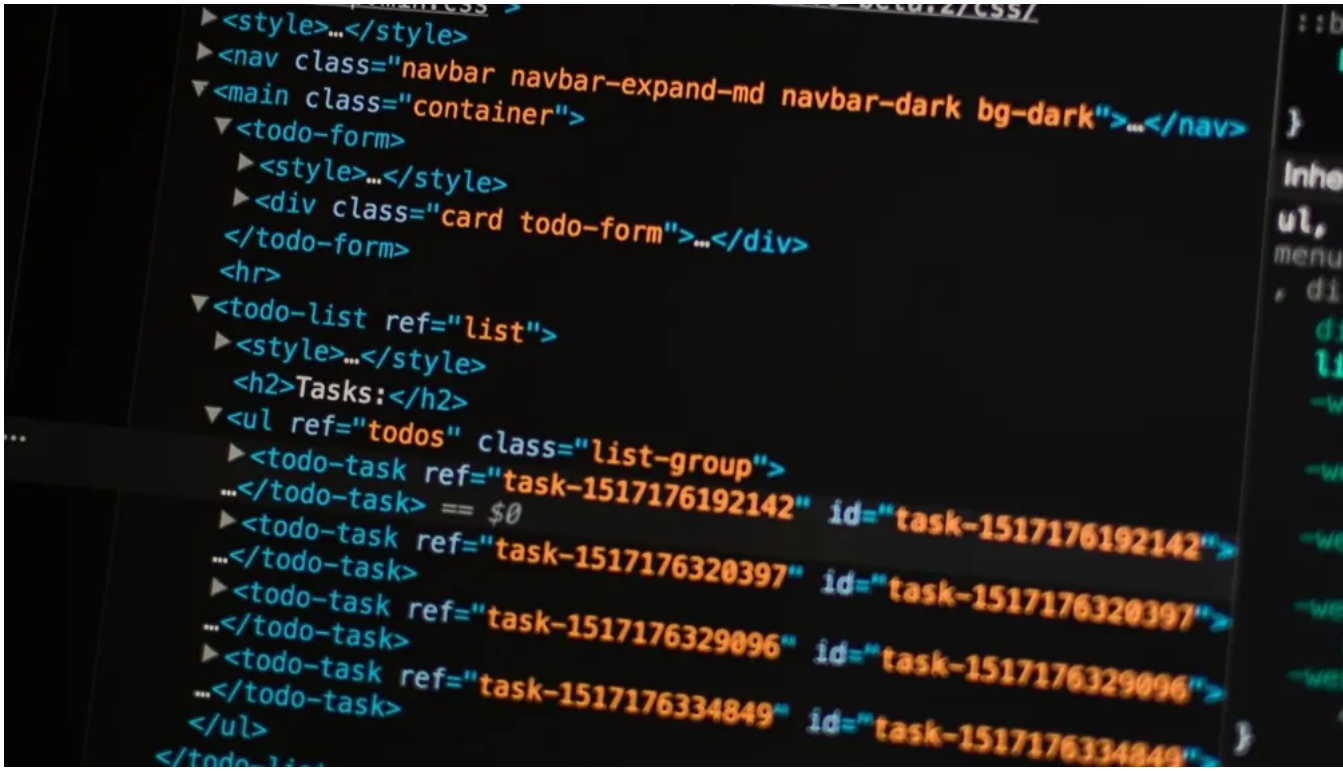


Web scraping, also known as web data extraction, is an automated method of extracting data or content from web pages.

Web scrapers automate data extraction without human interference. A scraper accesses a webpage by sending HTTP requests, much like a web browser does. However, instead of displaying the HTML it fetches, it processes it according to your instructions and stores the result.

Web scrapers come in handy for fetching data from websites that do not provide APIs. They are popular in fields like data science, cybersecurity, frontend, and backend development.

Web Scraping in Go

In Go, there are various web scraping packages. The popular ones include goquery, Colly, and ChromeDP.

net/html.

