**Web Scraping in Golang with Colly: 2024 Complete**

Follow this step-by-step tutorial to learn basic to advanced techniques for **scraping data easily in Golang using** popular scraping libraries like Colly and Chromedp.

Let's start!

**Prerequisites**

Before proceeding with this web scraping guide, ensure you have the necessary tools installed.

**Set Up the Environment**

Here are the prerequisites you have to meet for this tutorial:

* [**Go 1.19+**](https://go.dev/dl/): Any version of Go greater than or equal to 1.19 is okay. You'll see version 1.22.0 in action in this tutorial.
* **A Go IDE**: [Visual Studio Code](https://code.visualstudio.com/download) with the [Go extension](https://code.visualstudio.com/docs/languages/go) is recommended.

Open the links above to download, install, and set up the required tools by following their installation wizards.

**Set Up a Go Project**

After installing Go, it's time to initialize your Golang web scraper project. Create a web-scraper-go folder and enter it in your terminal:

Terminal

mkdir web-scraper-go

cd web-scraper-go

Launch a Go module with the following command:

Terminal

go mod init scraper

The init command will initialize a scraper Go module inside your project root folder. You should now see a go.mod file with the following content in your root folder:

go.mod

module scraper

go 1.22.0

Note that the last line can change depending on your Go version.

You're now ready to set up your web scraping Go script. Create a scraper.go file and initialize it as shown below:

scraper.go

package main

import (

"fmt"

)

func main() {

*//... scraping logic*

fmt.Println("Hello, World!")

}

The first line contains the name of the global package, followed by the import of fmt, a Go package for formatting and outputting values. The main() function represents the entry point of any Go program and will contain the Golang web scraping logic.

Run the script to verify that everything works as expected:

Terminal

go run scraper.go

The above code prints:

Output

Hello, World!

You've now set up a basic Golang project. Let's learn how to build your data scraper.

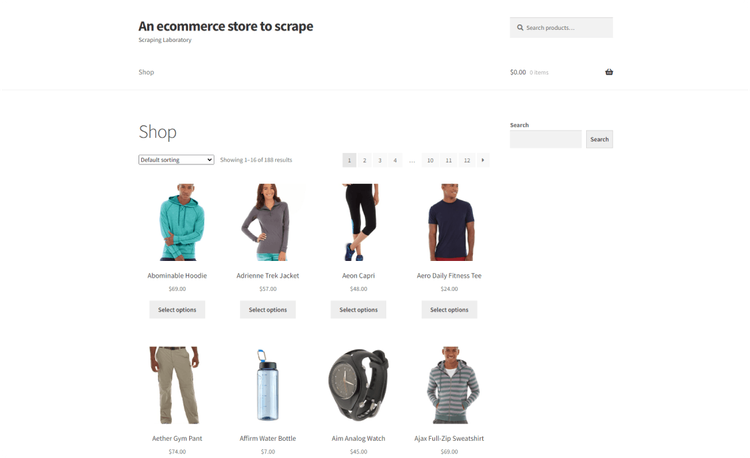
**Frustrated that your web scrapers are blocked once and again?**

ZenRows API handles rotating proxies and headless browsers for you.

[Try for FREE](https://app.zenrows.com/register)

**How to Scrape a Website in Go**

To learn how to scrape a website in Go, let's scrape a [demo e-commerce website](https://www.scrapingcourse.com/ecommerce/), ScrapingCourse. See the target website below:

[[](https://static.zenrows.com/content/large_scrapingcourse_ecommerce_homepage_bf630ec020.png)Click to open the image in full screen](https://static.zenrows.com/content/large_scrapingcourse_ecommerce_homepage_bf630ec020.png)

As you can see above, it's an e-commerce store with paginated product pages. Your mission is to extract product data, including names, prices, URLs, and image sources.

**Step 1: Getting Started With Colly**

[Colly](http://go-colly.org/) is an open-source library for extracting data in Go. Its advanced API allows you to download a page's HTML, automatically parse its content, select HTML elements from the [DOM](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model), and retrieve data from it. Colly is callback-based, providing an efficient and modular way to design your scraping logic.

To start, install Colly and its dependencies:

Terminal

go get github.com/gocolly/colly

The command above creates a go.sum file in your project root and updates the go.mod file with all the required dependencies.

Import Colly in your scraper.go file. Initialize a [NewCollector](https://pkg.go.dev/github.com/gocolly/colly" \l "NewCollector" \t "_blank) object and set the allowed domain:

scraper.go

package main

import (

"fmt"

*// importing Colly*

"github.com/gocolly/colly"

)

func main() {

*// instantiate a new collector object*

c := colly.NewCollector(

colly.AllowedDomains("www.scrapingcourse.com"),

)

}

First, the core of Colly's functionality is the [Collector](https://pkg.go.dev/github.com/gocolly/colly#Collector). The **Collector handles Colly's scraping instance, which provides event-driven callbacks** for sending HTTP requests, handling responses, and parsing HTML.

