode[+T](h: T, t: ListNode[T]) {
    def head: T = h
    def tail: ListNode[T] = t
    def prepend[U >: T](elem: U): ListNode[U] =
      ListNode(elem, this)
}
```

WHEN GEEKS GET TOGETHER

WE BRAG ABOUT LANGUAGE FEATURES

```
life←{↑1 ω∨.∧3 4=+/,¯1 0 1∘.⊖¯1 0 1∘.⌽⊂ω}
```

ONE OF THE BEST FEATURES OF GO IS ITS LACK OF FEATURES

JDBI: JAVA DAOS DONE BETTER

```
public interface MyDAO
{
  @SqlUpdate("create table something (id int primary key, name varchar(100))")
  void createSomethingTable();

  @SqlUpdate("insert into something (id, name) values (:id, :name)")
  void insert(@Bind("id") int id, @Bind("name") String name);

  @SqlQuery("select id, name from something where id = :id")
  Something findById(@Bind("id") int id);

  @SqlQuery("select id, name from something where id > :from and id < :to")
  List<Something> findBetween(@Bind("from") int from, @Bind("to") int to);
}
```

JDBI: JAVA DAOS DONE BETTER

```java
DBI dbi = new DBI("jdbc:h2:mem:test");
Handle h = dbi.open();
MyDAO dao = h.attach(MyDAO.class);

// do stuff with the dao
dao.createSomethingTable();

dao.insert(10,"Fred");

Something something = dao.findNameById(10);

List<Something> somethings = dao.findBetween(5,15);

h.close();
```

REQUIREMENTS

- Specify the queries inline with the code.
- Generate the code to implement the queries.
- Ensure type safety.
- Prevent SQL injection attacks.
- Make sure performance is reasonable.
- It needs to feel like Go.

PROTEUS

```go
var personDao PersonDao

func init() {
  err := Build(&personDao, Postgres)
  if err != nil {
    panic(err)
  }
}

func main() {
  db := setupDbPostgres()
  wrapper := Adapt(db)
  DoPersonStuff(wrapper)
}
```

```go
func DoPersonStuff(wrapper Wrapper) {
 count, err := personDao.Create(wrapper, "Fred", 20)
 fmt.Println("create:", count, err)

 person, err := personDao.Get(wrapper, 1)
 fmt.Println("get:", person, err)

 people, err := personDao.GetAll(wrapper)
 fmt.Println("get all:", people, err)

 count, err = personDao.Update(wrapper, 1, "Freddie", 30)
 fmt.Println("update:", count, err)

 person, err = personDao.Get(wrapper, 1)
 fmt.Println("get #2:", person, err)

 count, err = personDao.Delete(wrapper, 1)
 fmt.Println("delete:", count, err)
}
```

REFLECTION: FINDING YOUR TYPE

- Getting the type of a variable

```
varType := reflect.TypeOf(var)
```

- Getting the kind of a variable
(slice, map, pointer, struct, interface, string, etc.)

```
varKind := varType.Kind()
```

- Getting the type of a pointer, map, slice, channel, or array

```
underlyingType := varType.Elem()
```

- Creating a type token (an io.Reader in this case)

```
token := reflect.TypeOf((*io.Reader)(nil)).Elem()
```

REFLECTION: FINDING YOUR VALUES

- Creating a reflect.Value (use a pointer if you want to modify it)

```
var := 10
var2 := 20
varVal := reflect.ValueOf(&var)
varVal2 := reflect.ValueOf(var2)
```

- Reading the value of a pointer

```
varVal.Elem().Interface()
varVal2.Interface()
```

- Setting the value of a pointer (need to use a reflect.Value)

```
varVal.Elem().Set(varVal2)
```

- Creating a new pointer value (so you can modify it)

```
newVarVal := reflect.New(varVal2.Type())
```

PROBLEM #1: HOW DO I STORE METADATA?

STRUCT TAGS!

```go
func TagPrinter(s interface{}) {
  t := reflect.TypeOf(s)
  if t.Kind() != reflect.Struct {
    return
  }
  for i := 0; i < t.NumField(); i++ {
    f := t.Field(i)
    fmt.Printf("Field %s has %s for tag tag1 and %s for tag tag2\n",
               f.Name, f.Tag.Get("tag1"), f.Tag.Get("tag2"))
  }
}
```

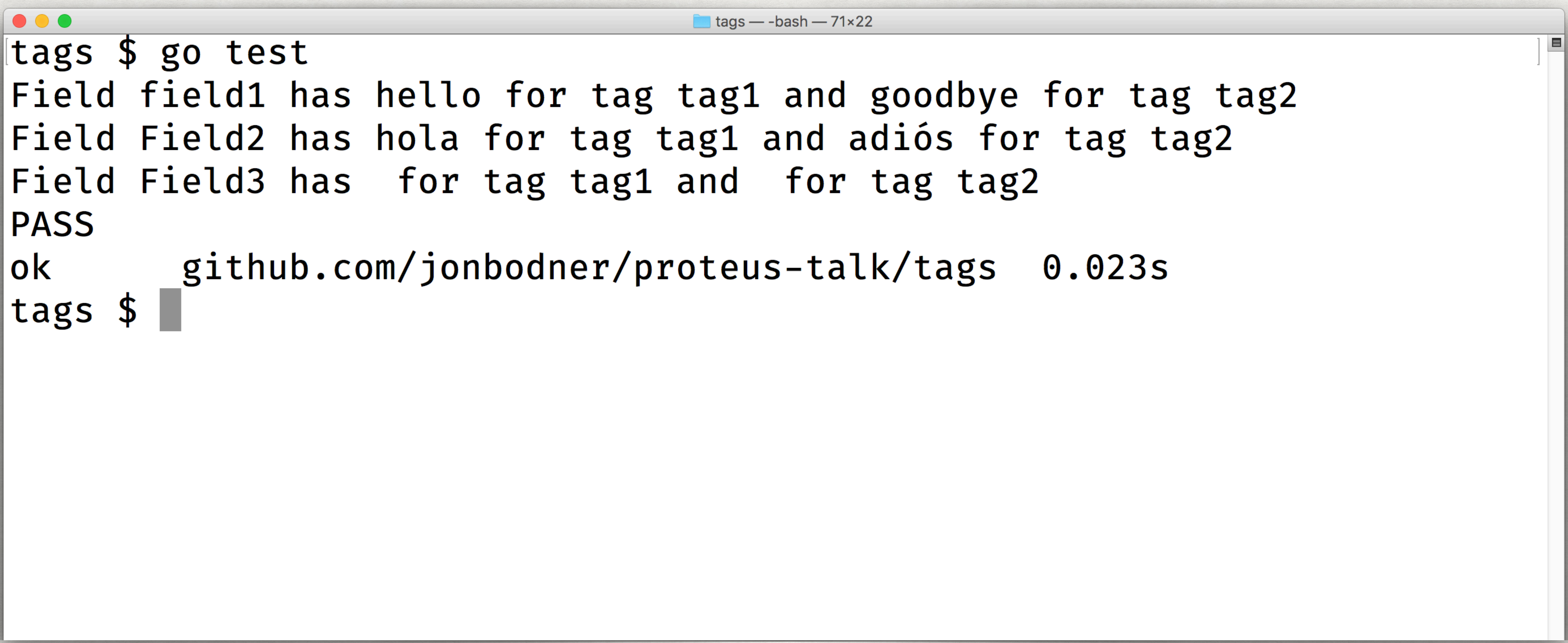
PROBLEM #1: HOW DO I STORE METADATA?

STRUCT TAGS!

```
type S struct {
  field1 int `tag1:"hello" tag2:"goodbye"`
  Field2 int `tag1:"hola" tag2:"adiós"`
  Field3 string
}
```

PROBLEM #1: HOW DO I STORE METADATA?

STRUCT TAGS!



```
tags $ go test
Field field1 has hello for tag tag1 and goodbye for tag tag2
Field Field2 has hola for tag tag1 and adiós for tag tag2
Field Field3 has  for tag tag1 and  for tag tag2
PASS
ok  	github.com/jonbodner/proteus-talk/tags	0.023s
tags $ 
```

PROBLEM #2: HOW DO YOU RUN A STRING?

GENERATE FUNCTIONS AT RUNTIME!

```
type Calculator func(a, b int) int
```

PROBLEM #2: HOW DO YOU RUN A STRING?

GENERATE FUNCTIONS AT RUNTIME!

This generated a function!

```
func MemoizeCalculator(c Calculator) Calculator {
  t := reflect.TypeOf(c)
  type vals struct {
    a,b int
  }
  cache := map[vals]int{}
  v := reflect.MakeFunc(t, func(args []reflect.Value) []reflect.Value {
    a := args[0].Interface().(int)
    b := args[1].Interface().(int)
    v := vals{a:a, b:b}
    result, ok := cache[v]
    if !ok {
      result = c(a,b)
      cache[v] = result
    }
    return []reflect.Value{reflect.ValueOf(result)}
  })
  return v.Interface().(Calculator)
}
```

PROBLEM #2: HOW DO YOU RUN A STRING?

GENERATE FUNCTIONS AT RUNTIME!

```
func AddSlowly(a, b int) int {
 time.Sleep(1 * time.Second)
 return a + b
}

func AddNormally(a, b int) int {
 return a + b
}
```

I LOVE IT WHEN A PLAN COMES TOGETHER

- Little known fact: struct fields can be of type function
- Define a struct that has function fields
- Put struct tags with SQL on those function fields
- At runtime
    - Pass an instance of the struct into Build function
    - Build dynamically generates functions for struct's fields

STRUCT FIELDS CAN BE FUNCTIONS

THIS IS NOT AN INTERFACE IMPLEMENTATION…

```
type PersonDao struct {
  Create func(/*Parameters*/) `proq:"CREATE SQL QUERY GOES HERE"`
  Get    func(/*Parameters*/) `proq:"READ SQL QUERY GOES HERE"`
  Update func(/*Parameters*/) `proq:"UPDATE SQL QUERY GOES HERE"`
  Delete func(/*Parameters*/) `proq:"DELETE SQL QUERY GOES HERE"`
}

var personDao PersonDao

func init() {
  err := Build(&personDao)
  if err != nil {
    panic(err)
  }
}
```

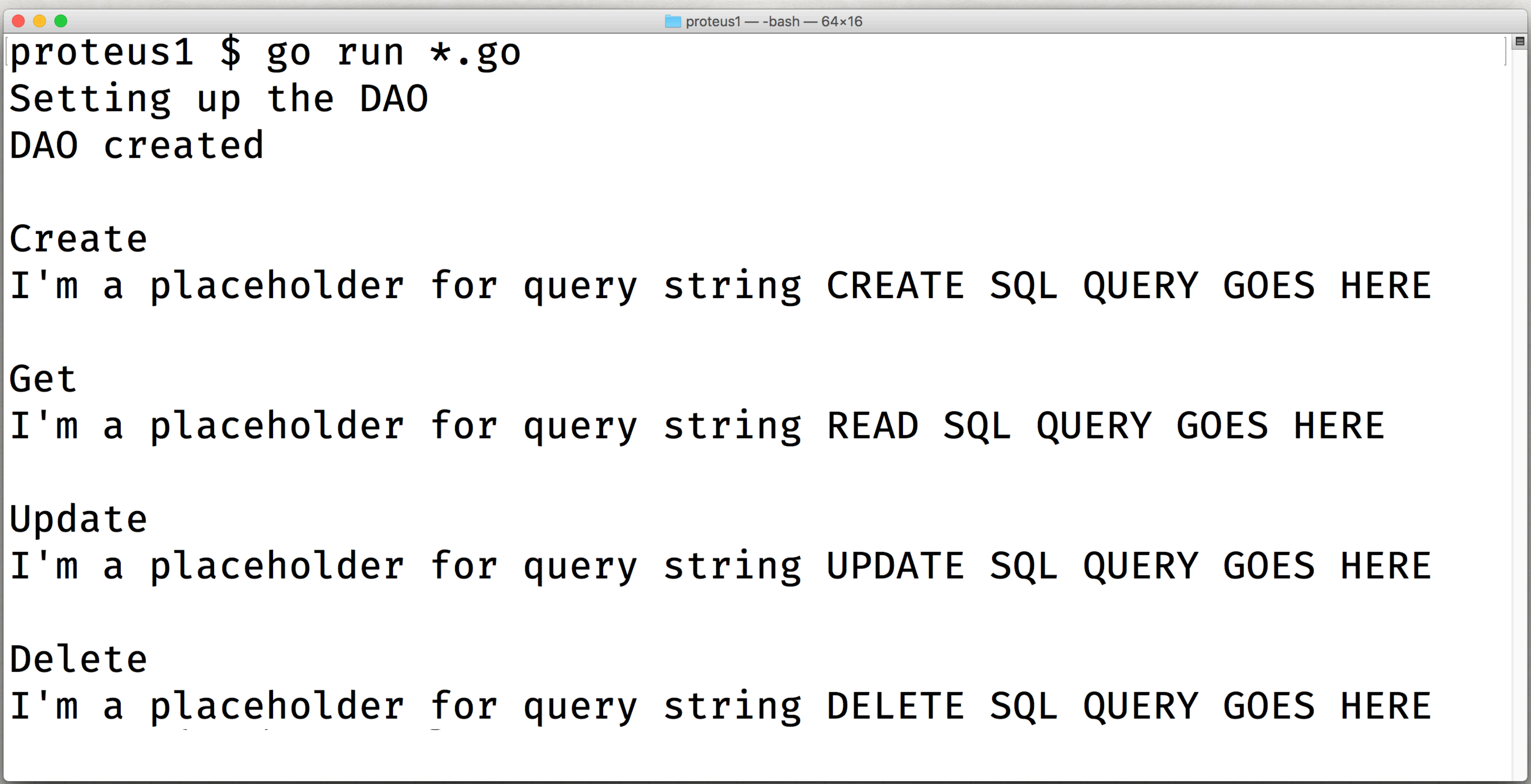
STRUCT FIELDS CAN BE FUNCTIONS

...BUT IT SURE LOOKS LIKE IT