Installing goquery

Run the command below in your terminal to install goquery. If you encounter any errors, try updating your Go version.

```
go get github.com/PuerkitoBio/goquery\n
```

The Web Scraping Process

You can divide the overall scraping process into three smaller tasks:

1. Making HTTP Requests.
2. **Using selectors and locators** to get the required data.
3. Saving data in a database or data structures for further processing.

Making HTTP Requests in Go

You can send HTTP requests using the **net/http** package, that the Go standard library includes.

```
package main\n \nimport "net/http"\nimport "log"\nimport "fmt"\n \nfunc main() {\n webUrl :=
"https://news.ycombinator.com/"\n response, err:= http.Get(webUrl)\n \n if err != nil {\n log.Fatalln(err)\n }
else if response.StatusCode == 200 {\n fmt.Println("We can scrape this")\n } else {\n log.Fatalln("Do not scrape
this")\n }\n}\n
```

http.Get returns a response body and an error. **response.StatusCode** is the request-response status code.

On making HTTP requests, if the **response status code** is **200** you can proceed to scrape the website.

Getting the Required Data Using goquery

Getting the Website HTML

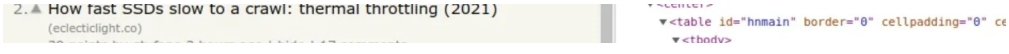
First, you have to parse the plain HTML from the response (**response.body**) to get a complete document object representing the webpage:

```
document, err := goquery.NewDocumentFromReader(response.Body)\n \nif err != nil {\n log.Fatalln(err)\n}\n
```

You can now use the document object to access the structure and content the webpage contains.

Selecting Required Elements From the HTML

You will need to inspect the webpage to check the structure of the data you need to extract. This will help you to construct a selector to access it.



Using selectors and locators, you can extract the HTML you need using the **Find** method of the document object.

The **Find** method takes a CSS selector to locate the element that contains the data you require:

```
document.Find("tr.athing")\n
```

The code above returns only the first HTML element matching the selector, or an empty list if there wasn't a match at all.

Selecting Multiple Elements From HTML

Most of the time, you'll want to fetch all the HTML elements that match your selector.

You can select all matching elements in the HTML using the **Each** method of the value that **Find()** returns. The **Each** method takes in a function with two parameters: an index and a selector of type ***goquery.Selection**.

```
document.Find("tr.athing").Each(func(index int, selector *goquery.Selection) {\n /* Process selector here */\n})
\n
```

In the function body, you can select the specific data you want from the HTML. In this case, you need the links and titles of every post the page lists. Use the **Find** method of the selector parameter to narrow down the set of elements and extract text or attribute values.

```
document.Find("tr.athing").Each(func(index int, selector *goquery.Selection) {\n title :=
selector.Find("td.title").Text()\n link, found := selector.Find("a.titlelink").Attr("href")\n})
```

The code above calls the **Text** method of the result from **selector.Find** to extract the contents of a table cell. Selecting attributes—like link and image URLs—requires you to use the **Attr** method. This method also returns a value indicating if the attribute exists at all.

The process is the same for selecting whatever elements and attributes off a web page.