**Colly's callbacks are function fragments that dictate what happens at each level of the scraping process**. For instance, the OnHTML callback triggers the scraping event when a CSS selector matches a target element in the parsed HTML.

You can also configure the Collector object with proxies and custom headers like the User Agent, allowing you to mimic a human while scraping.

You can attach different types of callback functions to a Collector, as shown:

scraper.go

func main() {

*// ...*

*// called before an HTTP request is triggered*

c.OnRequest(func(r \*colly.Request) {

fmt.Println("Visiting: ", r.URL)

})

*// triggered when the scraper encounters an error*

c.OnError(func(\_ \*colly.Response, err error) {

fmt.Println("Something went wrong: ", err)

})

*// fired when the server responds*

c.OnResponse(func(r \*colly.Response) {

fmt.Println("Page visited: ", r.Request.URL)

})

*// triggered when a CSS selector matches an element*

c.OnHTML("a", func(e \*colly.HTMLElement) {

*// printing all URLs associated with the <a> tag on the page*

fmt.Println("%v", e.Attr("href"))

})

*// triggered once scraping is done (e.g., write the data to a CSV file)*

c.OnScraped(func(r \*colly.Response) {

fmt.Println(r.Request.URL, " scraped!")

})

}

These functions are executed in the following order:

1. OnRequest(): Called before performing an HTTP request with Visit().
2. OnError(): Called if an error occurred during the HTTP request.
3. OnResponse(): Called after receiving a response from the server.
4. OnHTML(): Called right after OnResponse() if the received content is HTML.
5. OnScraped(): Called after all OnHTML() callback executions are completed.

These callback functions help you build a comprehensive event-driven Golang web scraper using Colly. Each callback function responds to a specific event, allowing you to define the scraper's actions at different stages of the scraping process.

**Step 2: Visit the Target HTML Page**

Perform an initial HTTP GET request to download the target HTML page using Colly's [Visit()](https://pkg.go.dev/github.com/gocolly/colly#Collector.Visit) function. This function fires the onRequest event to start Colly's lifecycle.

scraper.go

func main() {

*// ...*

*// open the target URL*

c.Visit("https://www.scrapingcourse.com/ecommerce")

}

Now, let's extract some data!

**Step 3: Extract Relevant Data**

In this data scraping Go tutorial, you'll retrieve all product data from the target page.

But first, you need a data structure to store the scraped data. So, define a Product [Struct](https://pkg.go.dev/github.com/fatih/structs) before the main function as follows:

scraper.go

*// initialize a data structure to keep the scraped data*

type Product struct {

Url, Image, Name, Price string

}

If you're unfamiliar with the Go Struct, it's a collection of strictly typed fields you can use to collect data.

Initialize a Product [slice](https://go.dev/blog/slices-intro) that will contain the scraped data. In Go, **slices provide an efficient way to work with sequences of typed data**. You can think of them as lists:

scraper.go

func main() {

*// initialize the slice of structs that will contain the scraped data*

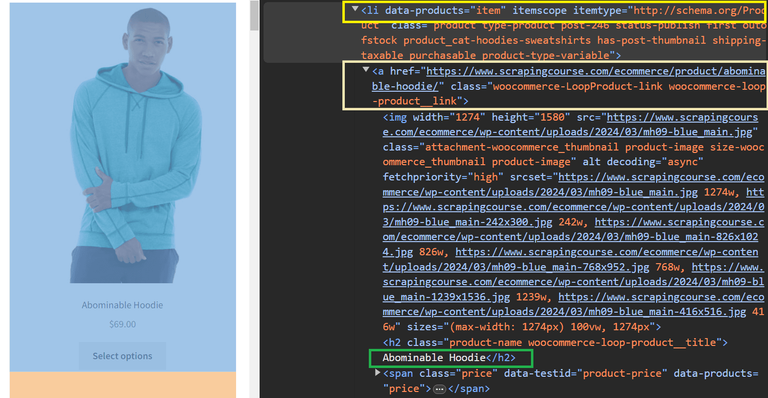
var products []Product

}

Let's now see how to extract data from an HTML element with the functions exposed by Colly.

Inspect the HTML of the product elements and grab their CSS selectors. Open the website on a browser, e.g., Chrome. Then, right-click on the first product element on the page and choose the "Inspect" option to access the DevTools section.

As seen in the image below, each product is inside a list tag (<li>):

[[](https://static.zenrows.com/content/large_scrapingcourse_e_commerce_first_element_inspection_68f4b7f674.png)Click to open the image in full screen](https://static.zenrows.com/content/large_scrapingcourse_e_commerce_first_element_inspection_68f4b7f674.png)

Here, note that the target li element has the .product class and contains the following elements:

* An a element with the product URL.
* An img element with the product image.
* The product name element with a .product-name class.
* A .price class name defining the product price element.

Although you've inspected only one product, you'll scrape all the product information on that page by parsing each element using its class name.