```
func DoPersonStuff() {
 personDao.Create()

 personDao.Get()

 personDao.Update()

 personDao.Delete()
}


func main() {
 DoPersonStuff()
}
```

```go
func Build(dao interface{}) error {
 daoPointerType := reflect.TypeOf(dao)

 //must be a pointer to struct
 if daoPointerType.Kind() != reflect.Ptr {
   return errors.New("Not a pointer")
 }

 daoType := daoPointerType.Elem()

 //if not a struct, error out
 if daoType.Kind() != reflect.Struct {
   return errors.New("Not a pointer to struct")
 }

 daoPointerValue := reflect.ValueOf(dao)
 daoValue := reflect.Elem(daoPointerValue)
 for i := 0; i < daoType.NumField(); i++ {
   curField := daoType.Field(i)
   query, ok := curField.Tag.Lookup("proq")
   if curField.Type.Kind() != reflect.Func || !ok {
     continue
   }
   funcType := curField.Type

   implementation, err := makeImplementation(funcType, query)
   if err != nil {
     continue
   }

   fieldValue := daoValue.Field(i)
   fieldValue.Set(reflect.MakeFunc(funcType, implementation))
 }

 return nil
}
```

```go
func makeImplementation(funcType reflect.Type, query string)
                        (func([]reflect.Value) []reflect.Value, error) {
  return func(args []reflect.Value) []reflect.Value {
    fmt.Printf("I'm a placeholder for query string %s\n", query)
    return nil
  }, nil
}
```



```
proteus1 $ go run *.go
Setting up the DAO
DAO created

Create
I'm a placeholder for query string CREATE SQL QUERY GOES HERE

Get
I'm a placeholder for query string READ SQL QUERY GOES HERE

Update
I'm a placeholder for query string UPDATE SQL QUERY GOES HERE

Delete
I'm a placeholder for query string DELETE SQL QUERY GOES HERE
```

INTEGRATING WITH THE GO SQL LIBRARIES

REUSE, DON'T REWRITE

- sql.DB and sql.TX
  - `Query(query string, args ...interface{}) (sql.Rows, error)`
  - `Exec(query string, args ...interface{}) (sql.Result, error)`
- New interfaces: Querier, Executor, Wrapper, and Rows
- Problem: returning sql.Rows != returning Rows
  - Solution: small adapter struct and factory function

```go
// Querier runs queries that return Rows from the data store
type Querier interface {
  Query(query string, args ...interface{}) (Rows, error)
}

// Executor runs queries that modify the data store.
type Executor interface {
  Exec(query string, args ...interface{}) (sql.Result, error)
}

type Wrapper interface {
  Executor
  Querier
}
```

```
type Rows interface {
  Next() bool

  Err() error

  Columns() ([]string, error)

  Scan(dest ...interface{}) error

  Close() error
}
```

```go
func Adapt(sqle Sql) Wrapper {
  return sqlAdapter{sqle}
}

type sqlAdapter struct {
  Sql
}

func (w sqlAdapter) Exec(query string, args ...interface{})
                        (sql.Result, error) {
  return w.Sql.Exec(query, args...)
}

func (w sqlAdapter) Query(query string, args ...interface{})
                         (Rows, error) {
  return w.Sql.Query(query, args...)
}
```

```go
// Sql matches the interface provided
// by several types in the standard go sql package.
type Sql interface {
  Exec(query string, args ...interface{}) (sql.Result, error)

  Query(query string, args ...interface{}) (*sql.Rows, error)
}
```

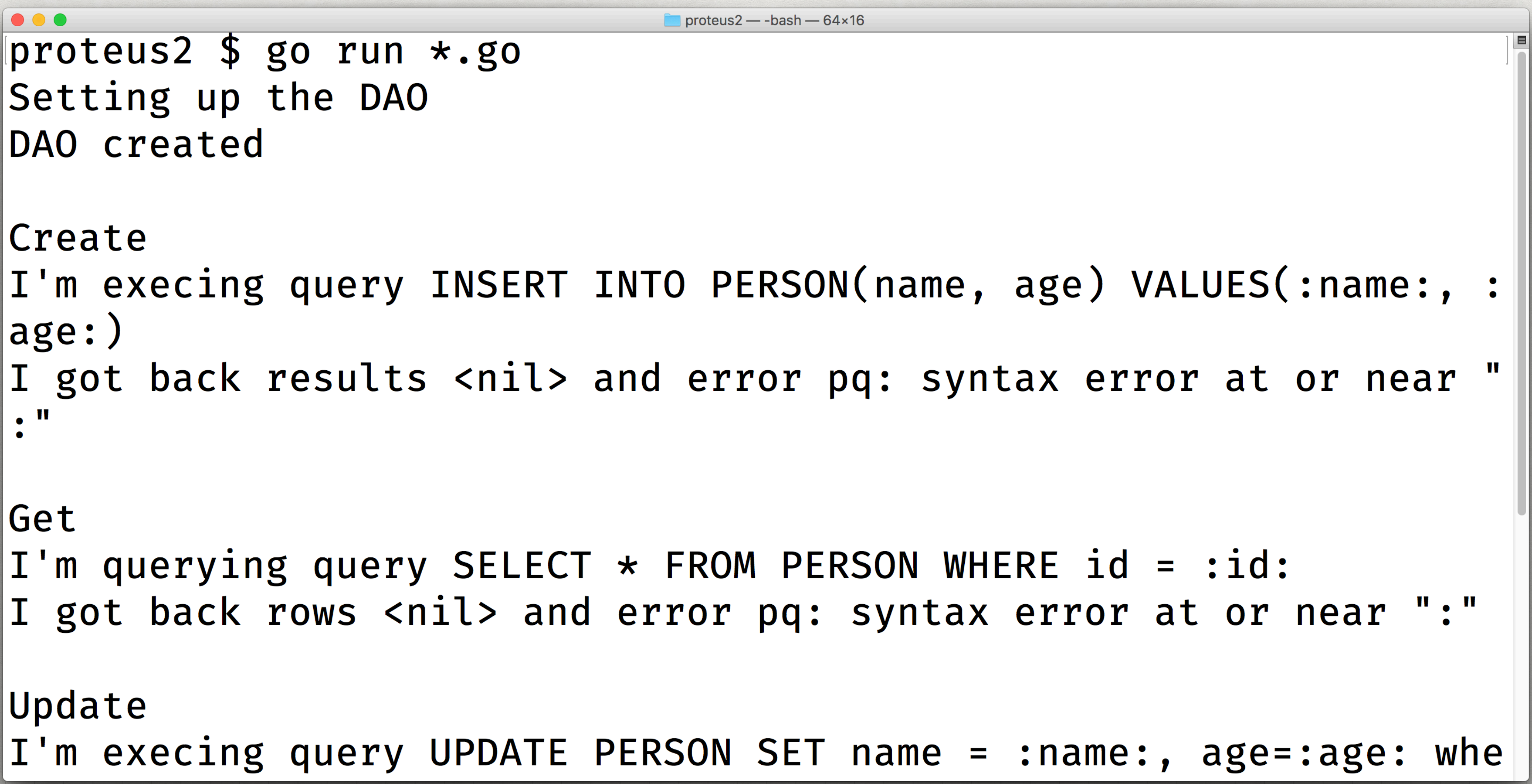
```go
var exType = reflect.TypeOf((*Executor)(nil)).Elem()
var qType = reflect.TypeOf((*Querier)(nil)).Elem()

func makeImplementation(funcType reflect.Type, query string)
                        (func([]reflect.Value) []reflect.Value, error) {
  if funcType.NumIn() == 0 {
    return nil, errors.New(
                         "need to supply an Executor or Querier parameter")
  }
  switch fType := funcType.In(0); {
  case fType.Implements(exType):
    return makeExecutorImplementation(funcType, query)
  case fType.Implements(qType):
    return makeQuerierImplementation(funcType, query)
  default:
    return nil, errors.New(
         "first parameter must be of type api.Executor or api.Querier")
  }
}
```

```
func makeExecutorImplementation(funcType reflect.Type, query string)
                              (func([]reflect.Value) []reflect.Value, error) {
  return func(args []reflect.Value) []reflect.Value {
    executor := args[0].Interface().(Executor)

    fmt.Println("I'm execing query",query)
    result, err := executor.Exec(query/*args are coming soon*/)
    fmt.Println("I got back results", result, "and error",err)

    return nil
  }, nil
}
```

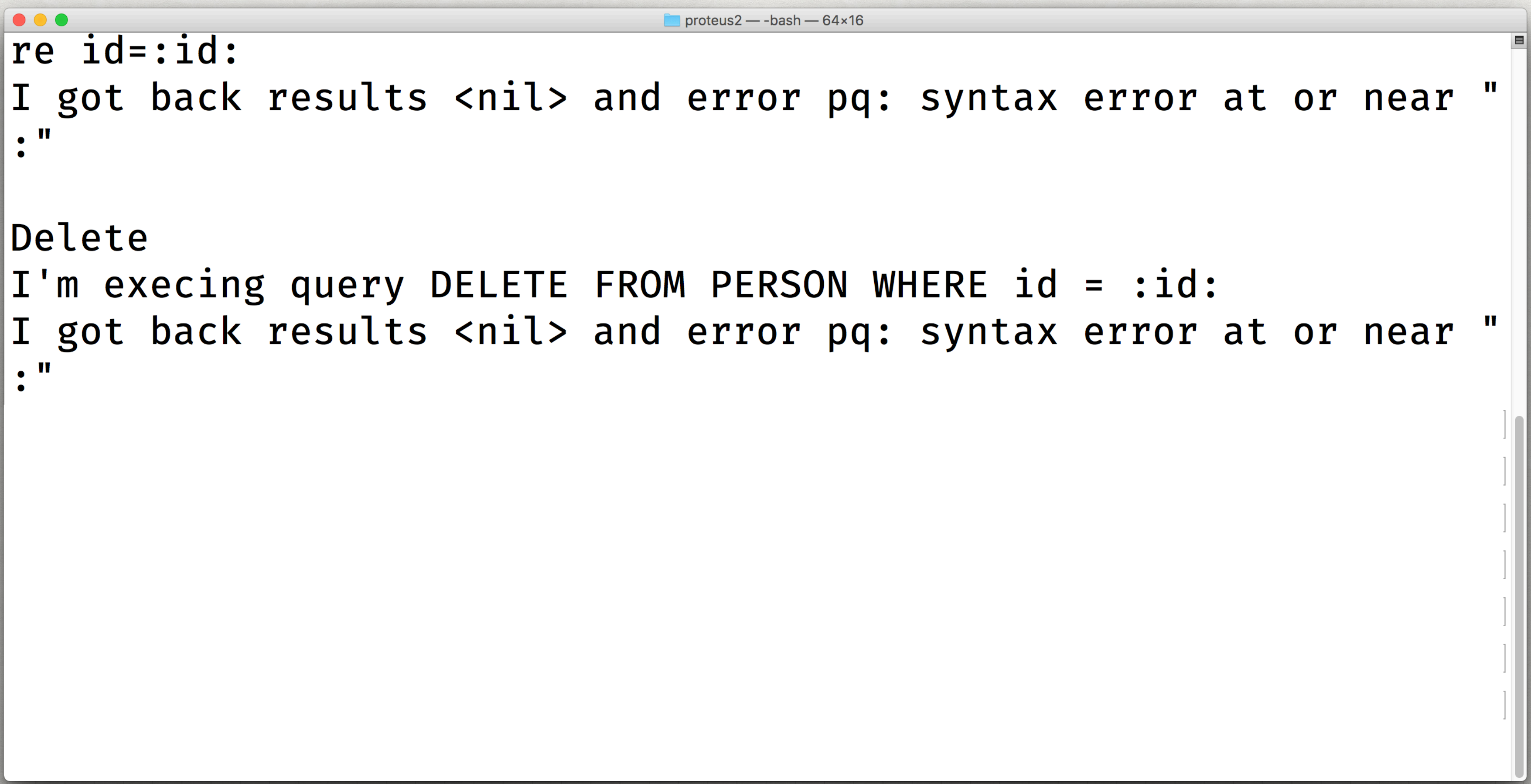
```go
func makeQuerierImplementation(funcType reflect.Type, query string)
                                 (func([]reflect.Value) []reflect.Value, error) {
  return func(args []reflect.Value) []reflect.Value {
    querier := args[0].Interface().(Querier)

