INTRODUCCIÓN A LA CLASE

GO WEB

Objetivos de esta clase

- Entender el concepto de arquitectura de software.
- Entender la arquitectura por capas de un proyecto en Go.
- Aplicar los conceptos anteriores en un proyecto.

CAPAS DE APLICACIÓN

GO WEB

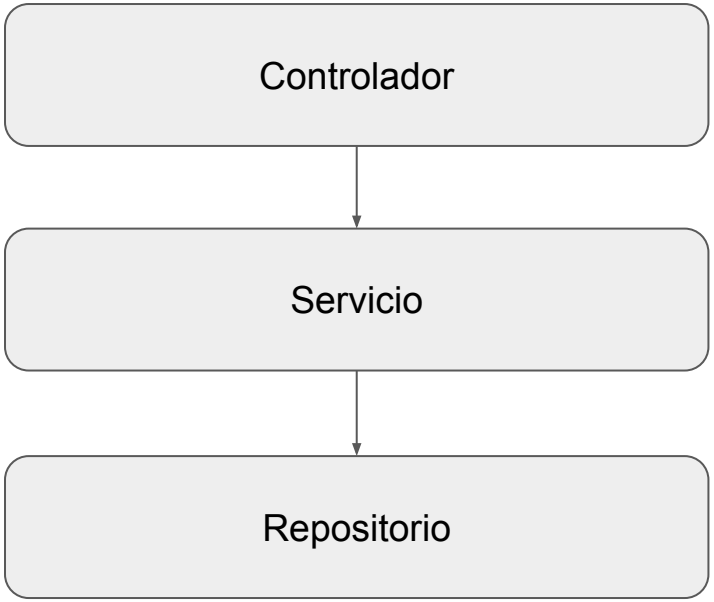
// ¿Qué son las capas de una aplicación web?

“Dividir una aplicación web en capas nos permite dividir las diferentes responsabilidades en diferentes niveles. Por ejemplo: validación de datos de la petición, lógica de negocio o acceso a la Base de Datos.”

Capas de una aplicación

Dividiremos nuestra aplicación en 3 capas:

- Controlador
- Servicio
- Repositorio



Controlador

// ¿Qué es la capa de Controlador?

IT BOARDING

BOOTCAMP

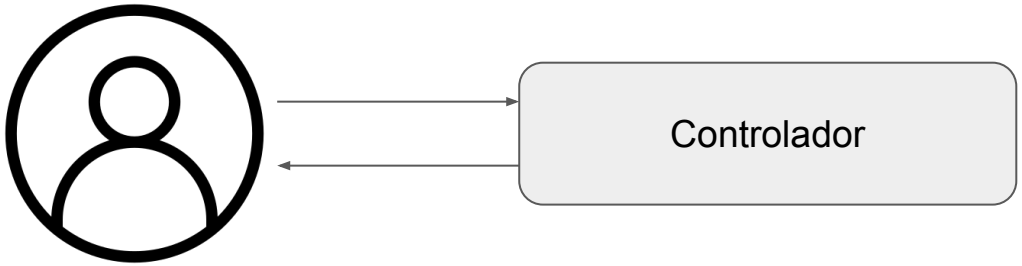
// ¿Qué es la capa de Controlador?

“El controlador se encarga de recibir la petición del cliente, validar valores requeridos, ejecutar los diferentes servicios y retornar una respuesta.”

Controlador

Todas las peticiones que reciba nuestra aplicación web deben pasar por la capa controlador.

El controlador se encarga de verificar la petición y si encuentra algún problema devuelve una respuesta informando al cliente.



Controlador

En caso que la información de la petición sea correcta, el controlador le pasa la tarea a la capa de servicio.

Una vez que el servicio termina, le devuelve la respuesta al cliente.

Servicio

// ¿Qué es la capa de Servicio?

IT BOARDING

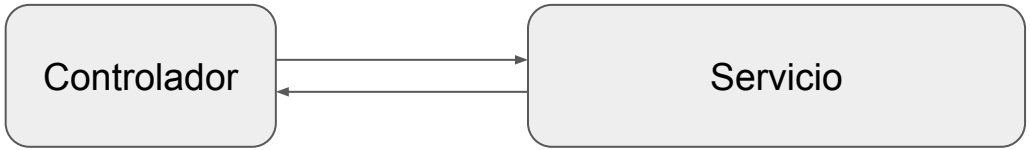
BOOTCAMP

// ¿Qué es la capa de Servicio?