Select all li.product HTML product elements on the page using Colly's OnHTML callback. The OnHTML() function works as a [CSS selector](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Selectors) and a callback function. As mentioned, Colly will execute the callback when the target CSS selector matches an HTML element on the page. The e parameter in the callback represents a single li.product [HTMLElement](https://pkg.go.dev/github.com/gocolly/colly" \l "HTMLElement" \t "_blank):

scraper.go

func main() {

*// ...*

*// OnHTML callback*

c.OnHTML(".product", func(e \*colly.HTMLElement) {

*// ... scraping logic*

})

*// ... visit the target page*

}

Now, implement the scraping logic inside the OnHTML function. This function extracts all the list tags and iteratively scrapes product information from them.

The HTMLElement interface exposes the [ChildAttr()](https://pkg.go.dev/github.com/gocolly/colly" \l "HTMLElement.ChildAttr" \t "_blank) and [ChildText()](https://pkg.go.dev/github.com/gocolly/colly" \l "HTMLElement.ChildText" \t "_blank) methods, allowing you to extract the text and attribute values from the child elements. Finally, append a new element to the slice of scraped elements with append():

scraper.go

func main() {

*// ...*

*// OnHTML callback*

c.OnHTML("li.product", func(e \*colly.HTMLElement) {

*// initialize a new Product instance*

product := Product{}

*// scrape the target data*

product.Url = e.ChildAttr("a", "href")

product.Image = e.ChildAttr("img", "src")

product.Name = e.ChildText(".product-name")

product.Price = e.ChildText(".price")

*// add the product instance with scraped data to the list of products*

products = append(products, product)

})

*// ... visit the target page*

}

The next step is to export your scraped data to a CSV file.

**Step 4: Convert Scraped Data to CSV**

Complete the scraping steps by storing your collected data in a CSV file. Since you want to write the data to a CSV after all the scraping steps, implement the write event inside the OnScraped callback function.

Inside the OnScraped callback, create a products.csv file and initialize it with the header columns. Then, iterate over the slice of scraped Products, convert each of them to a new CSV record, and append it to the CSV file:

scraper.go

import (

*// ...*

"encoding/csv"

"os"

)

func main() {

*// ...*

c.OnScraped(func(r \*colly.Response) {

*// open the CSV file*

file, err := os.Create("products.csv")

if err != nil {

log.Fatalln("Failed to create output CSV file", err)

}

defer file.Close()

*// initialize a file writer*

writer := csv.NewWriter(file)

*// write the CSV headers*

headers := []string{

"Url",

"Image",

"Name",

"Price",

}

writer.Write(headers)

*// write each product as a CSV row*

for \_, product := range products {

*// convert a Product to an array of strings*

record := []string{

product.Url,

product.Image,

product.Name,

product.Price,

}

*// add a CSV record to the output file*

writer.Write(record)

}

defer writer.Flush()

})

*// ... visit the target page*

}

Ensure you add the following packages to your imports:

scraper.go

import (

"encoding/csv"

"log"

"os"

*//...*

)

Combine the snippets, and this is what your complete scraper code looks like:

scraper.go

package main

import (

"encoding/csv"

"log"

"os"

"github.com/gocolly/colly"

)

*// initialize a data structure to keep the scraped data*

type Product struct {

Url, Image, Name, Price string

}

func main() {

*// instantiate a new collector object*

c := colly.NewCollector(

colly.AllowedDomains("www.scrapingcourse.com"),

)

*// initialize the slice of structs that will contain the scraped data*

var products []Product

*// OnHTML callback*

c.OnHTML("li.product", func(e \*colly.HTMLElement) {

*// initialize a new Product instance*

product := Product{}

*// scrape the target data*

product.Url = e.ChildAttr("a", "href")

product.Image = e.ChildAttr("img", "src")

product.Name = e.ChildText(".product-name")

product.Price = e.ChildText(".price")

*// add the product instance with scraped data to the list of products*

products = append(products, product)

})

*// store the data to a CSV after extraction*

c.OnScraped(func(r \*colly.Response) {

*// open the CSV file*

file, err := os.Create("products.csv")

if err != nil {

log.Fatalln("Failed to create output CSV file", err)

}

defer file.Close()

*// initialize a file writer*

writer := csv.NewWriter(file)

*// write the CSV headers*

headers := []string{

"Url",

"Image",

"Name",

"Price",

}

writer.Write(headers)

*// write each product as a CSV row*

for \_, product := range products {

*// convert a Product to an array of strings*

record := []string{

product.Url,

product.Image,

product.Name,

product.Price,

}

*// add a CSV record to the output file*

writer.Write(record)

}

defer writer.Flush()

})

*// open the target URL*

c.Visit("https://www.scrapingcourse.com/ecommerce")