    fmt.Println("I'm querying query",query)
    rows, err := querier.Query(query/*args are coming soon*/)
    fmt.Println("I got back rows", rows, "and error",err)

    return nil
  }, nil
}
```

```
type PersonDao struct {
  Create func(Executor /*Parameters*/)
   `proq:"INSERT INTO PERSON(name, age) VALUES(:name:, :age:)"`


   Get func(Querier /*Parameters*/)
   `proq:"SELECT * FROM PERSON WHERE id = :id:"`


  Update func(Executor /*Parameters*/)
   `proq:"UPDATE PERSON SET name = :name:, age=:age: where id=:id:"`


  Delete func(Executor /*Parameters*/)
   `proq:"DELETE FROM PERSON WHERE id = :id:"`
}
```

```
func DoPersonStuff(wrapper Wrapper) {
  personDao.Create(wrapper)

  personDao.Get(wrapper)

  personDao.Update(wrapper)

  personDao.Delete(wrapper)
}



func main() {
  db := setupDbPostgres()
  wrapper := Adapt(db)

  DoPersonStuff(wrapper)
}
```



```
proteus2 $ go run *.go
Setting up the DAO
DAO created

Create
I'm execing query INSERT INTO PERSON(name, age) VALUES(:name:, :
age:)
I got back results <nil> and error pq: syntax error at or near "
:"

Get
I'm querying query SELECT * FROM PERSON WHERE id = :id:
I got back rows <nil> and error pq: syntax error at or near ":"

Update
I'm execing query UPDATE PERSON SET name = :name:, age=:age: whe
```


proteus2 — -bash — 64×16

```
re id=:id:
I got back results <nil> and error pq: syntax error at or near "
:"

Delete
I'm execing query DELETE FROM PERSON WHERE id = :id:
I got back results <nil> and error pq: syntax error at or near "
:"
```

BUILDING SQL QUERIES

LET'S BE TYPESAFE OUT THERE

- Need to map function parameters to query placeholders
  - Function: `Update func(e Executor, id int, name string, age int)`
  - Query: `UPDATE PERSON SET name = :name:, age=:age: where id=:id:`
- Go doesn't save parameter names!
  - Use a second struct tag key/value pair to map the names to their position
    - `prop:"id,name,age"`
- Different databases use different parameter notation
  - Use a ParamAdapter to go from `:name:` to `$1`
  - ParamAdapter implementation is passed to Build

```go
type ParamAdapter func(pos int) string

func MySQL(pos int) string {
  return "?"
}

func Sqlite(pos int) string {
  return "?"
}

func Postgres(pos int) string {
  return fmt.Sprintf("$%d", pos)
}

func Oracle(pos int) string {
  return fmt.Sprintf(":%d", pos)
}
```

FROM STRUCT TAGS TO SQL

STEP BY STEP

Struct Tag proq: `UPDATE PERSON SET name = :name:, age=:age: where id=:id:`

Struct Tag prop: `id,name,age`

FROM STRUCT TAGS TO SQL

CONVERT PROP TO MAP[STRING]INT

Struct Tag proq: `UPDATE PERSON SET name = :name:, age=:age: where id=:id:`

nameOrderMap: `{“id”:1, “name”:2, “age”:3}`

FROM STRUCT TAGS TO SQL

CONVERT PROQ AND NAMEORDERMAP TO QUERY AND PARAMORDER

SQL Query: `UPDATE PERSON SET name = $1, age=$2 where id=$3`

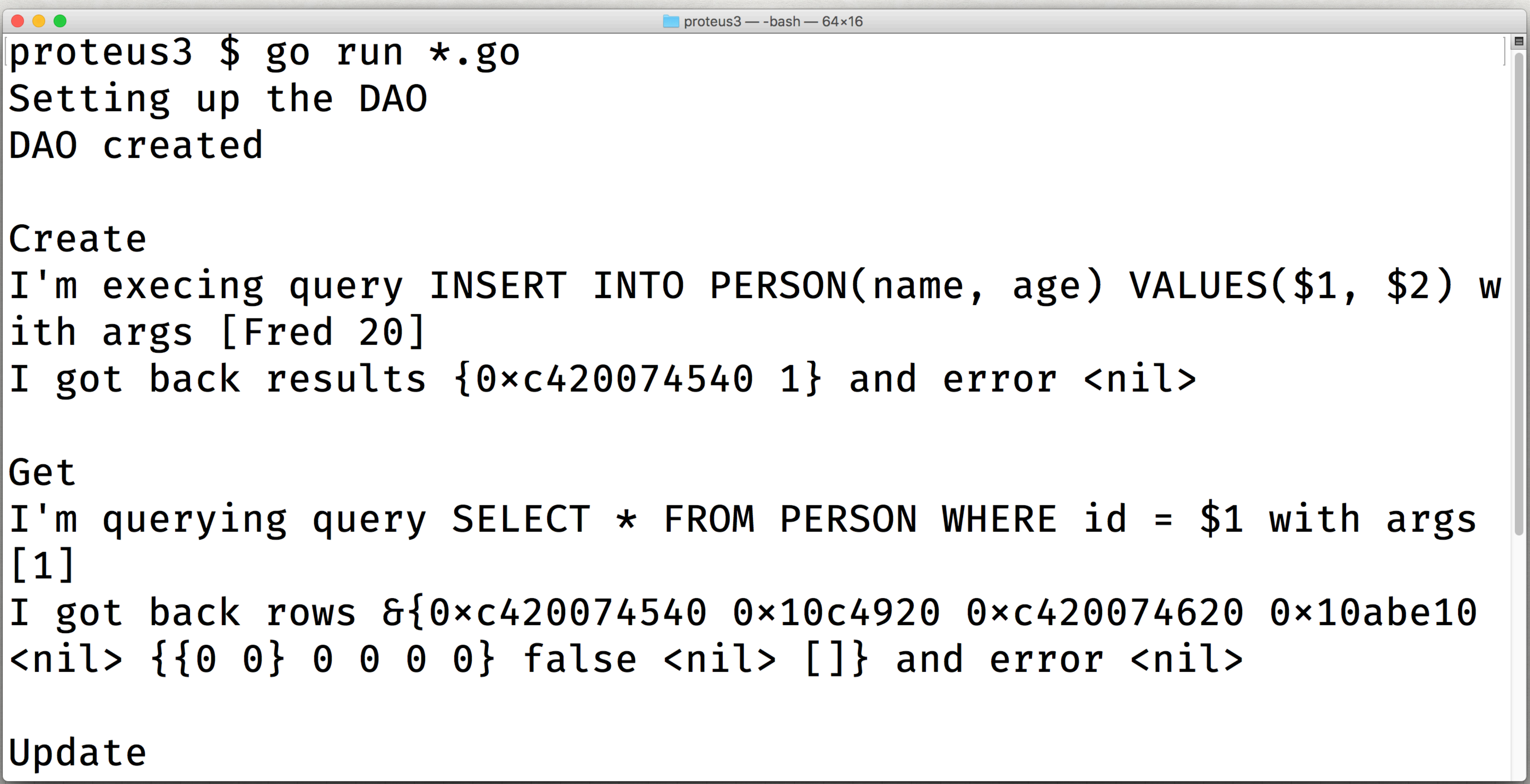
paramOrder: `[]paramInfo{{“name”,2}, {“age”,3}, {“id”,1}}`

```
func makeExecutorImplementation(
            funcType reflect.Type, query string, paramOrder []paramInfo)
            (func([]reflect.Value) []reflect.Value, error) {
  return func(args []reflect.Value) []reflect.Value {
    executor := args[0].Interface().(Executor)

    queryArgs := buildQueryArgs(args, paramOrder)

    fmt.Println("I'm execing query", query, "with args", queryArgs)
    result, err := executor.Exec(query, queryArgs...)
    fmt.Println("I got back results", result, "and error", err)

    return nil
  }, nil
}
```

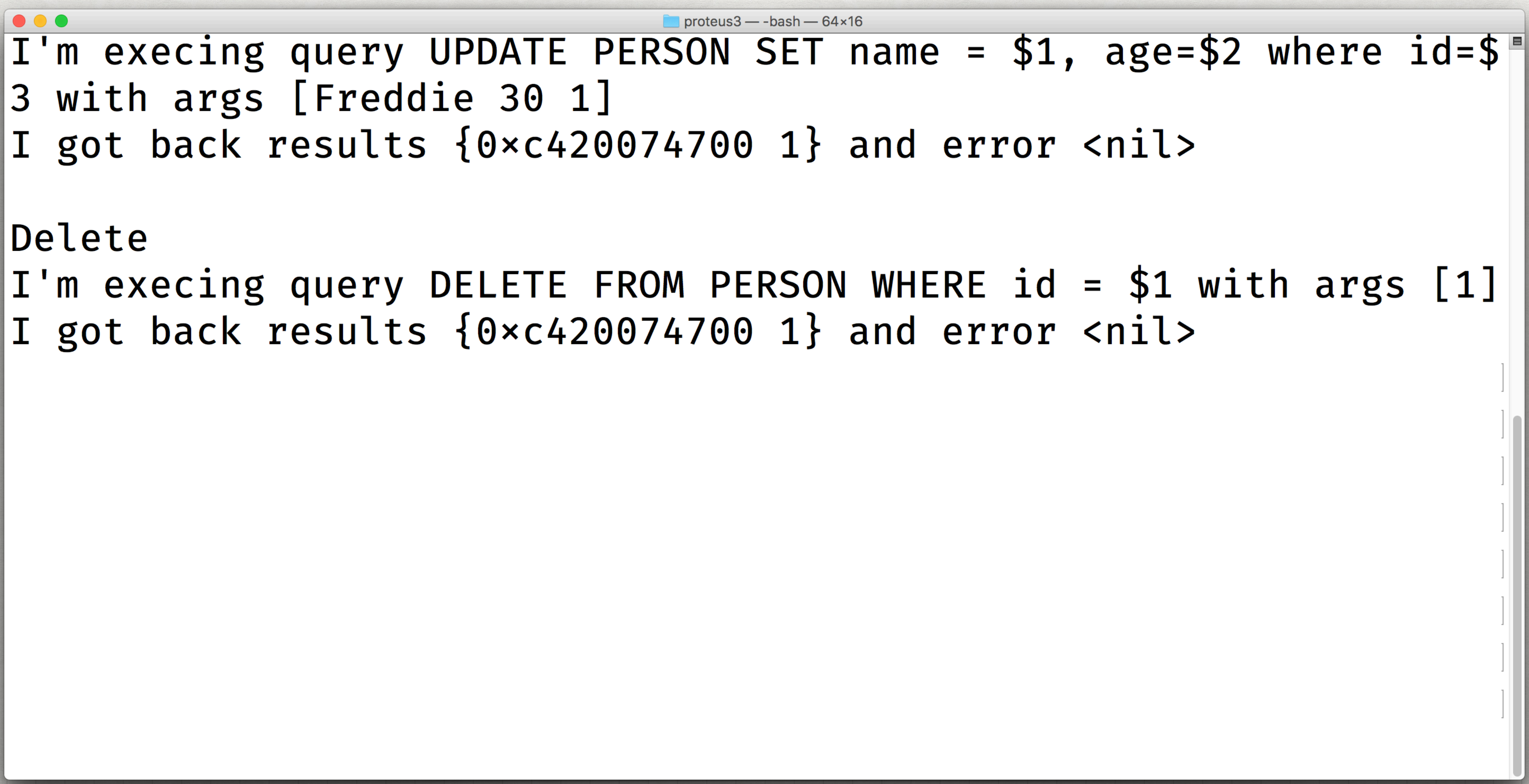
```go
func buildQueryArgs(funcArgs []reflect.Value, paramOrder []paramInfo)
                    []interface{} {
  out := []interface{}{}
  for _, v := range paramOrder {
    out = append(out, funcArgs[v.posInParams].Interface())
  }
  return out
}
```

