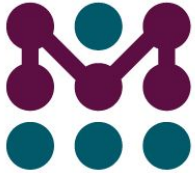


MENDER.io

Koduj aplikacje dla urządzeń embedded szybciej: porównanie Go i C

Marcin Pasinski
northern.tech
mender.io

Trochę o mnie

- Marcin Pasiński
  - 15+ lat w IT
  - mgr elektroniki i telekomunikacji
  - marcin.pasinski@northern.tech
  - mpasinski@gmail.com

Northern.tech

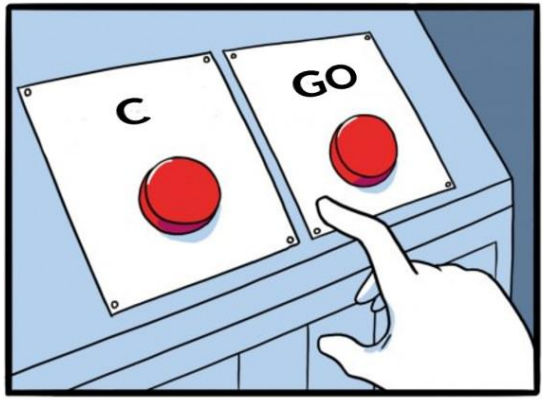
MENDER.io

- OTA updater for embedded/IoT
- Open source (Apache v2 license)
- Technologia: Go, Yocto

- Configuration Management
- Open source (GPL v3 license)
- Technologia: C

Moje (subiektywne) spojrzenie na C i Go

- Go jest bardzo produktywnym dla deweloperów/koderów językiem programowania
- Pokazuje swoje możliwości przy programowaniu aplikacji wykorzystujących sieć lub wielowątkowość

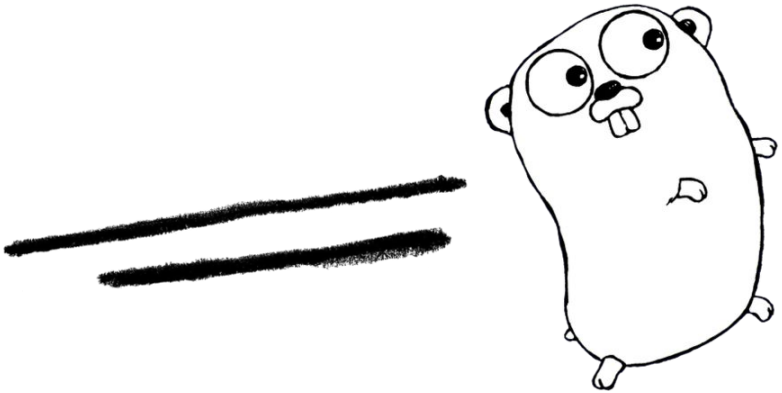


- Nie uważam, że jest konkurentem i może zastąpić C
- Samo “garbage collection” skutecznie wyklucza Go z konkurowania z C

Rozkład jazdy

- Krótko o powstaniu Go
- Dlaczego Go w northern.tech (mender.io)?
- Podstawy Go
- Go w “embedded”
- Demo

Trochę historii

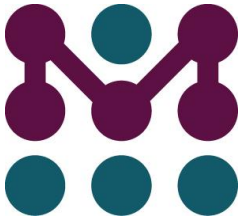


Robert Griesemer, Rob Pike and Ken Thompson: pierwsze pomysły	Ian Taylor: GCC front end	Public open source	Go v1	Go v1.19
September 21, 2007	May 2008	November 10, 2009	March 28, 2012	August 2, 2022

Dlaczego powstało Go?

- “Go was born out of frustration with existing languages and environments for **systems programming**.”
- “One had to choose either efficient compilation, efficient execution, or ease of programming; all three were not available in the same mainstream language.”

https://golang.org/doc/faq



Nasza historia z Go

mender.io

Wymagania dla języka programowania

1. “Czynniki zewnętrzne”
   - Rozmiar na urządzeniu docelowym
   - Wsparcie dla Yocto
   - Możliwość kompilacji na różne platformy

2. “Czynniki wewnętrzne”
   - “know-how” w firmie
   - Możliwość dzielenia kodu i pracy między zespołami
   - Szybkość pisania aplikacji
   - Dostęp do bibliotek (JSON, SSL, HTTP)
   - Automatyczne zarządzanie pamięcią
   - “Bezpieczeństwo” aplikacji (buffer overflow, etc.)