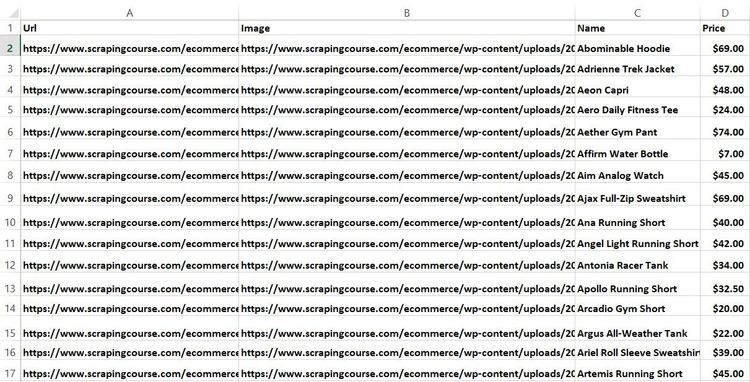
}

Run your Go scraper with the following command:

Terminal

go run scraper.go

You'll find a products.csv file in your project root directory. Open it, and it should contain the following scraped data:

[[](https://static.zenrows.com/content/large_scrapingcourse_ecommerce_product_output_csv_ed8501f6fc.jpg)Click to open the image in full screen](https://static.zenrows.com/content/large_scrapingcourse_ecommerce_product_output_csv_ed8501f6fc.jpg)

That's it! You just learned how to scrape a simple web page in Golang using Colly.

However, you must also know Colly's advanced concepts to scrape the entire website, extract content from dynamic web pages, and handle parallel scraping. We'll cover these in the next section.

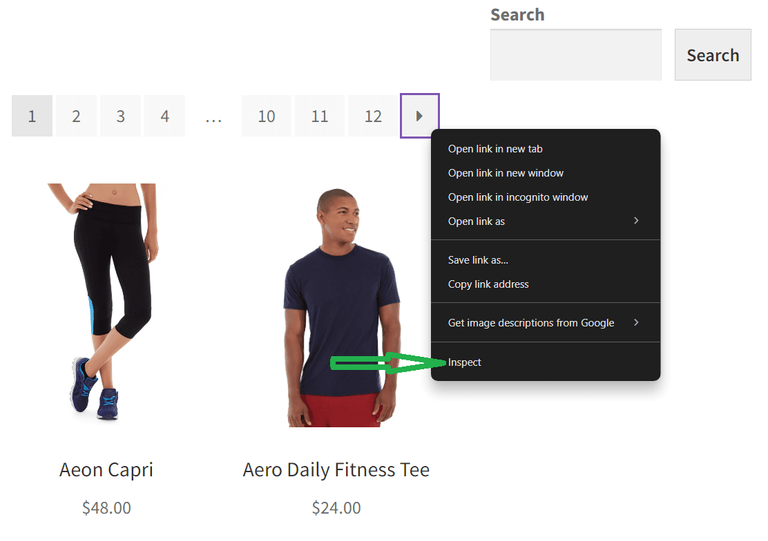
**Advanced Techniques in Web Scraping With Golang**

Now that you know the basics of web scraping with Colly in Go, it's time to dig into more advanced techniques.

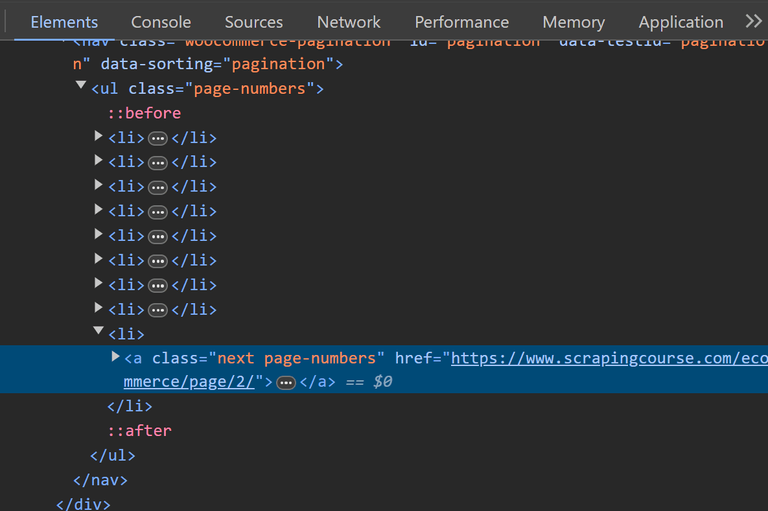
**Web Crawling With Go**

The target website distributes the products into many pages using pagination. You'll need to visit all its pages to extract all the product data on that website.

To perform web crawling in Go and scrape the entire website, you first need all the pagination links. So, right-click the next page arrow on the navigation bar and click the "Inspect" option:

[[](https://static.zenrows.com/content/large_scrapingcourse_next_page_link_inspect_option_e20773e47e.png)Click to open the image in full screen](https://static.zenrows.com/content/large_scrapingcourse_next_page_link_inspect_option_e20773e47e.png)

Your browser will give access to the DevTools section below with the selected HTML element highlighted:

[[](https://static.zenrows.com/content/large_scrapingcourse_navigation_inspection_c249c7aece.png)Click to open the image in full screen](https://static.zenrows.com/content/large_scrapingcourse_navigation_inspection_c249c7aece.png)

Look closely at the next link element; it has the class name .next and a next page URL. Clicking that next arrow increases the page number in the URL. You'll also notice it's no longer in the DOM once you get to the last page.

