

# Manual técnico

## BackEnd:

### Gorilla Mux:

El paquete gorilla / mux implementa un enrutador de solicitudes y un despachador para hacer coincidir las solicitudes entrantes con su respectivo controlador.

El nombre mux significa "multiplexor de solicitud HTTP". Al igual que el `http.ServeMux` estándar, `mux.Router` compara las solicitudes entrantes con una lista de rutas registradas y llama a un controlador para la ruta que coincide con la URL u otras condiciones. Las principales características son:

1. Implementa la interfaz `http.Handler` para que sea compatible con el estándar `http.ServeMux`.
2. Las solicitudes se pueden hacer coincidir según el host de URL, la ruta, el prefijo de la ruta, los esquemas, los valores de encabezado y consulta, los métodos HTTP o el uso de comparadores personalizados.
3. Los hosts de URL, las rutas y los valores de consulta pueden tener variables con una expresión regular opcional.
4. Las URL registradas se pueden crear o "revertir", lo que ayuda a mantener las referencias a los recursos.
5. Las rutas se pueden utilizar como subenrutadores: las rutas anidadas solo se prueban si la ruta principal coincide. Esto es útil para definir grupos de rutas que comparten condiciones comunes como un host, un prefijo de ruta u otros atributos repetidos. Como beneficio adicional, esto optimiza la coincidencia de solicitudes.

### Instalación:

```
go get -u github.com/gorilla/mux
```

### Ejemplo:

```
func main() {  
    r := mux.NewRouter()  
    r.HandleFunc("/", HomeHandler)  
    r.HandleFunc("/products", ProductsHandler)  
    r.HandleFunc("/articles", ArticlesHandler)  
    http.Handle("/", r)  
}
```

Aquí registramos tres rutas que mapean rutas de URL a controladores. Esto es equivalente a cómo funciona `http.HandleFunc()`: si una URL de solicitud entrante coincide con una de las rutas, el controlador correspondiente se llama pasando (`http.ResponseWriter`, `* http.Request`) como parámetros

Las rutas pueden tener variables. Se definen utilizando el formato `{nombre}` o `{nombre: patrón}`. Si no se define un patrón de expresión regular, la variable coincidente será cualquier cosa hasta la siguiente barra. Por ejemplo:

```
r := mux.NewRouter()
r.HandleFunc("/products/{key}", ProductHandler)
r.HandleFunc("/articles/{category}/", ArticlesCategoryHandler)
r.HandleFunc("/articles/{category}/{id:[0-9]+}", ArticleHandler)
```

Los nombres se utilizan para crear un mapa de variables de ruta que se pueden recuperar llamando a `mux.Vars()`:

```
func ArticlesCategoryHandler(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    w.WriteHeader(http.StatusOK)
    fmt.Fprintf(w, "Category: %v\n", vars["category"])
}
```

#### **Godror:**

#### **Instalación:**

`go get github.com/godror/godror`

#### **Conexión:**

Para conectarse a la base de datos Oracle, use `sql.Open("godror", dataSourceName)`, donde `dataSourceName` es una lista de parámetros codificada en `logfmt`. Especifique al menos "usuario", "contraseña" y "connectString". Por ejemplo:

```
db, err := sql.Open("godror", `user="scott" password="tiger" connectionString="dbhost:1521/orclpdb1"`)
```

`ConnectionString` puede ser CUALQUIER COSA que SQL \* Plus u Oracle Call Interface (OCI) acepten: un nombre de servicio, una cadena de Easy Connect como `host: puerto / nombre_servicio`, o un descriptor de conexión como `(DESCRIPCIÓN = ...)`.

