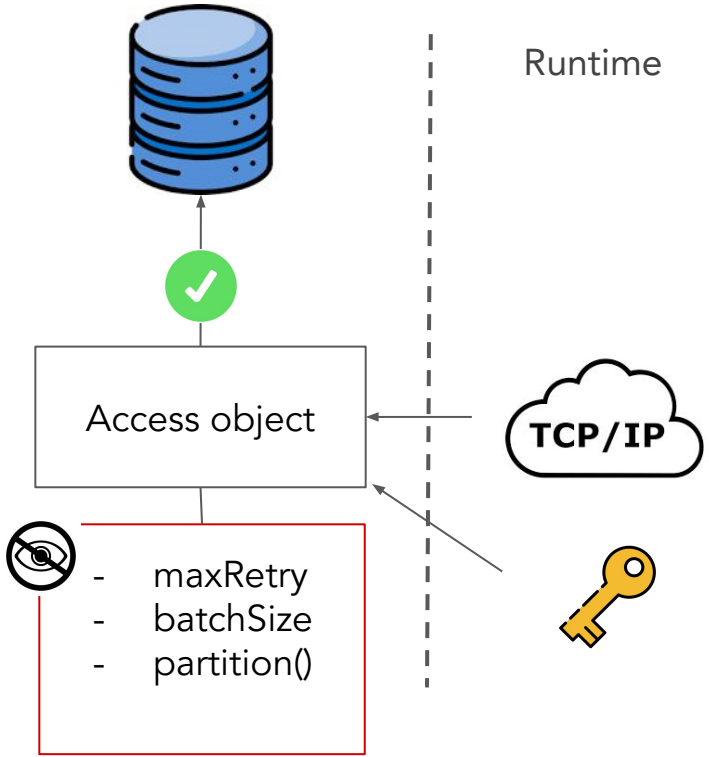
DWARVES FOUNDATION

Prototype

One of the creational patterns

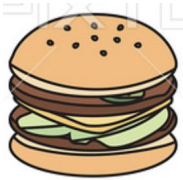
Problem statement

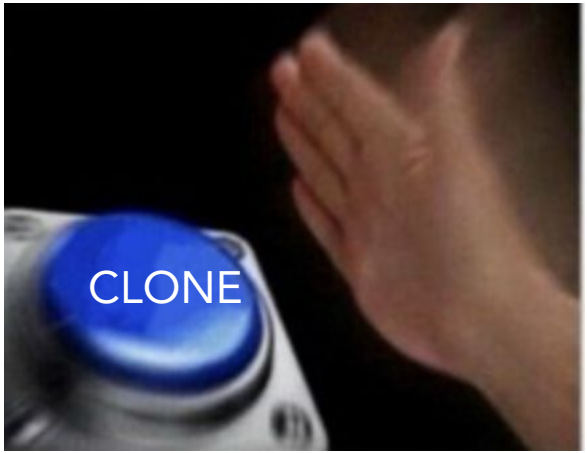
- We want to use a copy of the access object but the config only initialize at runtime. It fields and methods are also privated.