“La capa de servicio es la que se encarga de realizar las funciones principales de la aplicación: procesamiento de datos, implementación de funciones de negocio y administración de recursos externos, por ejemplo, base de datos o apis.”

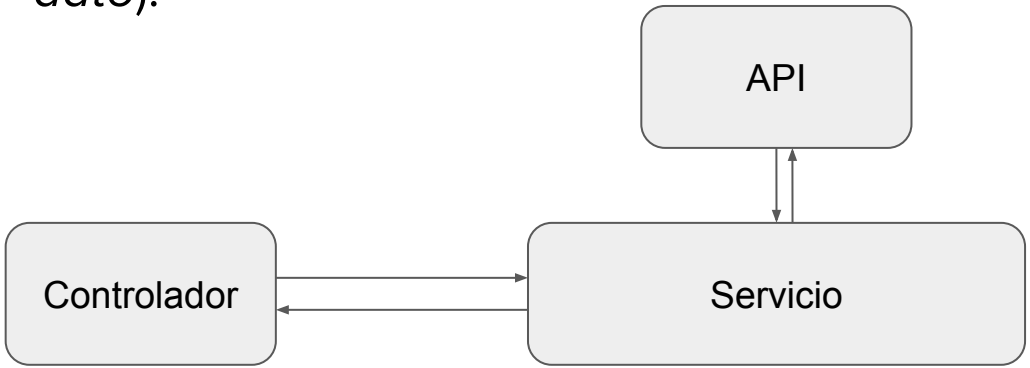
Servicio - Procesamiento de datos

El servicio recibe los datos enviados a través del controlador y se encarga de realizar las funcionalidades principales de la aplicación con la lógica de negocio. *No debemos enviarle el* ***request*** *sino los datos necesarios para* realizar el procesamientos de datos y enviar una respuesta al controlador. El controlador espera recibir el dato que está necesitando o un error en caso que haya surgido algún problema en el Servicio.



Servicio - Adm recursos externos

Se encarga de administrar los recursos externos, como enviar peticiones a una API y recibir una respuesta (ya sea un *error* o un *dato*).



Servicio - Adm recursos externos

En caso de necesitar información de la base de datos, el Servicio envía su petición al Repositorio.
El repositorio se encarga de enviarle una respuesta al Servicio, ya sea el dato que necesita el Servicio o un error.

Repositorio

// ¿Qué es la capa de Repositorio?

IT BOARDING

BOOTCAMP

// ¿Qué es la capa de Repositorio?

“La capa de repositorio se encarga de abstraer el acceso a los datos, siendo el encargado de interactuar con la base de datos.”