#### **Ejemplo:**

```

$ cat db.go
package main

import (
    "fmt"
    "database/sql"
    _ "github.com/godror/godror"
)

func main(){

    db, err := sql.Open("godror", "scott/bar@adb")
    if err != nil {
        fmt.Println(err)
        return
    }
    defer db.Close()

    rows,err := db.Query("select sysdate from dual")
    if err != nil {
        fmt.Println("Error running query")
        fmt.Println(err)
        return
    }
    defer rows.Close()

    var thedate string
    for rows.Next() {

        rows.Scan(&thedata)
    }
    fmt.Printf("The date is: %s\n", thedate)
}
$ █

```

### Database/sql:

El paquete sql proporciona una interfaz genérica en torno a bases de datos SQL (o similares a SQL).

### Métodos:

#### Open():

Abrir abre una base de datos especificada por su nombre de controlador de base de datos y un nombre de fuente de datos específico del controlador, que generalmente consta de al menos un nombre de base de datos e información de conexión.

La mayoría de los usuarios abrirán una base de datos a través de una función auxiliar de conexión específica del controlador que devuelve un \* DB. No se incluyen controladores de base de datos en la biblioteca estándar de Go. Consulte <https://golang.org/s/sqldrivers> para obtener una lista de controladores de terceros.

Open puede simplemente validar sus argumentos sin crear una conexión a la base de datos. Para verificar que el nombre de la fuente de datos sea válido, llame a Ping.

La base de datos devuelta es segura para el uso simultáneo de múltiples goroutines y mantiene su propio grupo de conexiones inactivas. Por lo tanto, la función Abrir debe llamarse solo una vez. Rara vez es necesario cerrar una base de datos.

```
func Open(driverName, dataSourceName string) (*DB, error)
```

### Query():

QueryContext ejecuta una consulta que devuelve filas, normalmente un SELECT. Los argumentos son para cualquier parámetro de marcador de posición en la consulta.

```
func (c *Conn) QueryContext(ctx context.Context, query string, args ...interface{}) (*Rows, error)
```

### Exec():

Exec ejecuta una consulta sin devolver ninguna fila. Los argumentos son para cualquier parámetro de marcador de posición en la consulta.

```
func (db *DB) Exec(query string, args ...interface{}) (Result, error)
```

## FrontEnd:

### Fullcalendar:

FullCalendar se integra a la perfección con el marco de React JavaScript. Proporciona un componente que coincide exactamente con la funcionalidad de la API estándar de FullCalendar.

Esto es más que un mero "conector". Le dice al paquete principal FullCalendar que comience a renderizar con los nodos DOM virtuales de React en lugar de los nodos de Preact que usa normalmente, transformando FullCalendar en un componente de React "real".

### Instalación:

```
npm install --save @fullcalendar/react @fullcalendar/daygrid
```

### Ejemplo:

```

import React from 'react'
import FullCalendar from '@fullcalendar/react'
import dayGridPlugin from '@fullcalendar/daygrid'

export default class DemoApp extends React.Component {
  render() {
    return (
      <FullCalendar
        plugins={[ dayGridPlugin ]}
        initialView="dayGridMonth"
      />
    )
  }
}

```

## Calendar API

Esto es especialmente útil para controlar la fecha actual. El accesorio `initialDate` establecerá la fecha inicial del calendario, pero para cambiarlo después de eso, deberá confiar en los métodos de navegación por fechas.

Para hacer algo como esto, deberá obtener la referencia del componente (abreviatura de "referencia"). Una vez que lo hace, llama al método `getApi` de la instancia del componente "actual":

```

export default class DemoApp extends React.Component {

  calendarRef = React.createRef()

  render() {
    return (
      <FullCalendar ref={this.calendarRef} plugins={[ dayGridPlugin ]} />
    )
  }

  someMethod() {
    let calendarApi = this.calendarRef.current.getApi()
    calendarApi.next()
  }
}