```
type PersonDao struct {
  Create func(e Executor, name string, age int)
   `proq:"INSERT INTO PERSON(name, age) VALUES(:name:, :age:)"
    prop:"name,age"`


  Get func(q Querier, id int)
   `proq:"SELECT * FROM PERSON WHERE id = :id:"
    prop:"id"`

  Update func(e Executor, id int, name string, age int)
   `proq:"UPDATE PERSON SET name = :name:, age=:age: where id=:id:"
    prop:"id,name,age"`


  Delete func(e Executor, id int)
   `proq:"DELETE FROM PERSON WHERE id = :id:"
    prop:"id"`
}
```

```go
func init() {
  err := Build(&personDao, Postgres)
  if err != nil {
    panic(err)
  }
}

func DoPersonStuff(wrapper Wrapper) {
  personDao.Create(wrapper, "Fred", 20)


  personDao.Get(wrapper, 1)


  personDao.Update(wrapper, 1, "Freddie", 30)


  personDao.Delete(wrapper, 1)
}
```



```
proteus3 $ go run *.go
Setting up the DAO
DAO created

Create
I'm execing query INSERT INTO PERSON(name, age) VALUES($1, $2) w
ith args [Fred 20]
I got back results {0xc420074540 1} and error <nil>

Get
I'm querying query SELECT * FROM PERSON WHERE id = $1 with args
[1]
I got back rows &{0xc420074540 0x10c4920 0xc420074620 0x10abe10
<nil> {{0 0} 0 0 0 0} false <nil> []} and error <nil>

Update
```


proteus3 — -bash — 64×16

```
I'm execing query UPDATE PERSON SET name = $1, age=$2 where id=$
3 with args [Freddie 30 1]
I got back results {0xc420074700 1} and error <nil>

Delete
I'm execing query DELETE FROM PERSON WHERE id = $1 with args [1]
I got back results {0xc420074700 1} and error <nil>
```

RETURNING BACK VALUES

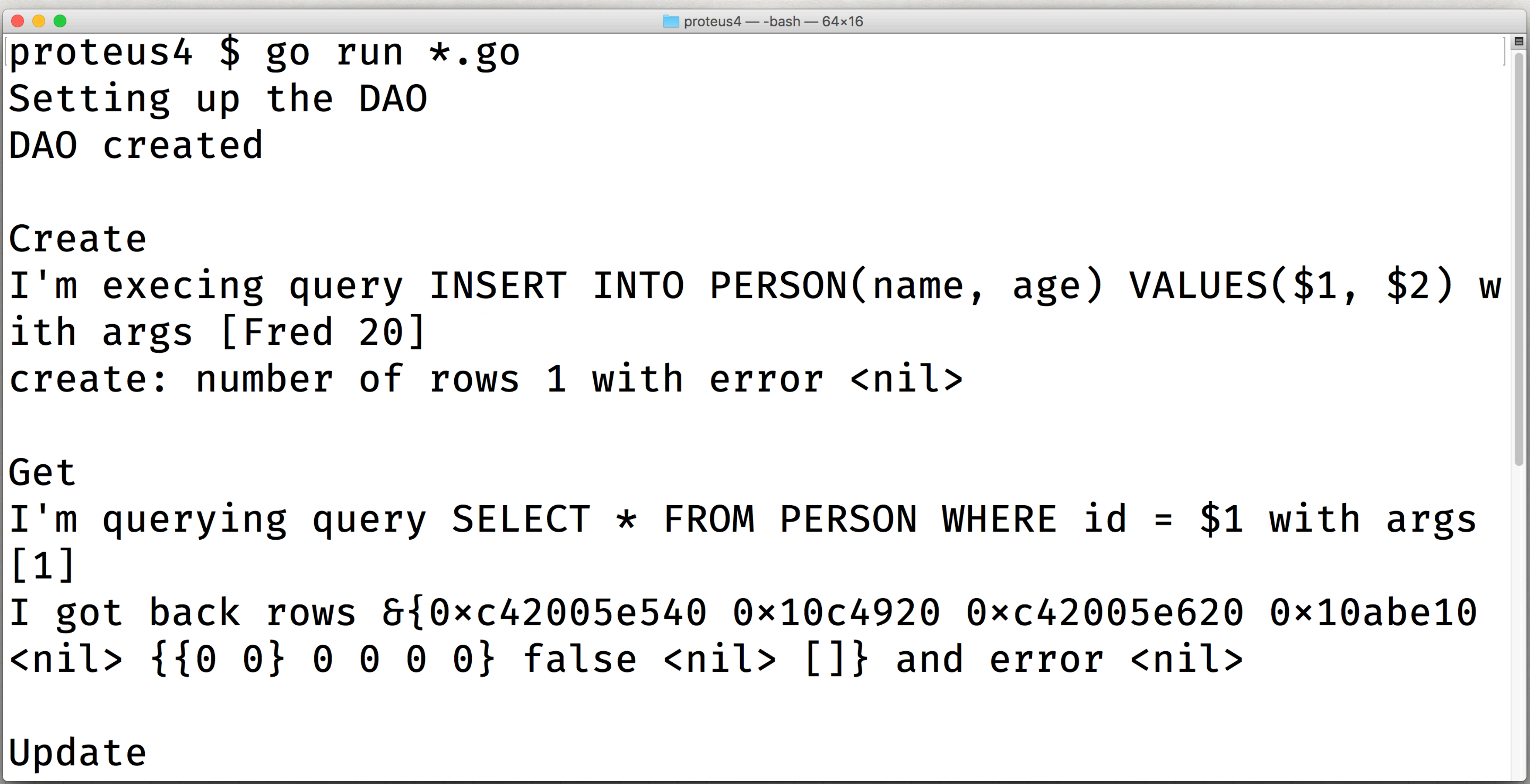
EXECUTORS FIRST

- Return type will be an int64 and an error
    - int64 is the number of rows affected
- Steps
    - Look at the sql.Result and error returned
    - Handle the error if necessary
    - Get the number of rows modified
    - Handle the error if necessary
    - Return the number of rows and a nil error

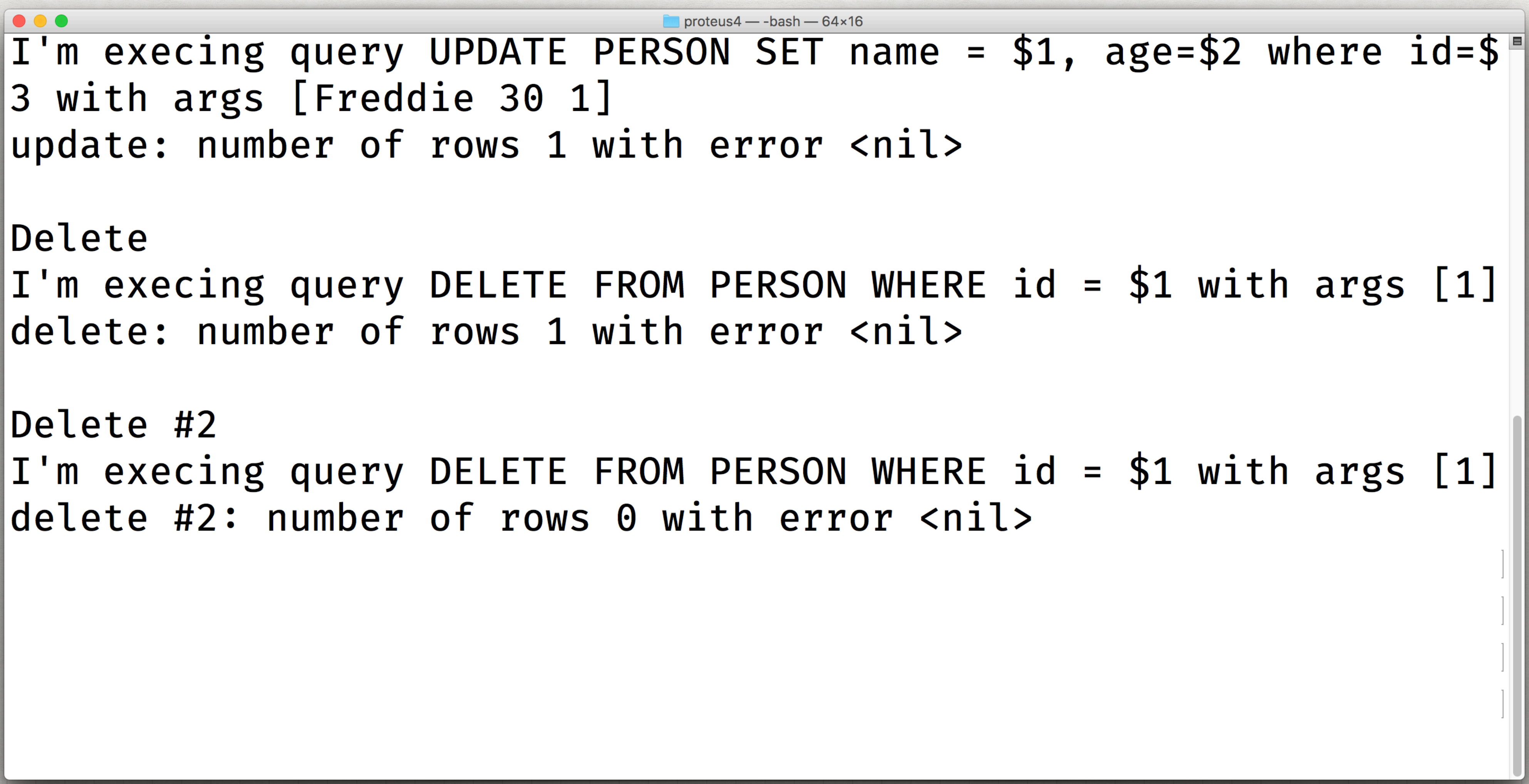
```go
var errType = reflect.TypeOf((*error)(nil)).Elem()
var errZero = reflect.Zero(errType)

func makeExecutorImplementation(funcType reflect.Type, query string, paramOrder []paramInfo)
                                (func([]reflect.Value) []reflect.Value, error) {
  return func(args []reflect.Value) []reflect.Value {
    executor := args[0].Interface().(Executor)

    queryArgs := buildQueryArgs(args, paramOrder)

    fmt.Println("I'm execing query", query, "with args", queryArgs)
    result, err := executor.Exec(query, queryArgs...)
    var count int64
    if err == nil {
      count, err = result.RowsAffected()
    }
    var errVal reflect.Value
    if err == nil {
      errVal = errZero
    } else {
      errVal = reflect.ValueOf(err).Convert(errType)
    }
    return []reflect.Value{reflect.ValueOf(count), errVal}
  }, nil
}
```

```go
type PersonDao struct {
  Create func(e Executor, name string, age int) (int64, error)
   `proq:"INSERT INTO PERSON(name, age) VALUES(:name:, :age:)"
    prop:"name,age"`