Porównanie różnych opcji

	C	C++	Go
Size requirements in devices	Lowest	Low (1.8MB more)	Low (2.1 MB more, however will increase with more binaries)
Setup requirements in Yocto	None	None	Requires 1 layer (golang)*
Competence in the company	Good	Have some long time users	Only couple of people know it
Buffer under/overflow protection	None	Little	Yes
Code reuse/sharing from CFEngine	Good	Easy (full backwards compatibility)	Can import C API
Automatic memory management	No	Available, but not enforced	Yes
Standard data containers	No	Yes	Yes
JSON	json-c	jsoncpp	Built-in
HTTP library	curl	curl	Built-in
SSL	OpenSSL	OpenSSL	Built-in

* Go is natively supported by Yocto Project from Pyro release (Yocto 2.3)

Porównanie różnych opcji

	C	C++	C++/Qt	Go	...
Base image size	8.4MB	10.2MB	20.8MB*	14.6MB	
Size with network stack	13.4MB (curl)	15.2MB (curl)	20.8MB*	14.6MB	
Shared dependencies	Yes	Yes	Yes	No/Maybe	
Extra Yocto layer needed	No	No	Yes	Yes**	
Deployment complexity	Binary	Binary	Binary + Qt	Binary	

* Required some changes to upstream Yocto layer
** Go is natively supported by Yocto from Pyro release (Yocto 2.3)

Dlaczego wybraliśmy Go?

1. Go jest językiem kompilowanym.
2. Go jest statycznie linkowany i nie potrzebuje żadnych zależności (*pod warunkiem nie używania CGO_ENABLED=1*).
3. Dobre wsparcie dla kross-kompilacji i wiele wspieranych platform
4. Możliwość pisania kodu klienta i backendu w tej samej technologii.
5. Go posiada sporo bibliotek i konstrukcję języka, która pozwala na szybkie pisanie kodu.
6. Znając C i Python łatwo zacząć pisać w Go i szybko zostać produktywnym developerem.

Go vs C: rozmiar binarki ;)

```
package main
func main() {
   println("hello world")
}
```

- $ go build
  - 938K
- $ go build -ldflags '-s -w'
  - **682K**
- $ go build & strip
  - 623K

```
package main
import "fmt"
func main() {
   fmt.Println("hello world")
}
```

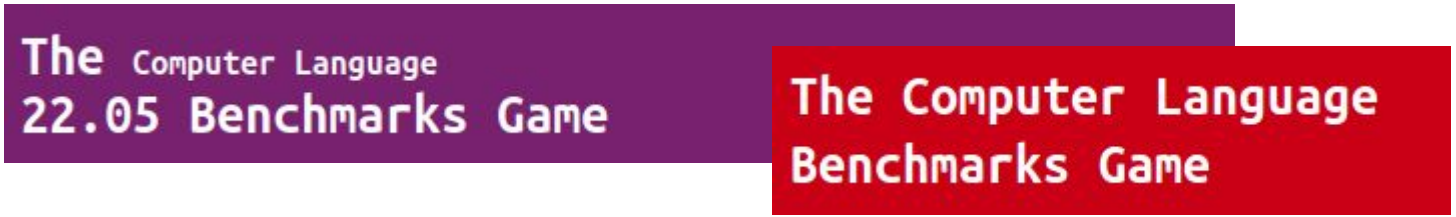
- $ go build
  - **1,5M**

```
#include <stdio.h>

int main(void)
{
  printf("hello world\n");
  return 0;
}
```

- gcc main.c
  - 8,5K
- ldd a.out
  - linux-vdso.so.1
  - libc.so.6
  - /lib64/ld-linux-x86-64.so.2
- gcc -static main.c
  - 892K
- gcc -static main.c & strip
  - **821K**

Go vs C: “szybkość”

1. Go jest w pełni “garbage-collected”
2. Automatyczne zarządzanie pamięcią
   - można sprawdzić: $ go build -gcflags -m
3. Szybka kompilacja (i łatwa)
4. Szybkość pisania kodu i wystartowania projektu



https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/go-gpp.html

Go: podstawy podstaw

- Biblioteka standardowa
- Narzędzia
- Kompilacja
- Wielowątkowość
- Linkowanie aplikacji C i C++
- Go w aplikacjach wbudowanych