```

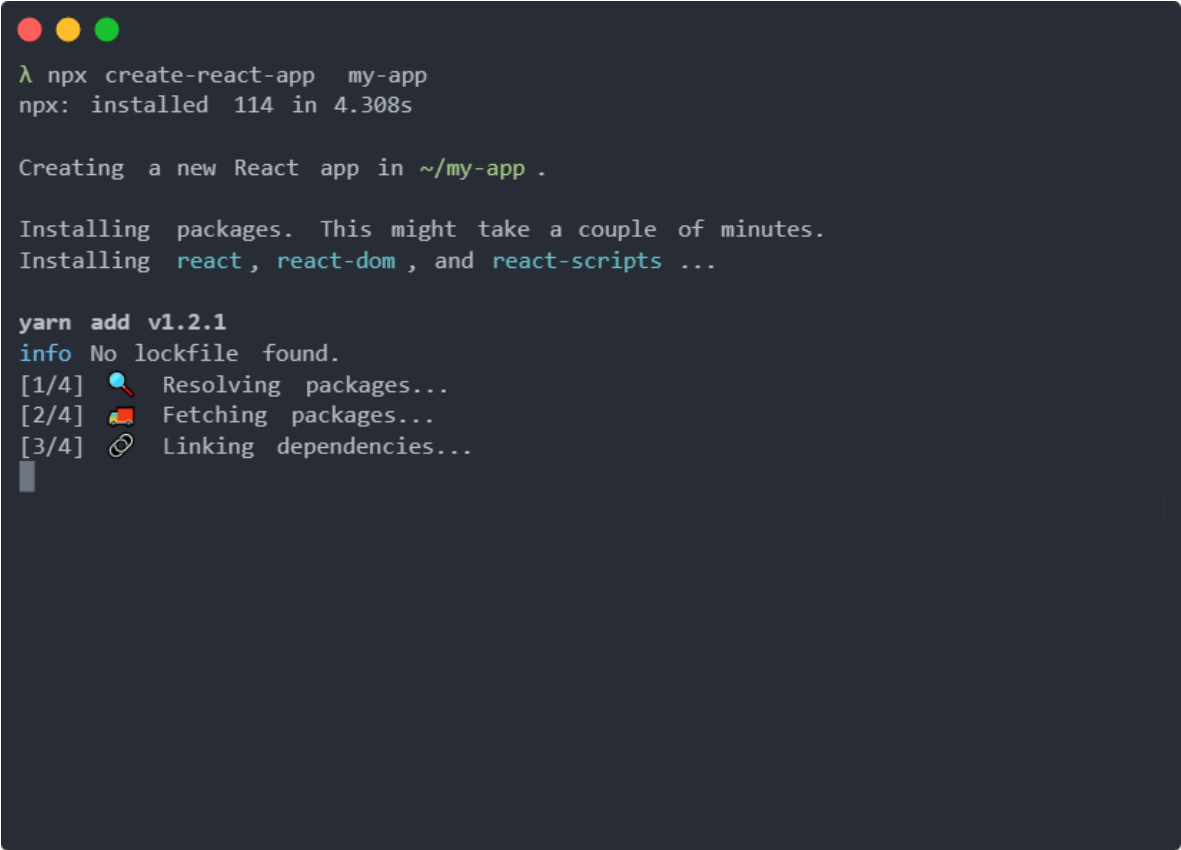
## ReactStrap:

Reactstrap es una biblioteca de componentes para React. Proporciona componentes Bootstrap incorporados que facilitan la creación de una interfaz de usuario con componentes autónomos que proporcionan flexibilidad y validaciones incorporadas. Reactstrap es fácil de usar y es compatible con Bootstrap 4.

## Instalación:

```
npm install --save reactstrap react react-dom
```

## Creación:



```
λ npx create-react-app my-app
npx: installed 114 in 4.308s

Creating a new React app in ~/my-app .

Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts ...

yarn add v1.2.1
info No lockfile found.
[1/4] 🔍 Resolving packages...
[2/4] 📦 Fetching packages...
[3/4] 🔗 Linking dependencies...
```