  Get func(q Querier, id int)
   `proq:"SELECT * FROM PERSON WHERE id = :id:"
    prop:"id"`

  Update func(e Executor, id int, name string, age int) (int64, error)
   `proq:"UPDATE PERSON SET name = :name:, age=:age: where id=:id:"
    prop:"id,name,age"`


  Delete func(e Executor, id int) (int64, error)
   `proq:"DELETE FROM PERSON WHERE id = :id:"
    prop:"id"`
}
```

```
func DoPersonStuff(wrapper Wrapper) {
  count, err := personDao.Create(wrapper, "Fred", 20)
  fmt.Println("create","number of rows", count, "with error", err)


  personDao.Get(wrapper, 1)


  count, err = personDao.Update(wrapper, 1, "Freddie", 30)
  fmt.Println("update:","number of rows", count, "with error", err)


  count, err = personDao.Delete(wrapper, 1)
  fmt.Println("delete:","number of rows", count, “with error", err)


  count, err = personDao.Delete(wrapper, 1)
  fmt.Println("delete #2:","number of rows", count, "with error", err)
}
```



```
proteus4 $ go run *.go
Setting up the DAO
DAO created

Create
I'm execing query INSERT INTO PERSON(name, age) VALUES($1, $2) w
ith args [Fred 20]
create: number of rows 1 with error <nil>

Get
I'm querying query SELECT * FROM PERSON WHERE id = $1 with args
[1]
I got back rows &{0xc42005e540 0x10c4920 0xc42005e620 0x10abe10
<nil> {{0 0} 0 0 0 0} false <nil> []} and error <nil>

Update
```



```
I'm execing query UPDATE PERSON SET name = $1, age=$2 where id=$
3 with args [Freddie 30 1]
update: number of rows 1 with error <nil>

Delete
I'm execing query DELETE FROM PERSON WHERE id = $1 with args [1]
delete: number of rows 1 with error <nil>

Delete #2
I'm execing query DELETE FROM PERSON WHERE id = $1 with args [1]
delete #2: number of rows 0 with error <nil>
```

RETURNING BACK VALUES

NOW DO QUERIERS

- Define a struct with struct tags to map fields to query results

```
type Person struct {
  Id   int `prof:"id"`
  Name string `prof:"name"`
  Age  int `prof:"age"`
}
```

- Return type will be a pointer to struct and an error

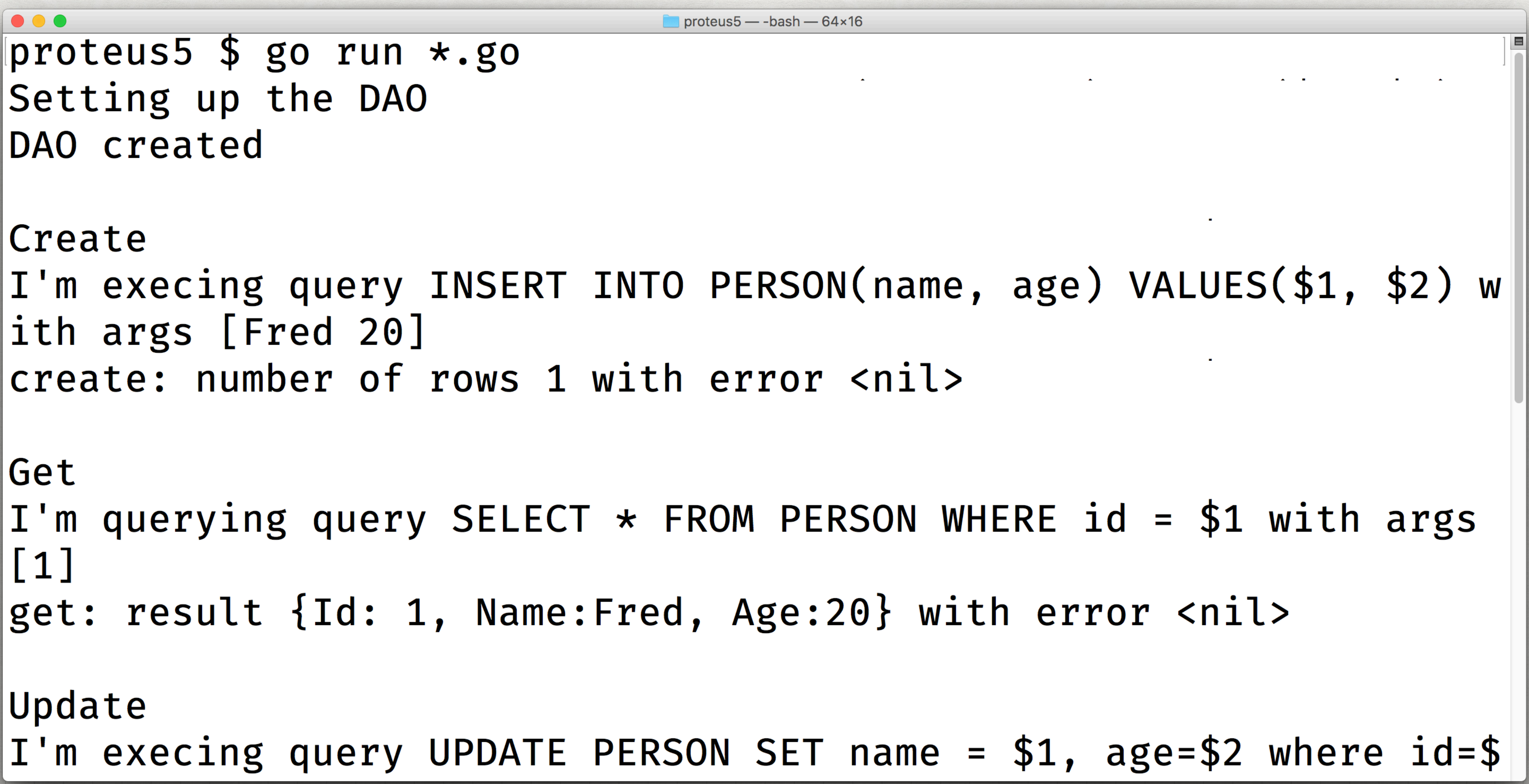
```
 Get    func(q Querier, id int) (*Person, error)
`proq:"SELECT * FROM PERSON WHERE id = :id:" prop:"id"`
```

```go
func makeQuerierImplementation(funcType reflect.Type, query string, paramOrder []paramInfo)
                              (func([]reflect.Value) []reflect.Value, error) {
  firstResult := funcType.Out(0)
  zeroVal := reflect.Zero(firstResult)
  returnType := firstResult.Elem()

  mapper := buildMapper(returnType, zeroVal)

  return func(args []reflect.Value) []reflect.Value {
    querier := args[0].Interface().(Querier)

    queryArgs := buildQueryArgs(args, paramOrder)
    fmt.Println("I'm querying query", query, "with args", queryArgs)
    rows, err := querier.Query(query, queryArgs...)

    if err != nil {
      return []reflect.Value{zeroVal, reflect.ValueOf(err).Convert(errType)}
    }

    result, err := mapOneRow(rows, mapper, zeroVal)
    rows.Close()

    if err != nil {
      return []reflect.Value{result, reflect.ValueOf(err).Convert(errType)}
    }

    return []reflect.Value{result, errZero}
  }, nil
}
```

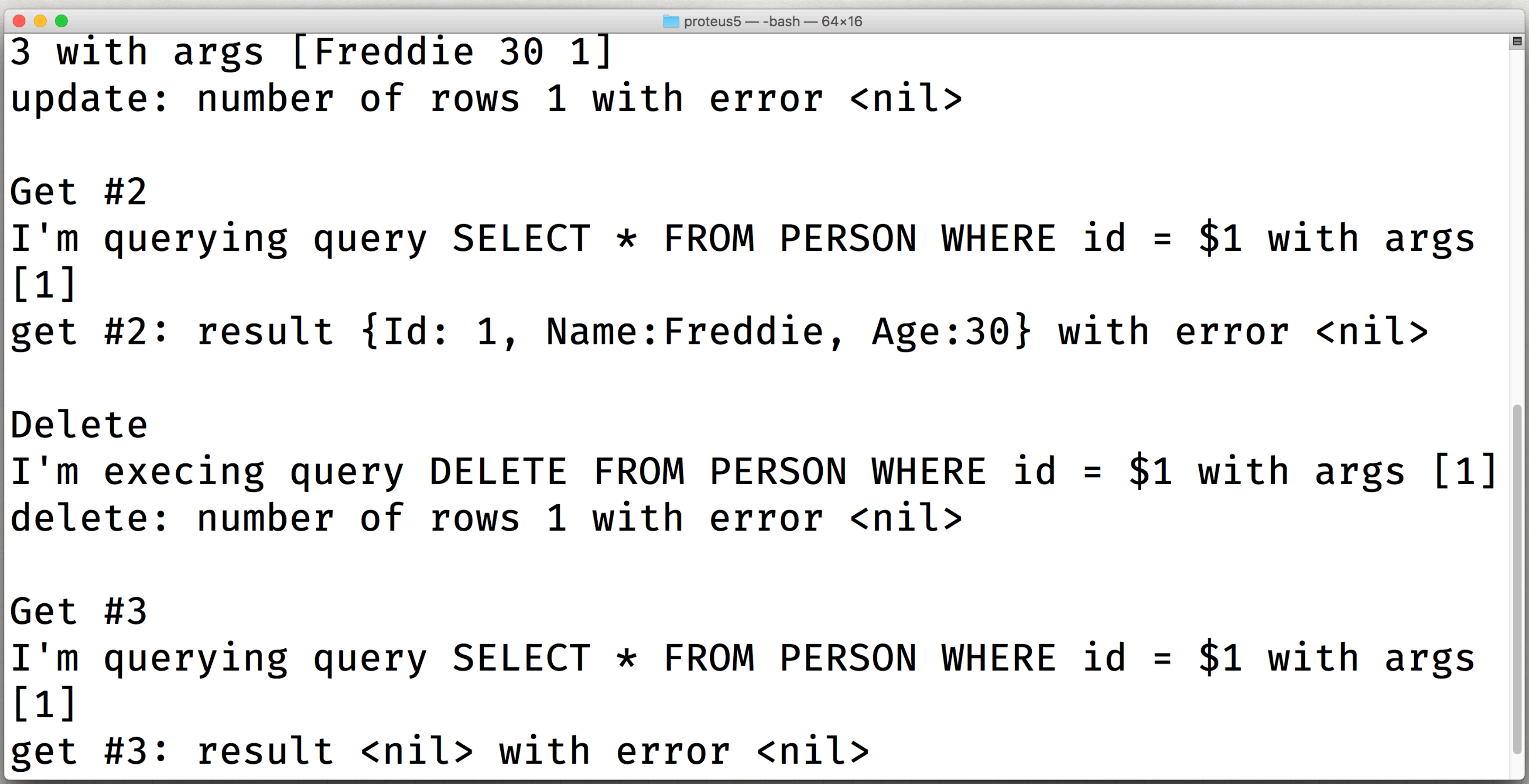
```
func buildMapper(returnType reflect.Type, zeroVal reflect.Value)
                Builder {
  //build map of col names to field names (makes this 2N instead of N^2)
  colFieldMap := map[string]fieldInfo{}

  for i := 0; i < returnType.NumField(); i++ {
    sf := returnType.Field(i)
    tagVal := sf.Tag.Get("prof")
    colFieldMap[tagVal] = fieldInfo{
      name:      sf.Name,
      fieldType: sf.Type,
      pos:       i,
    }
  }
  return func(cols []string, vals []interface{}) (reflect.Value, error) {
    returnVal := reflect.New(returnType)

    err := populateReturnVal(returnVal, cols, vals, colFieldMap)
    if err != nil {
      return zeroVal, err
    }
    return returnVal, err
  }
}
```

```go
func populateReturnVal(
         returnVal reflect.Value,
         cols []string,
         vals []interface{},
         colFieldMap map[string]fieldInfo) error {
  val := returnVal.Elem()
  for k, v := range cols {
    if sf, ok := colFieldMap[v]; ok {
      curVal := vals[k]
      rv := reflect.ValueOf(curVal)
      if rv.Elem().Elem().Type().ConvertibleTo(sf.fieldType) {
        val.Field(sf.pos).Set(rv.Elem().Elem().Convert(sf.fieldType))
      } else {
        return fmt.Errorf(
           "Unable to assign value %v of type %v to struct field %s of type %v",
           rv.Elem().Elem(), rv.Elem().Elem().Type(), sf.name, sf.fieldType)
      }
    }
  }
  return nil
}
```