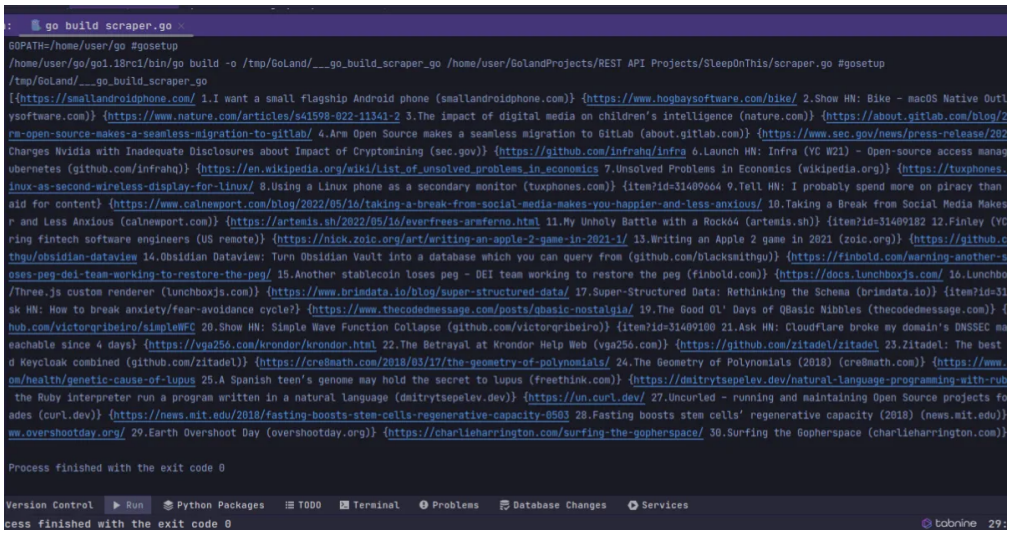
The **Find** method is very powerful, allowing a wide range of operations to select and locate HTML elements. You can explore these in the goquery documentation.

Saving the Scraped Data

The link attribute and title are strings that you can assign to variables. In real scenarios, you will be saving to a database or a data structure for manipulation. Often, a simple custom struct will suffice.

Create a struct with fields title and link and a slice of structs to hold the struct type.

```
fmt.Println(info)\n
```



It is reasonable to save the scraped data in a local cache so that you do not hit the server of the webpage more than you need to. This will not only reduce traffic but speed up your app since it is faster to retrieve local data than it is to make requests and scrape websites.

There are many database packages in Go that you could use to save the data. The **database/sql** package supports SQL databases. There are also NoSQL database clients like the **MongoDB Go driver**, and serverless databases like FaunaDB using the **FaunaDB driver**.

The Essence of Web Scraping in Go

If you're trying to scrape data from a website, goquery is an excellent place to begin. But it's a powerful package that can do more than just web scraping. You can find out about more of its functionality in the official project documentation.

Web scraping is an important skill across various technology fields and it will come in handy during many of your projects.