Biblioteka standardowa

- https://golang.org/pkg/
  - io/ioutil/os
  - flag
  - net (http, rpc, smtp)
  - encoding (JSON, xml, hex, csv, binary, ...)
  - compress and archive (tar, zip, gzip, bzip2, zlib, lzw, ...)
  - crypto (aes, des, ecdsa, hmac, md5, rsa, sha1, sha256, sha512, tls, x509, ...)
  - db (sql)
  - regexp
  - sync, atomic
  - unsafe, syscall

Narzędzia w pakiecie z językiem

- fmt
- test
- cover
- pprof
- doc
- vet
- mod
- i wiele więcej

Kompilacja

- Kompilatory
  - Oryginalny kompilator Go, **gc**, został napisany w C
  - Od Go 1.5 kompilator Go napisany natywnie w Go
  - **gccgo** (frontend dla GCC; https://golang.org/doc/install/gccgo)
    - gcc 11 wspiera Go 1.16.3

- Kompilacja
  - Bardzo szybka (duże moduły kompilowane w sekundy)
  - Pojedyncza “binarka” (brak zależności, brak konieczności użycia maszyn wirtualnych)
    - Od Go 1.5 wsparcie dla dzielonych bibliotek i dynamicznego linkowania
  - Wsparcie dla makefile (https://github.com/mendersoftware/mender/blob/master/Makefile)

Kompilacja (https://golang.org/doc/install/source#environment)

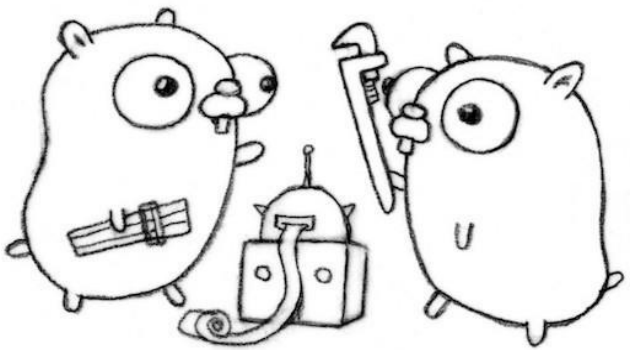
GO OS / GO ARCH	amd64	386	arm	arm64	ppc64le	ppc64	mips64le	mips64	mipsle	mips	wasm	risc64
aix						X						
android	X	X	X	X								
darwin	X			X								
freebsd	X	X	X									
illumos				X								
ios				X								
linux	X	X	X	X	X	X	X	X	X	X		X
netbsd	X	X	X									
openbsd	X	X	X	X								
plan9	X	X	X									
solaris	X											
windows	X	X	X	X								

Debugowanie

- gdb
- delve (https://github.com/derekparker/delve)

Wsparcie testowania

- Wbudowana biblioteka testów jednostkowych
- Benchmarks
- Fuzzing
- Bardzo łatwe kodowanie i wywoływanie:
  - import “testing”
  - dodaj “_test” do nazwy pliku
  - dodaj “Test” i “*testing.T“ albo “Benchmark” i “*testing.B” do sygnatury funkcji (“Fuzz” i “*testing.F“)



Variables

- Variable declarations

```
package main
var e, l, c bool
func main() {
    var ctr int
    var elm string = “gdansk”
    var a, s, d = true, false, “data”
    f := 1
}
```

- Basic types
  - bool
  - string
  - int, int8, int16, int32, int64
  - uint, uint8, uint16, uint32, uint64, uintptr
  - byte //alias for uint8
  - rune //represents a Unicode point; alias for int32
  - float, float64
  - complex64, complex128

Functions

- Functions
  - take zero or more arguments
  - arguments pass by value
  - multiple return values

```
func div(x, y int) (int, error) {
    if y == 0 {
      return 0, errors.New("div by 0")
    }
    return x / y, nil
}

func main() {
    fmt.Println(div(4, 0))
}
```