```go
func mapOneRow(rows Rows, mapper Mapper, zeroVal reflect.Value)
                  (reflect.Value, error) {
  if !rows.Next() {
    if err := rows.Err(); err != nil {
      return zeroVal, err
    }
    return zeroVal, nil
  }

  cols, err := rows.Columns()
  if err != nil {
    return zeroVal, err
  }

  vals := make([]interface{}, len(cols))
  for i := 0; i < len(vals); i++ {
    vals[i] = new(interface{})
  }
  err = rows.Scan(vals...)
  if err != nil {
    return zeroVal, err
  }

  return mapper(cols, vals)
}
```



```
proteus5 $ go run *.go
Setting up the DAO
DAO created

Create
I'm execing query INSERT INTO PERSON(name, age) VALUES($1, $2) w
ith args [Fred 20]
create: number of rows 1 with error <nil>

Get
I'm querying query SELECT * FROM PERSON WHERE id = $1 with args
[1]
get: result {Id: 1, Name:Fred, Age:20} with error <nil>

Update
I'm execing query UPDATE PERSON SET name = $1, age=$2 where id=$
```


proteus5 — -bash — 64×16

```
3 with args [Freddie 30 1]
update: number of rows 1 with error <nil>

Get #2
I'm querying query SELECT * FROM PERSON WHERE id = $1 with args
[1]
get #2: result {Id: 1, Name:Freddie, Age:30} with error <nil>

Delete
I'm execing query DELETE FROM PERSON WHERE id = $1 with args [1]
delete: number of rows 1 with error <nil>

Get #3
I'm querying query SELECT * FROM PERSON WHERE id = $1 with args
[1]
get #3: result <nil> with error <nil>
```

RETURN MULTIPLE VALUES

BECAUSE THAT'S WHAT WE USUALLY DO

- Use a slice of a struct and an error for the return type

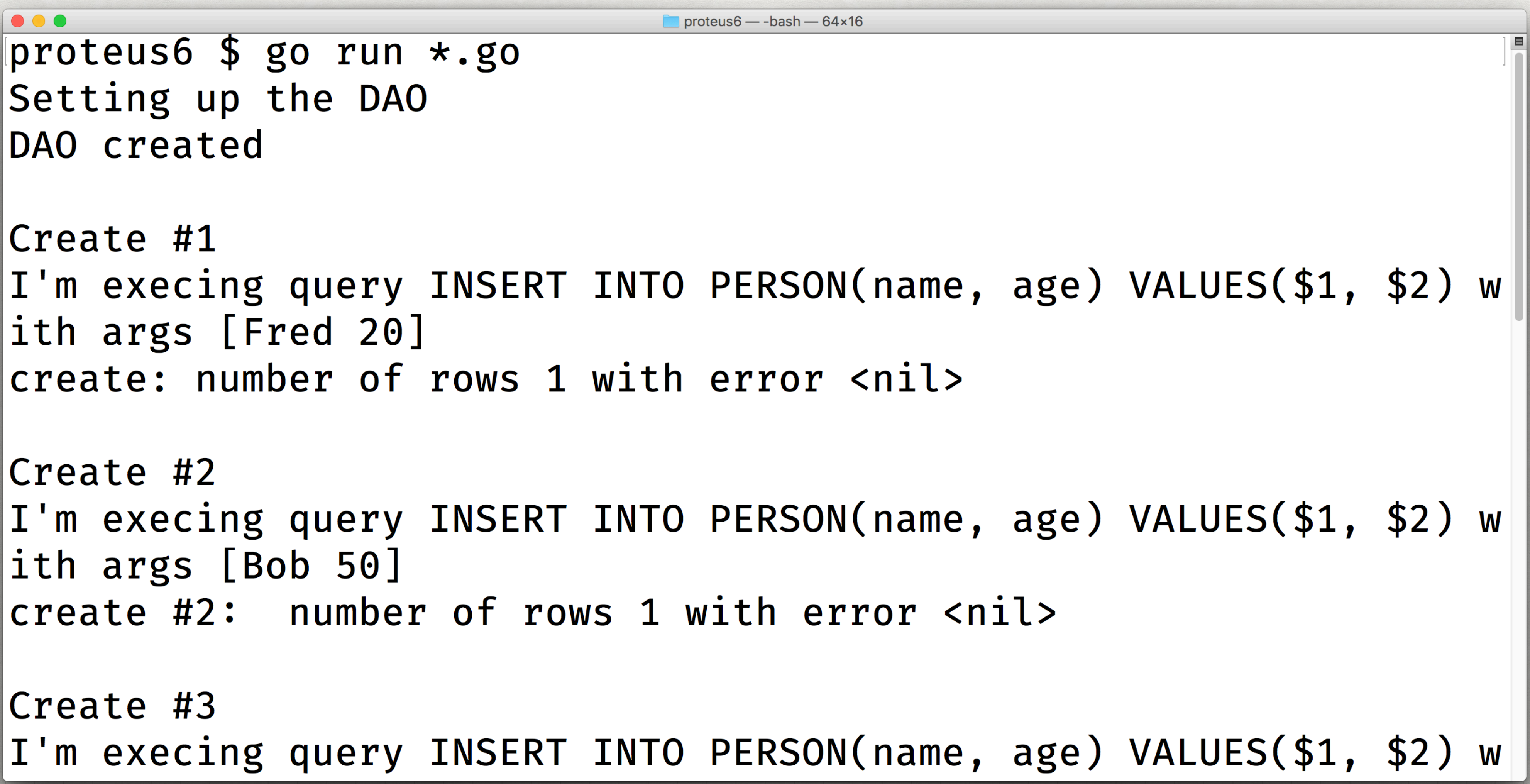
```
GetAll func(q Querier) ([]Person, error)
`proq:"SELECT * FROM PERSON"`
```

- Add a few Create calls so we can pull back multiple rows

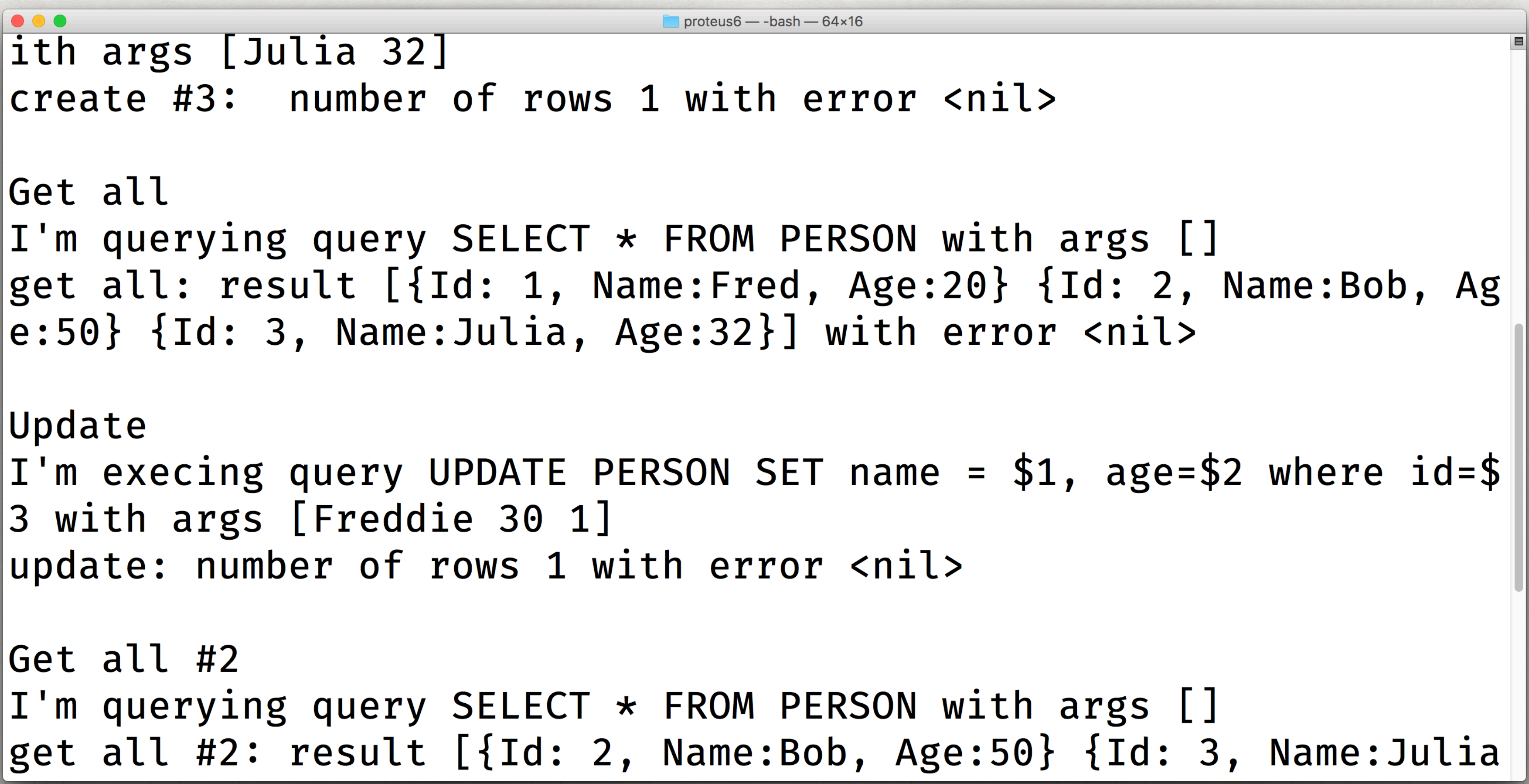
```
count, err = personDao.Create(wrapper, "Bob", 50)
fmt.Println("create #2:", count, err)

count, err = personDao.Create(wrapper, "Julia", 32)
fmt.Println("create #3:", count, err)

people, err := personDao.GetAll(wrapper)
fmt.Println("get all:", people, err)
```

```
func makeQuerierImplementation(funcType reflect.Type, query string,
                                                  paramOrder []paramInfo)
                                          (func([]reflect.Value) []reflect.Value, error) {


  // skipping unchanged code
  rowMapper := mapOneRow
  if firstResult.Kind() == reflect.Slice {
    rowMapper = func(rows Rows, mapper Mapper, zeroVal reflect.Value
                          (reflect.Value, error) {
      return mapAllRows(returnType, rows, mapper, zeroVal)
    }
  }