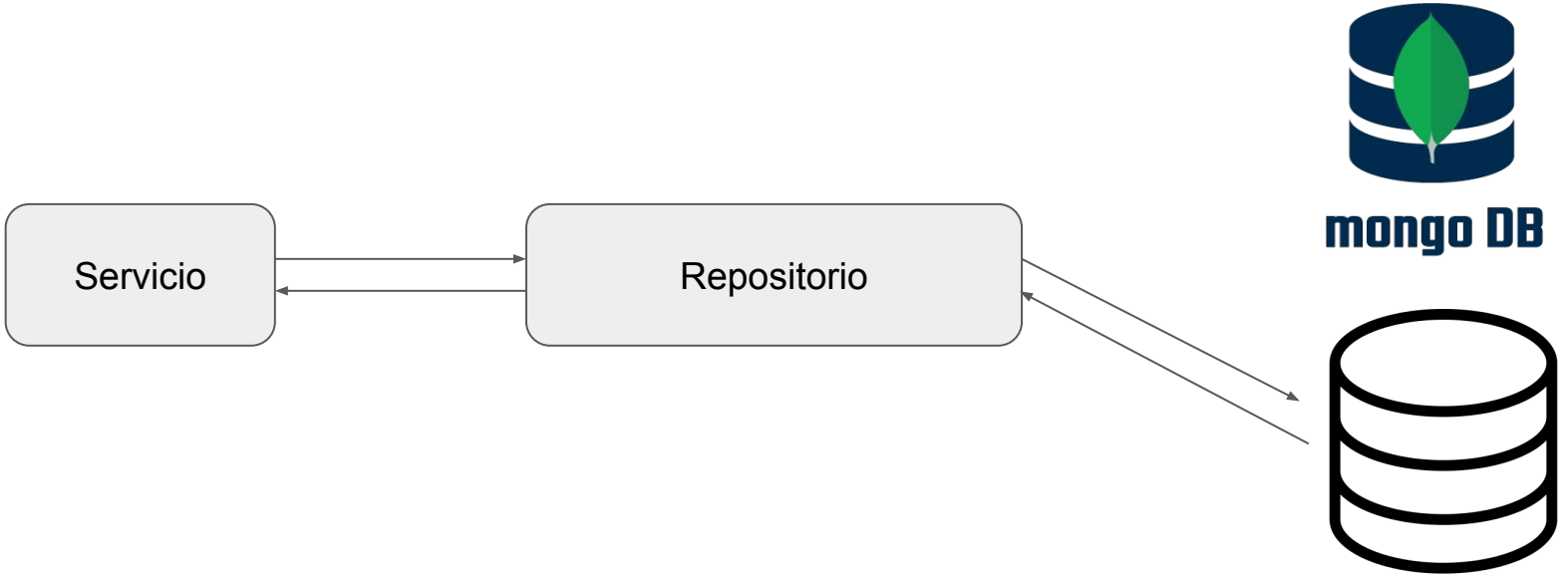
Repositorio

Siempre que el servicio requiera información de la base de datos, deberá pedirla a través de la capa de Repositorio.

La capa de Repositorio es la que se encarga de realizar las operaciones en la base de datos y devolverle la información al Servicio.

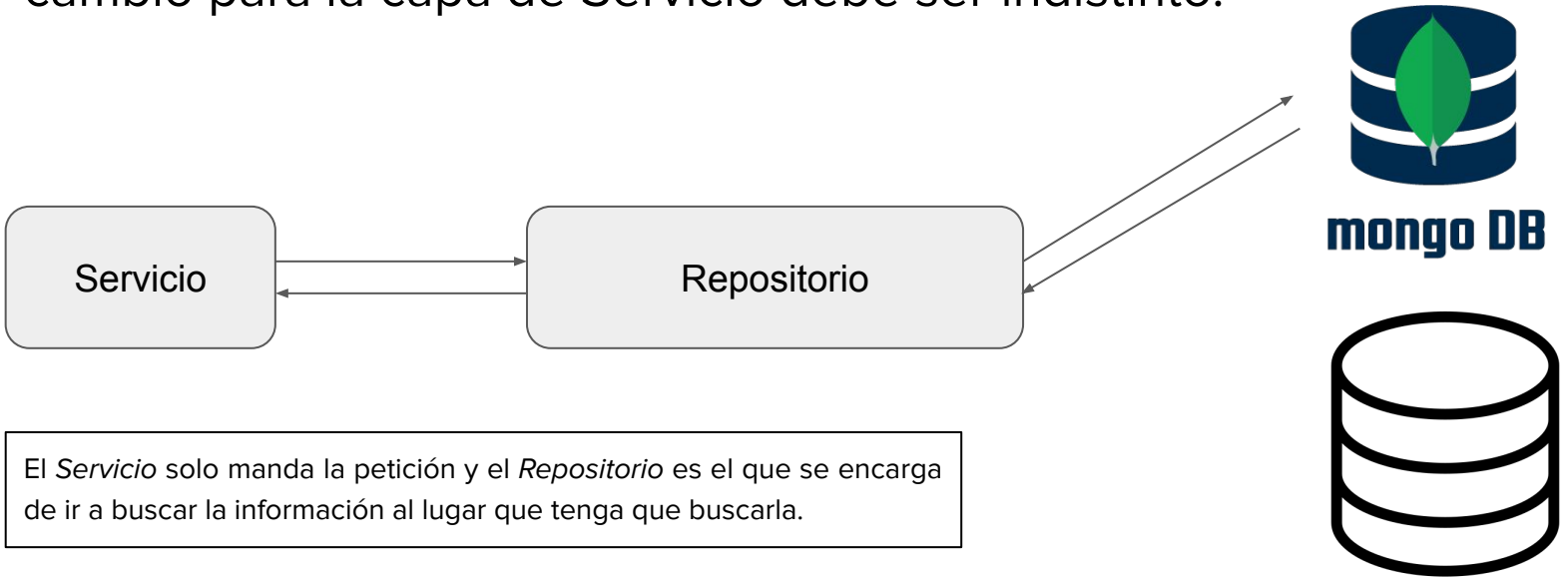
Repositorio

Si nosotros necesitamos hacer un cambio de base de datos, como pasar de una base de datos relacional a una no relacional, como mongoDB...