Modify your scraping logic to follow the URL in the next page element until it reaches the last page and has no more links to crawl.

Declare a global visitedUrls variable to use Go's standard sync package. This library helps keep track of visited links, preventing your script from visiting the same link twice:

scraper.go

func main() {

*//...*

*// define a sync to filter visited URLs*

var visitedUrls sync.Map

*//... scraping logic*

}

Add an extra OnHTML callback function to handle the pagination task. This function gets the next page URL from the next button element. It syncs the visited URLs into the visitedUrls variable for tracking, ensuring the scraper only visits new URLs. The **navigation task terminates once the scraper reaches the last page**, where the next button element disappears from the DOM:

scraper.go

func main{

*// ...*

*// OnHTML callback for handling pagination*

c.OnHTML("a.next", func(e \*colly.HTMLElement) {

*// extract the next page URL from the next button*

nextPage := e.Attr("href")

*// check if the nextPage URL has been visited*

if \_, found := visitedUrls.Load(nextPage); !found {

fmt.Println("scraping:", nextPage)

*// mark the URL as visited*

visitedUrls.Store(nextPage, struct{}{})

*// visit the next page*

e.Request.Visit(nextPage)

}

})

*//... store the data to a CSV after extraction*

*//... open the target URL*

}

Ensure you add Go's standard sync package to your imports:

scraper.go

import (

*// ...*

"sync"

)

Update the previous scraper, and your final code looks like this:

scraper.go

package main

import (

"encoding/csv"

"fmt"

"log"

"os"

"sync"

"github.com/gocolly/colly"

)

*// product structure to keep the scraped data*

type Product struct {

Url, Image, Name, Price string

}

func main() {

*// instantiate a new collector object*

c := colly.NewCollector(

colly.AllowedDomains("www.scrapingcourse.com"),

)

*// initialize the slice of structs that will contain the scraped data*

var products []Product

*// define a sync to filter visited URLs*

var visitedUrls sync.Map

*// OnHTML callback for scraping product information*

c.OnHTML("li.product", func(e \*colly.HTMLElement) {

*// initialize a new Product instance*

product := Product{}

*// scrape the target data*

product.Url = e.ChildAttr("a", "href")

product.Image = e.ChildAttr("img", "src")

product.Name = e.ChildText(".product-name")

product.Price = e.ChildText(".price")

*// add the product instance with scraped data to the list of products*

products = append(products, product)

})

*// OnHTML callback for handling pagination*

c.OnHTML("a.next", func(e \*colly.HTMLElement) {

*// extract the next page URL from the next button*

nextPage := e.Attr("href")

*// check if the nextPage URL has been visited*

if \_, found := visitedUrls.Load(nextPage); !found {

fmt.Println("scraping:", nextPage)

*// mark the URL as visited*

visitedUrls.Store(nextPage, struct{}{})

*// visit the next page*

e.Request.Visit(nextPage)

}

})

*// store the data to a CSV after extraction*

c.OnScraped(func(r \*colly.Response) {

*// open the CSV file*

file, err := os.Create("products.csv")

if err != nil {

log.Fatalln("Failed to create output CSV file", err)

}

defer file.Close()

*// initialize a file writer*

writer := csv.NewWriter(file)

*// write the CSV headers*

headers := []string{

"Url",

"Image",

"Name",

"Price",

}

writer.Write(headers)

*// write each product as a CSV row*

for \_, product := range products {

*// convert a Product to an array of strings*

record := []string{

product.Url,

product.Image,

product.Name,

product.Price,

}

*// add a CSV record to the output file*

writer.Write(record)

}

writer.Flush()

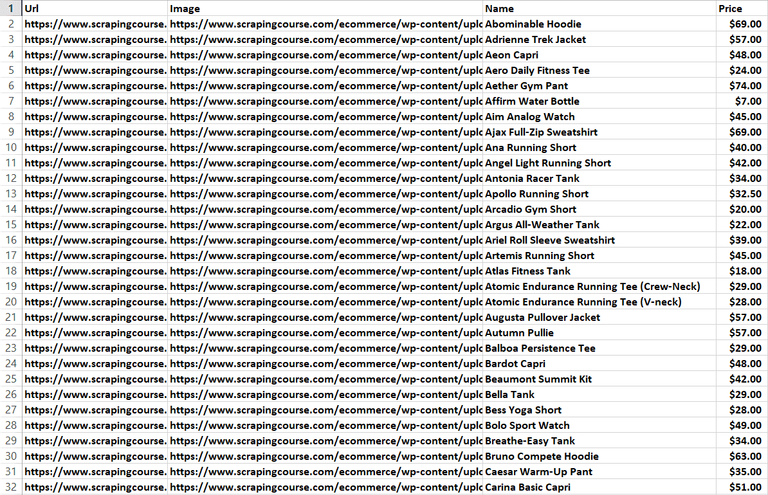
})

*// open the target URL*

c.Visit("https://www.scrapingcourse.com/ecommerce")

}

The new products.csv file now has more products:

[[](https://static.zenrows.com/content/large_scrapingcourse_full_extracted_data_csv_ff2dfa9efa.png)Click to open the image in full screen](https://static.zenrows.com/content/large_scrapingcourse_full_extracted_data_csv_ff2dfa9efa.png)

Well done! Now, you can crawl any paginated website with Colly in Golang.