  //skipping more unchanged code
    result, err := rowMapper(rows, mapper, zeroVal)
  // skipping the rest of the unchanged code
}
```

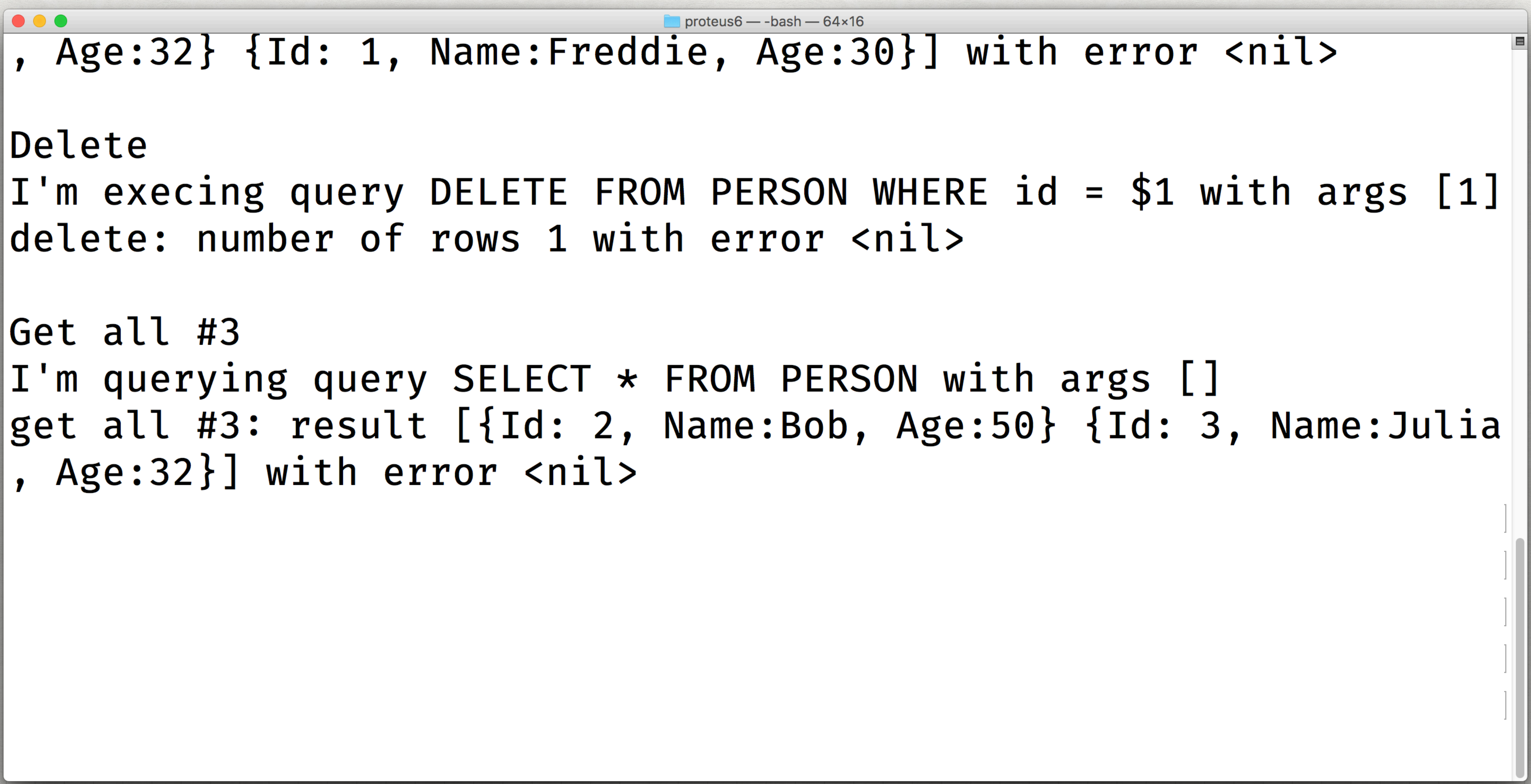
```go
func mapAllRows(returnType reflect.Type, rows Rows,
                 mapper Mapper, zeroVal reflect.Value)
                (reflect.Value, error) {
  cols, err := rows.Columns()
  if err != nil {
    return zeroVal, err
  }

  outSlice := reflect.MakeSlice(reflect.SliceOf(returnType), 0, 0)

  for rows.Next() {
    if err := rows.Err(); err != nil {
      return zeroVal, err
    }

    vals := make([]interface{}, len(cols))
    for i := 0; i < len(vals); i++ {
      vals[i] = new(interface{})
    }
    err = rows.Scan(vals...)
    if err != nil {
      return zeroVal, err
    }

    curVal, err := mapper(cols, vals)
    if err != nil {
      return zeroVal, err
    }
    outSlice = reflect.Append(outSlice, curVal.Elem())
  }
  if err := rows.Err(); err != nil {
    return zeroVal, err
  }
  if outSlice.Len() == 0 {
    return zeroVal, nil
  }
  return outSlice, nil
}
```


proteus6 — -bash — 64×16

```
proteus6 $ go run *.go
Setting up the DAO
DAO created

Create #1
I'm execing query INSERT INTO PERSON(name, age) VALUES($1, $2) w
ith args [Fred 20]
create: number of rows 1 with error <nil>

Create #2
I'm execing query INSERT INTO PERSON(name, age) VALUES($1, $2) w
ith args [Bob 50]
create #2:  number of rows 1 with error <nil>

Create #3
I'm execing query INSERT INTO PERSON(name, age) VALUES($1, $2) w
```



```
ith args [Julia 32]
create #3:  number of rows 1 with error <nil>

Get all
I'm querying query SELECT * FROM PERSON with args []
get all: result [{Id: 1, Name:Fred, Age:20} {Id: 2, Name:Bob, Ag
e:50} {Id: 3, Name:Julia, Age:32}] with error <nil>

Update
I'm execing query UPDATE PERSON SET name = $1, age=$2 where id=$
3 with args [Freddie 30 1]
update: number of rows 1 with error <nil>

Get all #2
I'm querying query SELECT * FROM PERSON with args []
get all #2: result [{Id: 2, Name:Bob, Age:50} {Id: 3, Name:Julia
```



```
, Age:32} {Id: 1, Name:Freddie, Age:30}] with error <nil>

Delete
I'm execing query DELETE FROM PERSON WHERE id = $1 with args [1]
delete: number of rows 1 with error <nil>

Get all #3
I'm querying query SELECT * FROM PERSON with args []
get all #3: result [{Id: 2, Name:Bob, Age:50} {Id: 3, Name:Julia
, Age:32}] with error <nil>
```

SLICES AS PARAMETERS

BUILD YOUR IN CLAUSES SAFELY

- Dynamically-sized IN clauses are tricky
  - Need a parameter for each element in the clause
  - Don’t want to use string concatenation to build it up
- Proteus can do it for you!

```
GetByAge func(q Querier, ages []int) ([]Person, error)
 `proq:"SELECT * from PERSON WHERE age in (:ages:)" prop:"ages"`
```

SLICES AS PARAMETERS

BUILD YOUR IN CLAUSES SAFELY

- Go Templates are the secret
  - Generate the query once if there are no slices in it
  - Generate on every invocation if there are slices in it

FROM STRUCT TAGS TO SQL TEMPLATES

STEP BY STEP

Struct Tag proq:

```
UPDATE PERSON SET name = :name:, age=:age:
where id=:id:
```

Struct Tag prop: `id,name,age`

FROM STRUCT TAGS TO SQL TEMPLATES

CONVERT PROP TO MAP[STRING]INT

Struct Tag proq: `UPDATE PERSON SET name = :name:, age=:age: where id=:id:`

nameOrderMap: `{“id”:1, “name”:2, “age”:3}`

FROM STRUCT TAGS TO SQL TEMPLATES

CONVERT PROQ AND NAMEORDERMAP TO TEMPLATE AND PARAMORDER

SQL Template:
```
UPDATE PERSON SET name = {{.name | join}},
 age={{.age | join}} WHERE id={{.id | join}}
```

paramOrder:
```
[]paramInfo{{“name”,2,false}, {“age”,3,false},
                 {“id”,1,false}}
```

FROM STRUCT TAGS TO SQL TEMPLATES

FINALIZE IMMEDIATELY BECAUSE THERE ARE NO SLICES

SQL Query: `UPDATE PERSON SET name = $1, age=$2 WHERE id=$3`

paramOrder:
```
[]paramInfo{{“name”,2,false}, {“age”,3,false},
            {“id”,1,false}}
```

FROM STRUCT TAGS TO SQL TEMPLATES

STEP BY STEP

Struct Tag proq: `SELECT * from PERSON WHERE age in (:ages:)`

Struct Tag prop: `ages`

FROM STRUCT TAGS TO SQL TEMPLATES

CONVERT PROP TO MAP[STRING]INT

Struct Tag proq: `SELECT * from PERSON WHERE age in (:ages:)`

nameOrderMap: `{“ages”:1}`

FROM STRUCT TAGS TO SQL TEMPLATES

CONVERT PROQ AND NAMEORDERMAP TO TEMPLATE AND PARAMORDER

SQL Template:
```
SELECT * from PERSON
 WHERE age in ({{.ages | join}})
```

paramOrder:
```
[]paramInfo{{“ages”,1,true}}
```

FROM SQL TEMPLATES TO QUERY

USE ARGUMENTS TO FINALIZE THE QUERY WHEN INVOKED

SQL Template:
```
SELECT * from PERSON
 WHERE age in ({{.ages | join}})
```

paramOrder:
```
[]paramInfo{{“ages”,1,true}}
```

args:
```
[]reflect.Value{{Querier}, []int{20,32}}
```

FROM SQL TEMPLATES TO QUERY

USE ARGUMENTS TO FINALIZE THE QUERY WHEN INVOKED

SQL Query: `SELECT * from PERSON WHERE age in ($1, $2)`

paramOrder: `[]paramInfo{{“ages”,1,true}}`

args: `[]reflect.Value{{Querier}, []int{20,32}}`

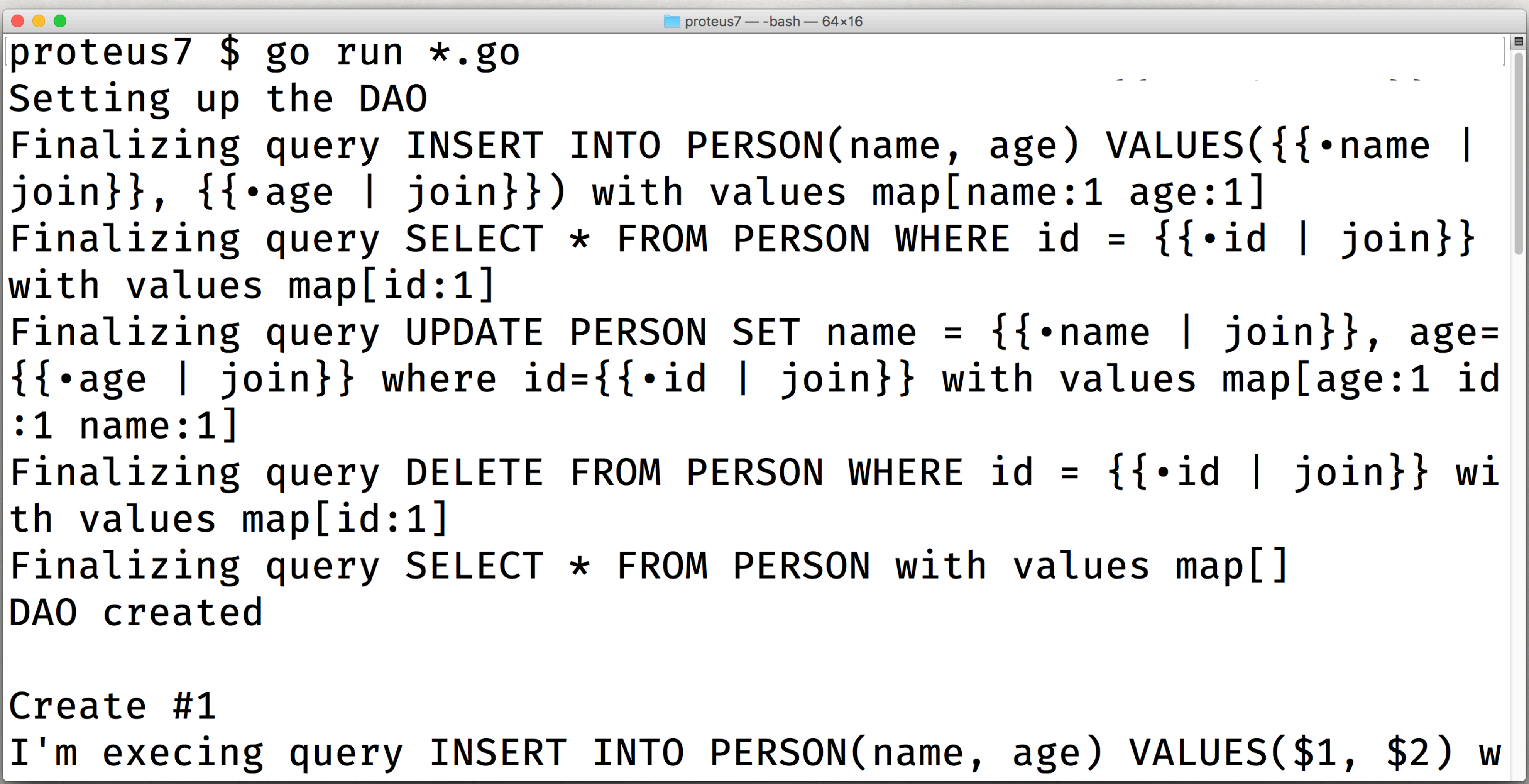
```go
type queryHolder interface {
  finalize(args []reflect.Value) (string, error)
}



type simpleQueryHolder string



func (sq simpleQueryHolder) finalize(args []reflect.Value)
                                              (string, error) {
  return string(sq), nil
}
```