Repositorio

El cambio solo lo debemos hacer en la capa de Repositorio. Ese cambio para la capa de Servicio debe ser indistinto.



El *Servicio* solo manda la petición y el *Repositorio* es el que se encarga de ir a buscar la información al lugar que tenga que buscarla.

// Para concluir

Es importante dividir nuestra aplicación en capas, de esta forma cada capa tiene su responsabilidad y si se requiere hacer un cambio en una capa, el resto no se vería afectado.

¡Continuemos aprendiendo!

ESTRUCTURA DE PROYECTO

GO WEB

// ¿Cómo podemos estructurar nuestro proyecto?

“Para estructurar un proyecto lo que hacemos es separarlo en paquetes, de esta forma podemos reutilizar los paquetes en otro proyecto o separarlos del proyecto de manera sencilla.”

// ¿De qué forma podemos estructurarlo?

Podemos estructurarlo de 2 formas:

- Por capa
- Por dominio

Estructura por Capa

Para estructurar el proyecto por capas lo que hacemos es generar un paquete por cada capa (Controlador, Servicio y Repositorio).

Si tenemos varias entidades como Producto y Empleado, todas sus capas estarían en cada paquete.

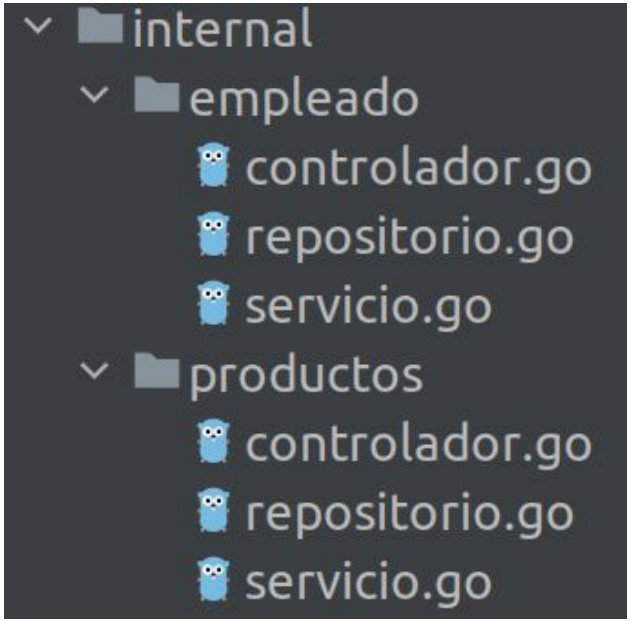
La desventaja es que si quisiéramos quitar la entidad producto para implementarlo en otro microservicio, deberíamos modificar todos los paquetes.

Estructura por Dominio

Para estructurar el proyecto por dominio lo que hacemos es generar un paquete por cada entidad.

Cada paquete tendrá todas las capas de la entidad.

De esta forma, si quisiéramos quitar la entidad producto para implementarlo en otro microservicio, el cambio sería muy sencillo.



Estructurar proyecto en Go

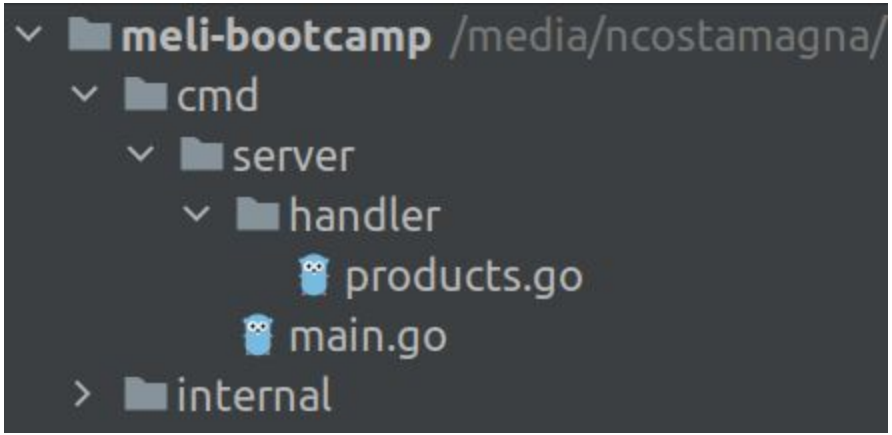
IT BOARDING

BOOTCAMP

Estructurar proyecto en Go

Vamos a estructurar nuestro proyecto por dominio. Lo primero que haremos es generar 2 directorios:

- el directorio **cmd/server**, donde estará nuestro main y los controladores
- el directorio **internal**, donde estarán nuestros paquetes (servicio y repositorio)

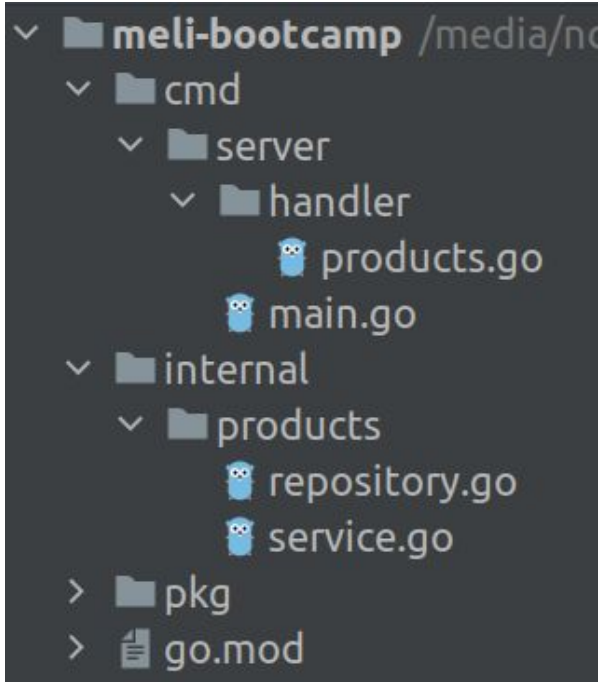


Dominios

Dentro del directorio **internal**, implementaremos nuestro paquete productos con sus capas (servicio y repositorio).

Dentro del directorio internal deberán ir los paquetes que no requieran de paquetes externos (el controlador va a requerir de la funcionalidad de gin).

Haremos la funcionalidad para guardar un producto y obtener todos los productos.



Repositorio en Go

IT BOARDING

BOOTCAMP

Implementar Entidades

Empezaremos con implementar el repositorio. Lo primero que haremos es declarar las entidades que utilizaremos: *Producto*, *Productos* y el último *ID* almacenado.

{}

```
type Product struct {
   ID       int     `json:"id"`
   Name     string  `json:"nombre"`
   Type     string  `json:"tipo"`
   Count    int     `json:"cantidad"`
   Price    float64 `json:"precio"`
}

var ps []Product
var lastID int
```

Estructura Repositorio

Implementaremos la interface **Repositorio** con sus métodos y una función que nos devuelva el repositorio que se utilizará.

```
{}
```

```go
type Repository interface{
   GetAll() ([]Product, error)
   Store(id int, nombre, tipo string, cantidad int, precio float64) (Product, error)
   LastID() (int, error)
}

type repository struct {}

func NewRepository() Repository {
   return &repository{}
}
```

Métodos del Repositorio

Implementaremos los métodos:

- **GetAll**: Obtener todos los Productos.
- **LastID**: Obtener el último ID almacenado.

```
{}
```

```go
func (r *repository) GetAll() ([]Product, error) {
    return ps, nil
}

func (r *repository) LastID() (int, error) {
    return lastID, nil
}
```

Métodos del Repositorio

Implementaremos los métodos:

- **Store**: Guardará la información de producto, asignará el último ID a la variable y nos retorna la entidad Producto.

{}

```
func (r *repository) Store(id int, nombre, tipo string, cantidad int, precio float64)
(Product, error) {
    p := Product{id, nombre, tipo, cantidad, precio}
    ps = append(ps, p)
    lastID = p.ID
    return p, nil
}
```

Servicio en Go

IT BOARDING

BOOTCAMP

Estructura Servicio

Implementaremos la interface **Servicio** con sus métodos y una función que reciba un **Repositorio** y nos devuelva el servicio que se utilizará, instanciado.

{}

```
type Service interface {
   GetAll() ([]Product, error)
   Store(nombre, tipo string, cantidad int, precio float64) (Product, error)
}
type service struct {
   repository Repository
}
func NewService(r Repository) Service {
   return &service{
       repository: r,
   }
}
```

Método obtener productos

Implementaremos el método **GetAll** que se encargará de pasarle la tarea al **Repositorio** y nos retorna un array de Productos.