**Avoid Getting Blocked**

Many websites you'll want to scrape use anti-bot measures to prevent you from extracting their content. The most basic anti-bot approach involves banning HTTP requests based on their headers, specifically those with an invalid [User Agent](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/User-Agent) header.

For example, Colly sends the following User Agent by default:

Example

"user-agent": "colly - https://github.com/gocolly/colly"

The above User Agent tells the server you're a bot, resulting in potential blocking. However, there are ways to deal with anti-bot detection.

**One technique is to spoof a real browser's User Agent request header to mimic a human user**.

Set a global User Agent header for all the requests performed by Colly with the [UserAgent](https://pkg.go.dev/github.com/gocolly/colly" \l "Collector.UserAgent" \t "_blank) Collector field, as shown below:

scraper.go

func main() {

*//...*

*// set a global User Agent*

c.UserAgent = "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/126.0.0.0 Safari/537.36"

*//... your scraping logic*

}

Another way to bypass anti-bots while scraping with Colly in Go is to use proxies to avoid IP bans due to rate limiting or geo-restrictions. Proxies route your request through another location, allowing you to appear as a different user.

There are two proxy types based on cost: free and premium. However, free proxies are short-lived and unreliable, so they're only suitable for testing, not real-life projects.

The best option is to use [premium web scraping proxies](https://www.zenrows.com/blog/web-scraping-proxy). Most premium proxies also auto-rotate residential IP addresses belonging to regular users, reducing your chances of getting detected as a bot.

Here's how to add a free proxy to your Golang Colly scraper from the [Free Proxy List](https://free-proxy-list.net/):

scraper.go

func main() {

*// ...*

*// set up the proxy*

err := c.SetProxy("http://35.185.196.38:3128")

if err != nil {

log.Fatal(err)

}

*//... your scraping logic*

}

**Note**

The above proxy is free and may not work at the time of reading. Feel free to grab a new one from the Free Proxy List website.

If using a premium proxy instead, include your password and username as follows:

scraper.go

func main() {

*// ...*

*// set up the proxy*

err := c.SetProxy("http://<YOUR\_USERNAME>:<YOUR\_PASSWORD>@<PROXY\_IP\_ADDRESS>:<PROXY\_PORT>

")

if err != nil {

log.Fatal(err)

}

*//... your scraping logic*

}

After adding the User Agent header and the proxy address, the previous final crawling code with Colly becomes:

scraper.go

package main

import (

"encoding/csv"

"fmt"

"log"

"os"

"sync"

"github.com/gocolly/colly"

)

*// product structure to keep the scraped data*

type Product struct {

Url, Image, Name, Price string

}

func main() {

*// instantiate a new collector object*

c := colly.NewCollector(

colly.AllowedDomains("www.scrapingcourse.com"),

)

*// initialize the slice of structs that will contain the scraped data*

var products []Product

*// define a sync to filter visited URLs*

var visitedUrls sync.Map

*// set a global User Agent*

c.UserAgent = "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/126.0.0.0 Safari/537.36"

*// set up the proxy*

err := c.SetProxy("http://35.185.196.38:3128")

if err != nil {

log.Fatal(err)

}

*// OnHTML callback for scraping product information*

c.OnHTML("li.product", func(e \*colly.HTMLElement) {

*// initialize a new Product instance*

product := Product{}

*// scrape the target data*

product.Url = e.ChildAttr("a", "href")

product.Image = e.ChildAttr("img", "src")

product.Name = e.ChildText(".product-name")

product.Price = e.ChildText(".price")

*// add the product instance with scraped data to the list of products*

products = append(products, product)

})

*// OnHTML callback for handling pagination*

c.OnHTML("a.next", func(e \*colly.HTMLElement) {

*// extract the next page URL from the next button*

nextPage := e.Attr("href")

*// check if the nextPage URL has been visited*

if \_, found := visitedUrls.Load(nextPage); !found {

fmt.Println("scraping:", nextPage)

*// mark the URL as visited*

visitedUrls.Store(nextPage, struct{}{})

*// visit the next page*

e.Request.Visit(nextPage)

}

})

*// store the data to a CSV after extraction*

c.OnScraped(func(r \*colly.Response) {

*// open the CSV file*

file, err := os.Create("products.csv")

if err != nil {

log.Fatalln("Failed to create output CSV file", err)