Structures and methods

- Structs
  - Struct is collection of fields
- Methods
  - Functions with receiver argument
  - Can be declared on non-struct objects

```
type Point struct {
    X int
    Y int
}

type Square struct {
    Vertex Point
    Size int
}
func (s Square) area() int {
    return s.Size * s.Size
}
func (s *Square) setPoint(p Point) {
    s.Vertex = p
}

s := Square{
    Vertex: Point{X: 2, Y: 3},
    Size:   3}
fmt.Printf("area: %d", s.area())
```

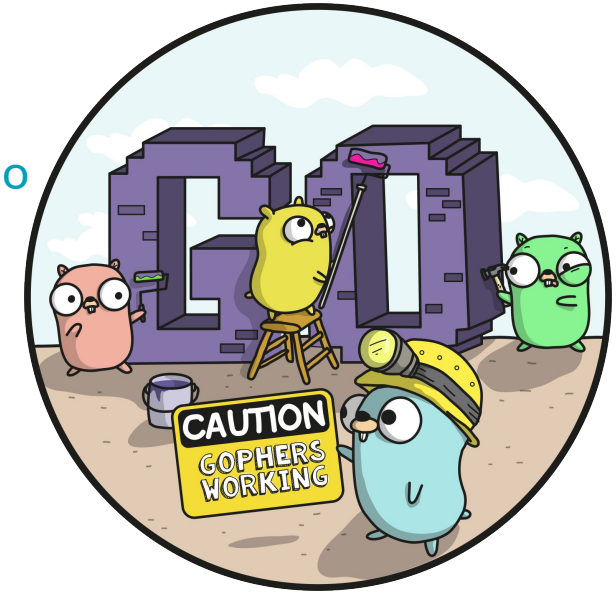
Interfaces

- Interfaces
  - Set of method signatures
  - Implemented implicitly
    - no explicit declaration
    - no “implements”
- Decoupled definition and implementation
- Empty interface *interface{}*
- Use “duck typing” (technically they call it ‘structural typing’)

```
type Printer interface {
  Print() (string, error)
}

type myType int
func (mt myType) Print() (string, error) {
  return “this is my int”, nil
}

main() {
  var p Printer = myType(1)
  i.Print()
}
```

Wielowątkowość

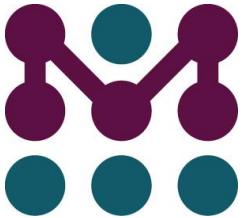
- Goroutines
  - Funkcje wywoływane wielowątkowo razem z innymi funkcjami
  - Tylko kilka kB narzutu (2kB)
  - Zarządzane przez Go i multipleksowane do wątków systemu operacyjnego
- Channels
  - Używane do komunikacji i synchronizacji
  - Wysyłanie i odbieranie są operacjami “blokowanymi”
  - Mogą być używane jako buforowane lub bez bufora



Wielowątkowość

- Goroutines
  - *go func()*
- Channels
  - *c := make(chan int)*

```
package main

func main() {
  messages := make(chan string)
  go func() { messages <- "ping" }()

  select {
     case msg := <- messages:
       fmt.Println(msg)
     case <- time.After(time.Second):
       fmt.Println("timeout")
     default:
       fmt.Println("no activity")
       time.Sleep(50 * time.Millisecond)
  }
}
```



Go w embedded

Zbiór możliwości

C razem z Go

- CGO (https://golang.org/cmd/cgo/)
  - CGO_ENABLED
  - Pozwala na dostęp do funkcji i zmiennych z C
  - Zaimportowane funkcje są dostępne w “wirtualnym” module “C”
  - Wywołania funkcji C powodują narzut wydajnościowy (~150ns on Xeon processor)

```
/*
#cgo LDFLAGS: -lpcap
#include <stdlib.h>
#include <pcap.h>
*/
import "C"

func getDevice() (string, error) {
    var errMsg string
    cerr := C.CString(errMsg)
    defer C.free(unsafe.Pointer(cerr))

    cdev := C.pcap_lookupdev(cerr)
    dev := C.GoString(cdev)
        return dev, nil
}
```

C++ razem z Go

- SWIG
  - Simplified Wrapper and Interface Generator
  - Narzędzie do łączenia aplikacji napisanych w C i C++ z innymi językami programowania
  - http://www.swig.org/Doc2.0/Go.html

```
// helloclass.cpp
std::string HelloClass::hello(){
    return "world";
}

// helloclass.h
class HelloClass
{
public:
     std::string hello();
}

// mylib.swig
%module mylib
%{
#include "helloclass.h"
%}
```