RELATED TOPICS

PROGRAMMING WEB DEVELOPMENT PROGRAMMING

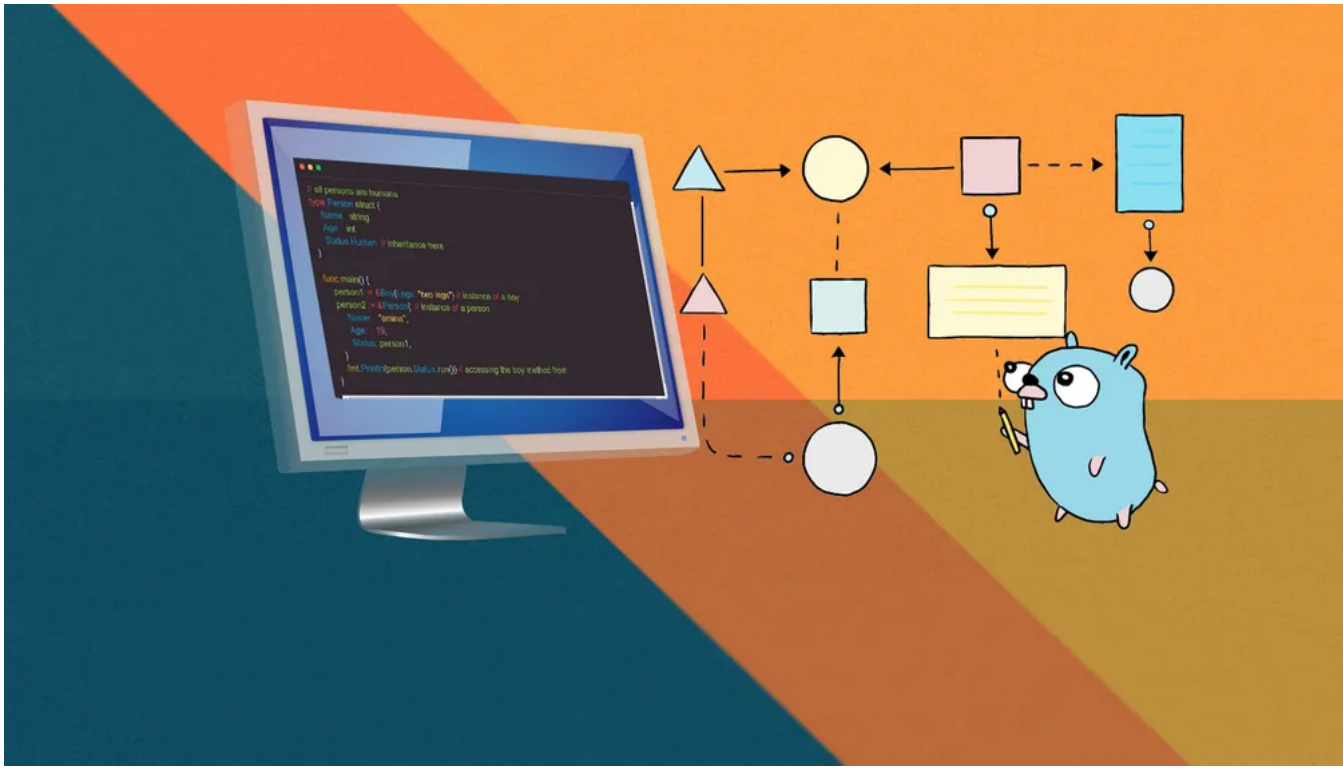
ABOUT THE AUTHOR

Ukeje Chukwuemeriwo Goodness (29 Articles Published)

Goodness is a mechanical engineering student and software developer passionate about cloud technologies and the Go programming language.

Getting to grips with object oriented programming in Go isn't as tough as you think.

BY **UKEJE CHUKWUEMERIWO GOODNESS** PUBLISHED MAY 14, 2022



Readers like you help support MUO. When you make a purchase using links on our site, we may earn an affiliate commission. ***Read More.***

Object-Oriented Programming (OOP) is a programming paradigm based on objects as the central concept. In OOP, code is formatted based on functionality, enabling code maintenance, abstraction, reusability, efficiency, and numerous functionality on the object.

The object has attributes (variables) that define its characteristics, properties, and methods (functions) which define the actions (procedures) and behaviors of the object.

Object-oriented programming in Go is different from other languages. Object-Oriented concepts are implemented in Go using structs, interfaces, and custom types.

Customizing Types in Go

Custom types make it easy to group and identify similar code for reuse.

The code for declaring custom types is:

```
type typeName dataType \n
```

On creating a custom type and assigning a variable, you can check the type using **reflect.TypeOf()** which takes in a variable and returns the type of the variable.

```
import( "fmt"\n "reflect")\ntype two int // creates type "two"\nvar number two // variable of type "two"\nfmt.Println(reflect.TypeOf(number))\n
```

The **number** variable is a type of **two** which is an integer. You can go further to create more of the custom type.

Creating Structs in Go

Structs (structures) are the blueprints for object-oriented programming in Go. Structs are user-defined collections of fields.

A struct can contain a variety of data types, including compound types and methods.

You can create a struct using this syntax:

```
type StructName struct {\n// some code\n}\n
```

Conventionally, struct names are usually capitalized and camel-cased for readability.

The struct type takes in field names and data types. Structs can take in any Go data type, including custom types.

```
type User struct {\n field1 string\n field2 int\n fieldMap map[string]int\n}\n
```

You can instantiate a struct type by assigning the struct as a variable.


```
instance := User{\n// some code\n}\n
```

The struct instance can be populated with fields on instantiation as defined at initialization or set to null.

```
instance := User{\n field1: "a string field",\n field2: 10,\n fieldMap: map[string]int{},\n }\n
```

Accessing Struct Elements

You can access the fields of a struct instance using a dot notation to the field.

```
fmt.Println("Accessing a field of value", instance.field2)\n
```

This outputs the **field2** of the struct instance instantiated.