```go
type templateQueryHolder struct {
  queryString string
  pa          ParamAdapter
  paramOrder  []paramInfo
}



func (tq templateQueryHolder) finalize(args []reflect.Value)
                                                 (string, error) {
  return doFinalize(tq.queryString, tq.paramOrder, tq.pa, args)
}
```

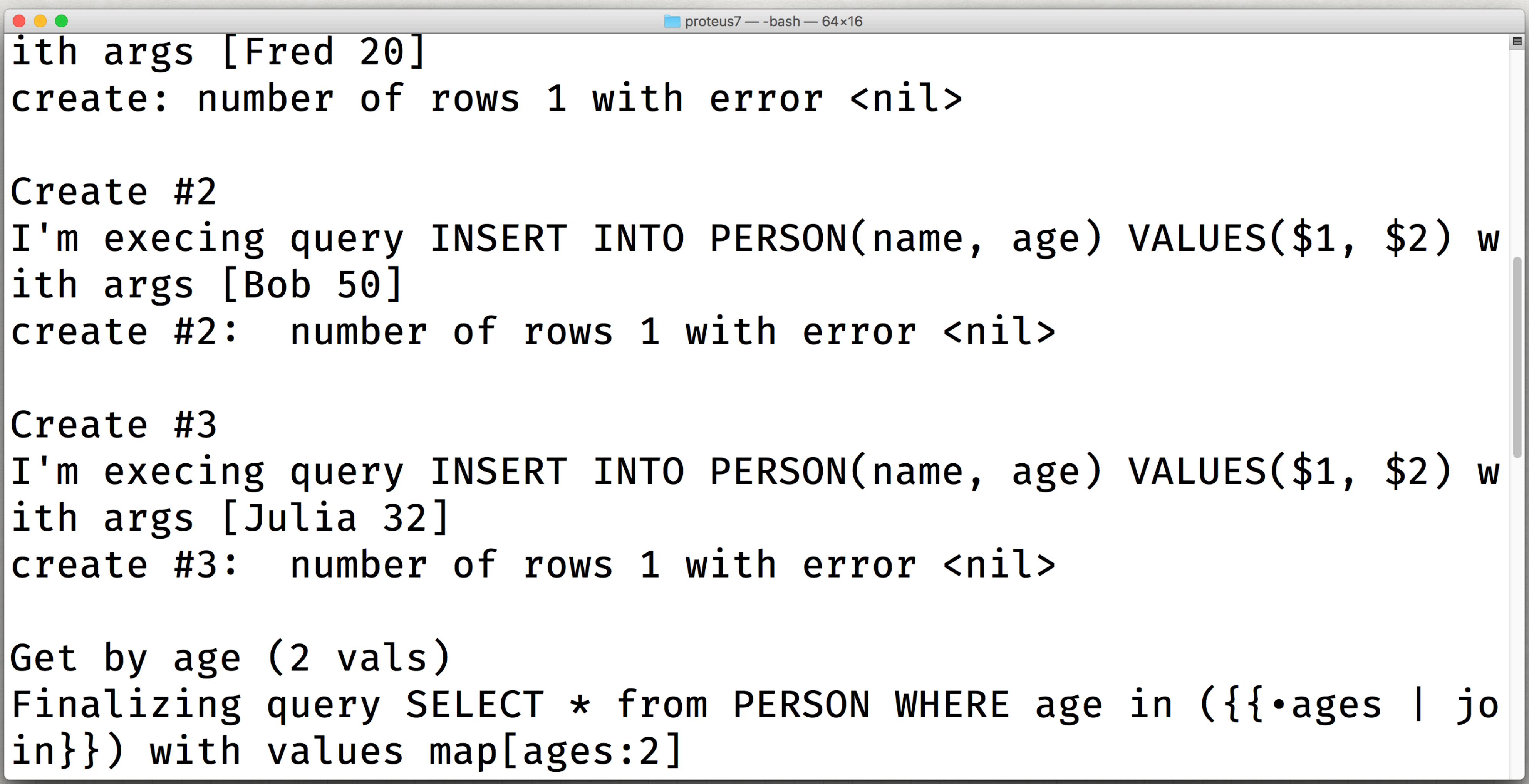
```go
func doFinalize(queryString string, paramOrder []paramInfo, pa ParamAdapter, args []reflect.Value)
               (string, error) {
  temp, err := template.New(“query”).Funcs(template.FuncMap{"join": joinFactory(1, pa)}).
                        Parse(queryString)
  if err != nil {
    return "", err
  }

  var b bytes.Buffer
  sliceMap := map[string]interface{}{}
  for _, v := range paramOrder {
    if v.isSlice {
      sliceMap[v.name] = args[v.posInParams].Len()
    } else {
      sliceMap[v.name] = 1
    }
  }
  if err == nil {
    fmt.Println("Finalizing query", queryString, "with values", sliceMap)
    err = temp.Execute(&b, sliceMap)
  }
  if err != nil {
    return "", err
  }
  return b.String(), err
}
```

```
finalQuery, err := query.finalize(args)
```

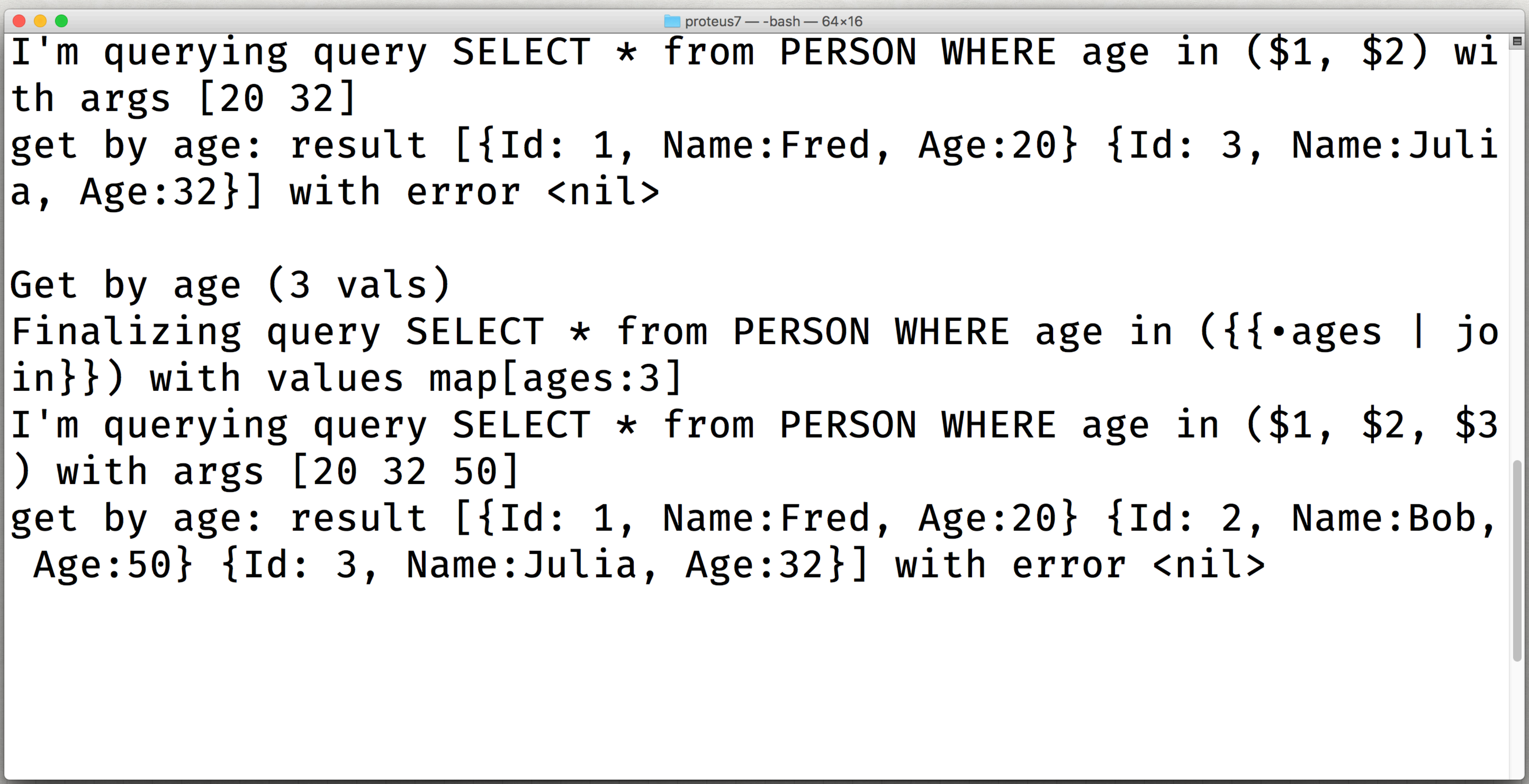
```go
func buildQueryArgs(funcArgs []reflect.Value, paramOrder []paramInfo)
                    []interface{} {
  out := []interface{}{}
  for _, v := range paramOrder {
    if v.isSlice {
      curSlice := funcArgs[v.posInParams]
      for i := 0; i < curSlice.Len(); i++ {
        out = append(out, curSlice.Index(i).Interface())
      }
    } else {
      out = append(out, funcArgs[v.posInParams].Interface())
    }
  }
  return out
}
```


proteus7 — -bash — 64×16

```
proteus7 $ go run *.go
Setting up the DAO
Finalizing query INSERT INTO PERSON(name, age) VALUES({{•name |
join}}, {{•age | join}}) with values map[name:1 age:1]
Finalizing query SELECT * FROM PERSON WHERE id = {{•id | join}}
with values map[id:1]
Finalizing query UPDATE PERSON SET name = {{•name | join}}, age=
{{•age | join}} where id={{•id | join}} with values map[age:1 id
:1 name:1]
Finalizing query DELETE FROM PERSON WHERE id = {{•id | join}} wi
th values map[id:1]
Finalizing query SELECT * FROM PERSON with values map[]
DAO created

Create #1
I'm execing query INSERT INTO PERSON(name, age) VALUES($1, $2) w
```



```
ith args [Fred 20]
create: number of rows 1 with error <nil>

Create #2
I'm execing query INSERT INTO PERSON(name, age) VALUES($1, $2) w
ith args [Bob 50]
create #2:  number of rows 1 with error <nil>

Create #3
I'm execing query INSERT INTO PERSON(name, age) VALUES($1, $2) w
ith args [Julia 32]
create #3:  number of rows 1 with error <nil>

Get by age (2 vals)
Finalizing query SELECT * from PERSON WHERE age in ({{•ages | jo
in}}) with values map[ages:2]
```



```
I'm querying query SELECT * from PERSON WHERE age in ($1, $2) wi
th args [20 32]
get by age: result [{Id: 1, Name:Fred, Age:20} {Id: 3, Name:Juli
a, Age:32}] with error <nil>

Get by age (3 vals)
Finalizing query SELECT * from PERSON WHERE age in ({{.ages | jo
in}}) with values map[ages:3]
I'm querying query SELECT * from PERSON WHERE age in ($1, $2, $3
) with args [20 32 50]
get by age: result [{Id: 1, Name:Fred, Age:20} {Id: 2, Name:Bob,
 Age:50} {Id: 3, Name:Julia, Age:32}] with error <nil>
```

PROTEUS FEATURES

- Runtime generated
- Typesafe
- SQL injection-proof
- Works with standard Go SQL libraries
- Adapts to different databases easily
- Full code base has
    - More features (Query Mappers, struts as input parameters, flexible out parameters)
    - More error checking
    - More validations

IS IT FAST ENOUGH?

NOSQL

PROTEUS CAN DO THAT TOO!

SQL IS A DSL

AND THERE ARE A LOT OF THEM OUT THERE

- Domain-Specific Languages
- Describe what you want to do instead of how to do it
    - REST clients

KEEP ON EXPLORING!

- Go can be used to create declaration-driven code
- By combining struct tags, function generation, reflection, and templating we can:
  - Increase productivity
  - Provide the functionality of other languages
  - Keep type safety and performance
  - Write code that feels like Go

- Proteus: https://github.com/jonbodner/proteus
- Sample Code: https://github.com/jonbodner/proteus-talk

- https://www.flickr.com/photos/eltpics/8754972328
- http://www.tomsitpro.com/articles/open-source-cloud-computing-software,2-754-8.html