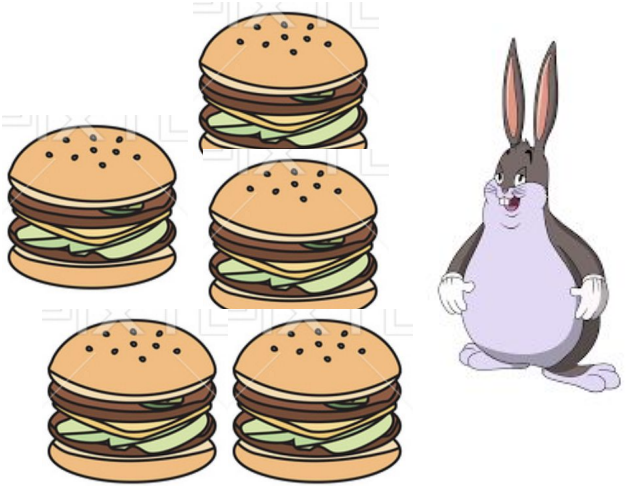


Prototype

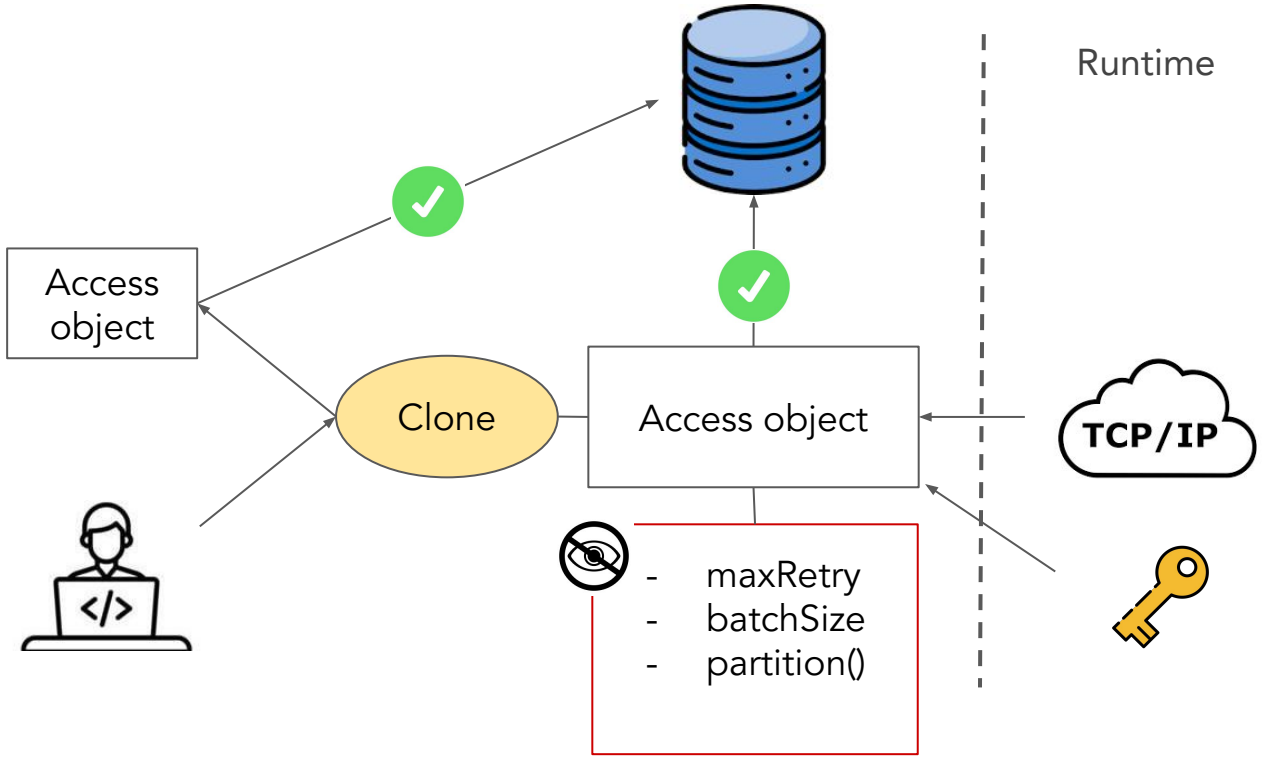
- Request a **Clone** of from the object(Prototype) without the need to look up it class and implementation.







Prototype



DWARVES FOUNDATION

Go example

- Given a Counter interface. Main will select the counter implementation at runtime

```go
package counter

type Counter interface {
    Count() int
}

type normalCounter struct {
    count int
}

func (a *normalCounter) Count() int {
    a.count++
    return a.count
}

type defectedCounter struct {
    count int
}

func (a *defectedCounter) Count() int {
    a.count++
    return -a.count
}

func NewCounter(isDefected bool) Counter {
    if isDefected {
        return &defectedCounter{}
    }
    return &normalCounter{}
}
```

```go
package main

import (
    "[redacted]/counter"
)

func main() {
    isDefected := true // set at runtime

    counter := counter.NewCounter(isDefected)
}
```

DWARVES FOUNDATION

Go example

- We have a **store** that will use **Counter** interface to manage inventory
- Each inventory must have a separate counter instance

```go
package store

import (
    "log"
    [redacted]/counter"
)

type store struct {
    phoneInventory    counter.Counter
    computerInventory counter.Counter
}

func (s store) AddPhone() {
    log.Printf("number phone total: %d\n", s.phoneInventory.Count())
}

func (s store) AddComputer() {
    log.Printf("number computer total: %d\n", s.computerInventory.Count())
}
```

Go example

- The store can not decide the counter instance itself and must rely on main to provide the counter instance at runtime.

```go
package store

func NewStore(counter counter.Counter) store {
    phoneCounter := counter
    computerCounter := counter

    return store{
        phoneInventory:      phoneCounter,
        computerInventory:   computerCounter,
    }
}
```

```go
package main

func main() {
    isDefected := true // set at runtime

    counter := counter.NewCounter(isDefected)

    store := store.NewStore(counter)

    for i := 0; i < 2; i++ {
        store.AddPhone()
    }

    for i := 0; i < 1; i++ {
        store.AddComputer()
    }
}
```

Go example

- So at runtime, both phone and computer inventory are using the same counter instance, output the incorrect result.

isDefected = false

```
number phone total: 1
number phone total: 2
number computer total: 3
```

isDefected = true

```
number phone total: -1
number phone total: -2
number computer total: -3
```

Go example

- We add clone() to Counter to replicate the prototype instance and update Store

```go
package counter

type Counter interface {
    Count() int
    Clone() Counter
}

func (a *normalCounter) Clone() Counter {
    return &normalCounter{}
}

func (a *defectedCounter) Clone() Counter {
    return &normalCounter{
        count: 100,
    }
}
```

```go
package store

func NewStore(counter counter.Counter) store {
    phoneCounter := counter.Clone()
    computerCounter := counter.Clone()

    return store{
        phoneInventory:    phoneCounter,
        computerInventory: computerCounter,
```

Go example

- Prototype cloned runtime outputs:

isDefected = false

```
number phone total: 1
number phone total: 2
number computer total: 1
```

isDefected = true

```
number phone total: 101
number phone total: 102
number computer total: 101
```