Assigning Methods to Structs

Functions(methods) are assigned to struct types by specifying a receiver name and the struct name before the function name as shown in the syntax below.

```
func (receiver StructName) functionName() {\n // some code\n}\n
```

The method **functionName** can only be used on the struct type specified.

Implementing Inheritance in Go

Inheritance is **the ability of objects and types to access and use methods and attributes of other objects.** Go doesn't have Inheritance as a feature, but you can use compositions. In Go, composition entails referring to a superstruct (the struct to be inherited) in a substruct by providing the superstruct's name to the substruct.

Using the struct example above:

Encapsulating Type Fields in Go

Encapsulation, also known as “information hiding,” is a **technique of bundling the methods and attributes of an object into units** to restrict the use and access except specified (enabling read/write privileges).

Encapsulation is implemented in Go using exported and unexported identifiers in packages.

Exported Identifiers (Read and Write)

Exported identifiers are exported from their defined packages and access to other programs. Capitalizing a field identifier exports the field fo.

```
type User struct {\n Field1 string\n Field2 int\n FieldMap map[string]int\n}\ntype User2 struct {\n User\n}\n
```

Unexported Identifiers (Read-Only)

Unexported identifiers are not exported from the defined package and are conventionally lowercased.

```
type User struct {\n field1 string\n field2 int\n fieldMap map[string]int\n}\ntype User2 struct {\n User\n}\n
```

The concept of exported and unexported identifiers also applies to the methods of an object.

Polymorphism in Go

Polymorphism is a technique used to give different forms to an object for flexibility.

Go implements polymorphism using interfaces. Interfaces are custom types used to define method signatures.

Declaring Interfaces

Declaring interfaces is similar to declaring structs. However, interfaces are declared using the **interface** keyword.

```
type InterfaceName interface{\n //some methods\n}\n
```

Interface declarations contain methods that are to be implemented by struct types.

Implementing Interfaces in Structs

The types that implement the interface have to be declared after which the methods of the type implement the interface.


```
// The Interface\ntype Color interface{\n Paint() string\n}\n// Declaring the structs\ntype Green struct {\n // some struct specific code\n}\ntype Blue struct {\n // some specific code\n}\n
```

The code snippet above has a **Color** interface declared with a **Paint** method to be implemented by the **Green** and **Blue** struct types.

Abstraction is **the process of hiding unimportant methods and attributes of a type**, making it easier to secure parts of the program from abnormal, unintended use.

Go doesn't have abstraction implemented right off the bat; however, you can work our way through implementing abstraction using interfaces.

```
// humans can run\ntype Human interface {\n run() string\n}\n// Boy is a human with legs\ntype Boy struct {\n Legs string\n}\n// a method on boy implements the run method of the Human interface\nfunc (h Boy) run() string {\n return h.Legs\n}\n
```

The code above creates a **Human** interface with a **run** interface that returns a string. The **Boy** type implements the **run** method of the **Human** interface and returns a string on instantiation.

One of the ways to implement abstraction is by making a struct inherit the interface whose methods are to be abstracted. There are many other approaches, but this is the easiest.


```
type Person struct {\n Name string\n Age int\n Status Human \n}\nfunc main() {\n person1 := &Boy{Legs: "two legs"}\n person2 := &Person{ // instance of a person\n Name: "amina",\n Age: 19,\n Status: person1,\n }\n fmt.Println(person.Status.run()) \n}\n
```

example-of-abstraction-in-Go-3

The **Person** struct inherits the **Human** interface and can access all its methods using the variable **Status** inheriting the interface.

On instantiation by reference(using a pointer), the instance of the **Person** struct **Person2** references an instance of the **Boy** struct **Person1** and gets access to the methods.

This way, you get to specify specific methods to be implemented by the type.

OOP vs Functional Programming

Object-oriented programming is an important paradigm as it gives you more control of your program and encourages code reuse in ways functional programming doesn't.

the Go programming language.