{}

```
func (s *service) GetAll() ([]Product, error) {
    ps, err := s.repository.GetAll()
    if err != nil{
        return nil, err
    }

    return ps, nil
}
```

Método guardar producto

El método **Store** se encargará de pasarle la tarea de obtener el último ID y guardar el producto al **Repositorio**, el servicio se encargará de incrementar el ID.

{}

```
func (s *service) Store(nombre, tipo string, cantidad int, precio float64) (Product, error) {
    lastID, err := s.repository.LastID()
    if err != nil{
        return Product{}, err
    }

    lastID++

    producto, err := s.repository.Store(lastID,nombre, tipo, cantidad, precio)
    if err != nil{
        return Product{}, err
    }

    return producto, nil
}
```

Handler en Go

IT BOARDING

BOOTCAMP

Implementar Request

Se debe implementar el controlador de productos. Primero generamos el *request* con los valores que esperamos recibir de la petición e importamos nuestro paquete interno de productos.

```
{}
```

```go
import (
    "github.com/gin-gonic/gin"
    "github.com/meli-bootcamp/internal/products"
)
type request struct {
    Name   string  `json:"nombre"`
    Type   string  `json:"tipo"`
    Count  int     `json:"cantidad"`
    Price  float64 `json:"precio"`
}
```

Estructura Controlador

Implementaremos la estructura **Producto** y una función que reciba un **Servicio** (del paquete interno) y devuelva el controlador instanciado.

{}

```
type Product struct {
    service products.Service
}

func NewProduct(p products.Service) *Product {
    return &Product{
        service: p,
    }
}
```

Método obtener productos

El método obtener productos se encargará de realizar las validaciones de la petición, pasarle la tarea al **Servicio** y retornar la respuesta correspondiente al cliente.

{}

```
func (c *Product) GetAll() gin.HandlerFunc {
   return func(ctx *gin.Context) {
       token := ctx.Request.Header.Get("token")
       if token != "123456" {
          ctx.JSON(401, gin.H{
             "error": "token inválido",
          })
          return
       }

       p, err := c.service.GetAll()
       if err != nil {
          ctx.JSON(404, gin.H{
             "error": err.Error(),
          })
          return
       }
       ctx.JSON(200, p)
    }
  }
```

Método guardar

De la misma forma, el método **guardar.**

{}

```
func (c *Product) Store() gin.HandlerFunc {
   return func(ctx *gin.Context) {
       token := ctx.Request.Header.Get("token")
       if token != "123456" {
          ctx.JSON(401, gin.H{ "error": "token inválido" })
          return
       }
       var req request
       if err := ctx.Bind(&req); err != nil {
          ctx.JSON(404, gin.H{
             "error": err.Error(),
          })
          return
       }
       p, err := c.service.Store(req.Name, req.Type, req.Count, req.Price)
       if err != nil {
          ctx.JSON(404, gin.H{ "error": err.Error() })
          return
       }
       ctx.JSON(200, p)
    }
  }
```

Main del programa

IT BOARDING

BOOTCAMP

Importación de dependencias

El main se encontrará en el directorio **cmd/server.**
Importamos las dependencias necesarias.

```
import (
    "github.com/gin-gonic/gin"
    "github.com/meli-bootcamp/cmd/server/handler"
    "github.com/meli-bootcamp/internal/products"
)
```

Main del programa

Instanciamos cada capa del dominio **Productos** y utilizaremos los métodos del controlador para cada endpoint.

{}

```
func main() {
    repo := products.NewRepository()
    service := products.NewService(repo)
    p := handler.NewProduct(service)

    r := gin.Default()
    pr := r.Group("/products")
    pr.POST("/", p.Store())
    pr.GET("/", p.GetAll())
    r.Run()
}
```

Para correr nuestro programa lo hacemos con el comando `go run cmd/server/main.go`.

// Para concluir

Es importante que los métodos de cada capa nos retornen error, para poder ser controlado e informado al cliente.

¡Continuemos aprendiendo!

Gracias.

IT BOARDING

BOOTCAMP

Historial de Cambios

Fecha	Version	Autor	Comentarios
27/06/2021	**0.0.1**	**Nahuel Costamagna**	**Creación de documento**

IT BOARDING

BOOTCAMP