Dzielone biblioteki Go

- Dostępne od Go 1.5
  - *-buildmode* argument
    - archive
    - c-archive
    - c-shared
    - shared
    - exe
- ~ go build -buildmode=shared -o myshared
- ~ go build -linkshared -o app myshared

```
// package name: mygolib
package main

import "C"
import "fmt"

//export SayHiGdansk
func SayHiElc(name string) {
  fmt.Printf("Hello Gdansk: %s!\n",
name)
}

func main() {
  // We need the main for Go to
  // compile C shared library
}
```

Dzielone biblioteki Go

- ~ go build -buildmode=c-shared -o mygolib.a mygolib.go
- ~ gcc -o myapp myapp.c mygolib.a

```
// mygolib.h
typedef signed char GoInt8;
typedef struct { char *p; GoInt n; }
GoString;


extern void SayHiGdansk(GoString p0);


// myapp.c
#include "mygolib.h"
#include <stdio.h>

int main() {
  printf("Go from C app.\n");
  GoString name = {"Prague", 6};
  SayHiGdansk(name);
  return 0;
}
```

Alokacja pamięci

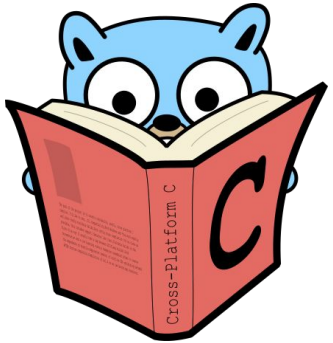
- Stos i sterta
  - go build -gcflags -m
  - *./main.go:17: msg escapes to heap*
- Unsafe
  - C: *(uint8_t*)0x1111 = 0xFF;
  - *“Manipulating hardware directly is possible with GO, but it has been made intentionally cumbersome.”*

```
file, _ := os.OpenFile("/dev/gpiomem",
      os.O_RDWR|os.O_SYNC, 0);

mem, _ := syscall.Mmap(int(file.Fd()),
    0x20000000, 4096,
    syscall.PROT_READ|syscall.PROT_WRITE,
    syscall.MAP_SHARED)

header :=
*(*reflect.SliceHeader)(unsafe.Pointer(&mem))

memory =
*(*[]uint32)(unsafe.Pointer(&header))
```

Biblioteki/narzędzia

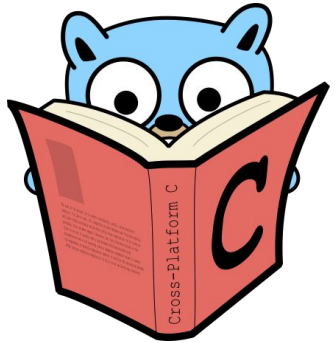
- TinyGo (bazuje na LLVM)
  - https://github.com/tinygo-org/tinygo
- gollvm
- EMBD: https://embd.kidoman.io/

Doświadczenia z Go



1. Problemy z zarządzaniem bibliotekami zewnętrznymi
   - Mocno usprawnione z wprowadzeniem wsparcia dla go mod (od wersji 1.11)
2. Jakość dostępnych bibliotek pozostawia wiele do życzenia
   - Mocna poprawa z czasem i adopcją języka
3. Problemy ze wsparciem Yocto
   - Do momentu oficjalnego wsparcia Go w Yocto
4. Użycie cgo mocno ogranicza zalety łatwej kompilacji
5. Adopcja języka i trochę “nudne” pisanie kodu
6. Mała społeczność ”embedded”