}

defer file.Close()

*// initialize a file writer*

writer := csv.NewWriter(file)

*// write the CSV headers*

headers := []string{

"Url",

"Image",

"Name",

"Price",

}

writer.Write(headers)

*// write each product as a CSV row*

for \_, product := range products {

*// convert a Product to an array of strings*

record := []string{

product.Url,

product.Image,

product.Name,

product.Price,

}

*// add a CSV record to the output file*

writer.Write(record)

}

writer.Flush()

})

*// open the target URL*

c.Visit("https://www.scrapingcourse.com/ecommerce")

}

However, these approaches are often insufficient against sophisticated anti-bot measures like Cloudflare, Akamai, DataDome, etc.

Let's try to use Colly to access a heavily protected website like the [G2 Reviews](https://www.g2.com/products/asana/reviews) with the User Agent and proxy address still intact:

Example

package main

import (

"fmt"

"log"

"github.com/gocolly/colly"

)

func main() {

*// instantiate a new collector object*

c := colly.NewCollector(

colly.AllowedDomains("www.g2.com"),

)

*// set a global User Agent*

c.UserAgent = "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/126.0.0.0 Safari/537.36"

*// set up the proxy*

err := c.SetProxy("http://35.185.196.38:3128")

if err != nil {

log.Fatal(err)

}

*// OnError callback*

c.OnError(func(\_ \*colly.Response, err error) {

log.Println("Something went wrong:", err)

})

*// OnResponse callback to print the full HTML*

c.OnResponse(func(r \*colly.Response) {

fmt.Println("Page visited:", r.Request.URL.String())

fmt.Println("Full HTML:\n", string(r.Body))

})

*// OnScraped callback*

c.OnScraped(func(r \*colly.Response) {

fmt.Println("Finished scraping:", r.Request.URL.String())

})

*// open the target URL*

c.Visit("https://www.g2.com/products/asana/reviews")

}

Despite setting a User Agent header and a proxy, Colly triggers the OnError callback, showing that it got blocked by a [403 forbidden error](https://www.zenrows.com/blog/403-web-scraping):

Output

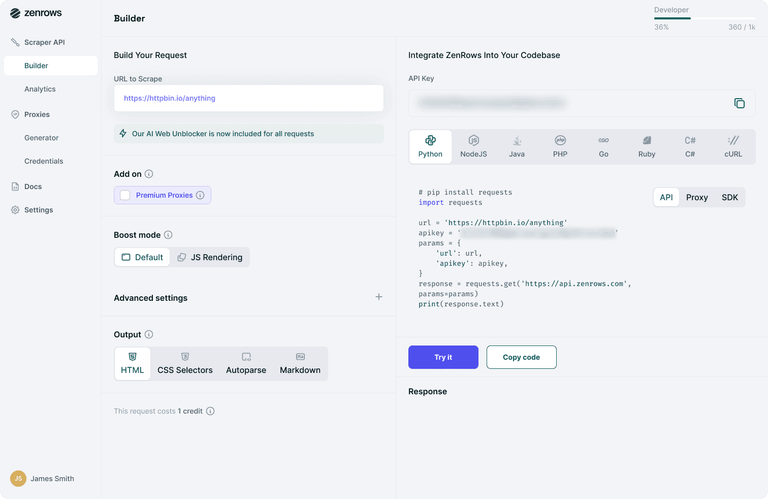
Something went wrong: Forbidden

The best way to bypass any anti-bot system and scrape without limitations is to use a web scraping API like [ZenRows](https://www.zenrows.com/" \t "_blank).

ZenRows automatically auto-rotates premium proxies, manages your request headers, and auto-bypasses CAPTCHAs and other anti-bot measures, removing the stress of manual configurations. It also acts as a headless browser for mimicking human behavior and scraping content from dynamic websites.

Let's show you how ZenRows works by scraping the previous G2 page that got you blocked.

[Sign up](https://www.zenrows.com/) to open the ZenRows Request Builder. Paste the target URL in the link box, activate Premium Proxies, and select JS Rendering. Select the API connection mode and choose Go as your preferred language. Then, copy and paste the generated code into your Go file:

[[](https://static.zenrows.com/content/large_zenrows_request_builder_only_4ddcf188b7.png)Click to open the image in full screen](https://static.zenrows.com/content/large_zenrows_request_builder_only_4ddcf188b7.png)

The generated code should look like this:

Example

package main

import (

"io"

"log"

"net/http"

)

func main() {

client := &http.Client{}

req, err := http.NewRequest("GET", "https://api.zenrows.com/v1/?apikey=<YOUR\_ZENROWS\_API\_KEY>&url=https%3A%2F%2Fwww.g2.com%2Fproducts%2Fasana%2Freviews&js\_render=true&premium\_proxy=true", nil)

resp, err := client.Do(req)

if err != nil {

log.Fatalln(err)

}

defer resp.Body.Close()

body, err := io.ReadAll(resp.Body)

if err != nil {

log.Fatalln(err)

}

log.Println(string(body))

}

The code prints the protected website's full-page HTML:

Output

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<link href="https://www.g2.com/images/favicon.ico" rel="shortcut icon" type="image/x-icon" />

<title>Asana Reviews, Pros + Cons, and Top Rated Features</title>

</head>

<body>

*<!-- other content omitted for brevity -->*

</body>

</html>

Fantastic! You've just used ZenRows to bypass an advanced anti-bot measure with less than 25 code lines.