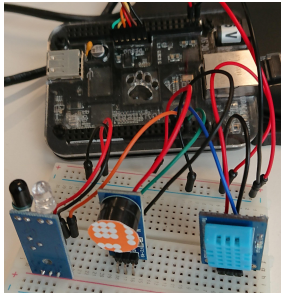
Doświadczenia z Go

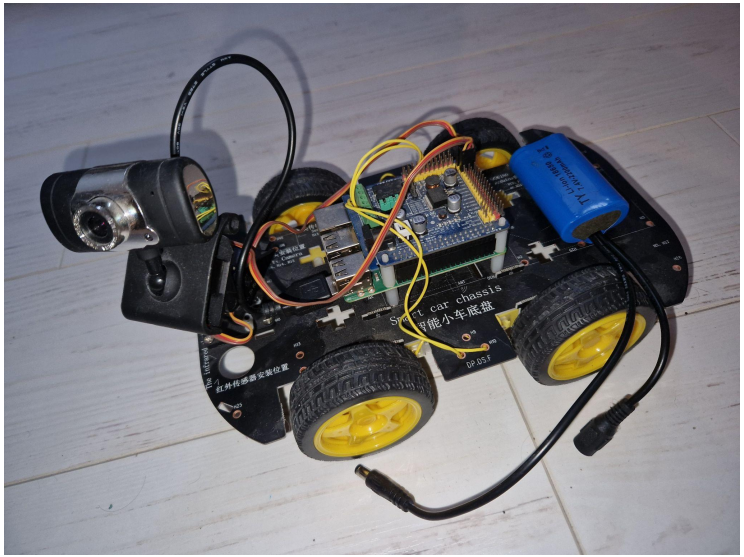


1. Bardzo łatwy start przy znajomości C/Python (wystarczy kilka dni, żeby być w miarę produktywnym z Go)
2. Świetne wsparcie dla deweloperów (biblioteka standardowa, narzędzia dostępne w pakiecie z językiem)
3. Możliwość użycia tych samych technologii do pisania klienta i backendu (do pewnego stopnia); możliwość dzielenia kodu
4. Łatwość wystartowania projektu
    - Z góry ustalony “coding standard”
    - Narzędzia dostępne z językiem
5. Szybkość pisania kodu (szczególnie aplikacje wielowątkowe i komunikacja sieciowa)
6. Adopcja i rozwój języka

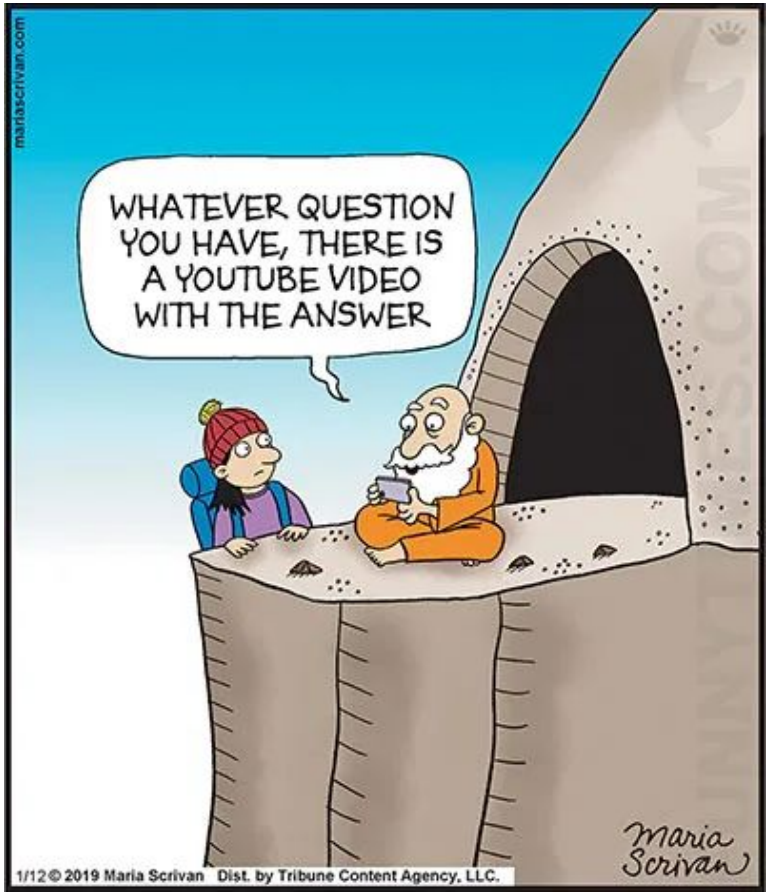
Demo

- mender.io
- GoCar(t) ™
  - https://github.com/mendersoftware/thermostat





Q&A


mariascrivan.com
WHATEVER QUESTION YOU HAVE, THERE IS A YOUTUBE VIDEO WITH THE ANSWER
1/12 © 2019 Maria Scrivan Dist. by Tribune Content Agency, LLC.
Maria Scrivan