Let's explore more advanced features, including parallel scraping and dynamic content extraction.

**Parallel Web Scraping in Golang**

Data scraping in Go can take a lot of time. The reason could be a slow internet connection, an overloaded web server or simply a large number of pages to scrape.

That's why [Colly supports parallel scraping](http://go-colly.org/docs/examples/parallel/). **Parallel web scraping in Go involves extracting data from multiple pages simultaneously**.

The target [demo e-commerce website](https://www.scrapingcourse.com/ecommerce/) has 12 pages and formats its navigation URL like so:

Example

https://www.scrapingcourse.com/ecommerce/page/<PAGE\_NUMBER>/

For example, the page 3 URL looks like this in a browser's link box:

Example

https://www.scrapingcourse.com/ecommerce/page/3/

This is the list of all pagination pages you want your crawler to visit:

scraper.go

func main{

pagesToScrape := []string{

"https://www.scrapingcourse.com/ecommerce/page/1/",

"https://www.scrapingcourse.com/ecommerce/page/2/",

*// ... omitted for brevity*

"https://www.scrapingcourse.com/ecommerce/page/11/",

"https://www.scrapingcourse.com/ecommerce/page/12/",

}

}

With parallel scraping, your Go spider will be able to visit and extract data from several web pages at the same time. That will make your scraping process way faster!

Colly has an [async mode](https://pkg.go.dev/github.com/gocolly/colly/v2#Collector.Async), which allows it to visit several pages simultaneously when enabled. Specifically, Colly will visit as many pages concurrently as the value of the [Parallelism](https://pkg.go.dev/github.com/gocolly/colly/v2#LimitRule.Parallelism) parameter allows.

Use Colly to implement a parallel web spider:

scraper.go

func main{

*// ...*

c := colly.NewCollector(

*// ...*

*// turn on the asynchronous request mode in Colly*

colly.Async(true),

)

c.Limit(&colly.LimitRule{

*// limit the parallel requests to 4 request at a time*

Parallelism: 4,

})

c.OnHTML("li.product", func(e \*colly.HTMLElement) {

*// scraping logic...*

})

*// register all pages to scrape*

for \_, pageToScrape := range pagesToScrape {

c.Visit(pageToScrape)

}

*//... scraping logic*

*// wait for Colly to visit all pages*

c.Wait()

*//... export logic*

}

Here's the updated final code:

scraper.go

package main

import (

"encoding/csv"

"log"

"os"

"github.com/gocolly/colly"

)

*// product structure to keep the scraped data*

type Product struct {

Url, Image, Name, Price string

}

func main() {

pagesToScrape := []string{

"https://www.scrapingcourse.com/ecommerce/page/1/",

"https://www.scrapingcourse.com/ecommerce/page/2/",

"https://www.scrapingcourse.com/ecommerce/page/3/",

"https://www.scrapingcourse.com/ecommerce/page/4/",

"https://www.scrapingcourse.com/ecommerce/page/5/",

"https://www.scrapingcourse.com/ecommerce/page/6/",

"https://www.scrapingcourse.com/ecommerce/page/7/",

"https://www.scrapingcourse.com/ecommerce/page/8/",

"https://www.scrapingcourse.com/ecommerce/page/9/",

"https://www.scrapingcourse.com/ecommerce/page/10/",

"https://www.scrapingcourse.com/ecommerce/page/11/",

"https://www.scrapingcourse.com/ecommerce/page/12/",

}

*// instantiate a new collector object*

c := colly.NewCollector(

colly.AllowedDomains("www.scrapingcourse.com"),

colly.Async(true),

)

c.Limit(&colly.LimitRule{

*// limit the parallel requests to 4 request at a time*

Parallelism: 4,

})

*// initialize the slice of structs that will contain the scraped data*

var products []Product

*// set a global User Agent*

c.UserAgent = "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/126.0.0.0 Safari/537.36"

*// set up the proxy*

err := c.SetProxy("http://35.185.196.38:3128")

if err != nil {

log.Fatal(err)

}

*// OnHTML callback for scraping product information*

c.OnHTML("li.product", func(e \*colly.HTMLElement) {

*// initialize a new Product instance*

product := Product{}

*// scrape the target data*

product.Url = e.ChildAttr("a", "href")

product.Image = e.ChildAttr("img", "src")

product.Name = e.ChildText(".product-name")

product.Price = e.ChildText(".price")

*// add the product instance with scraped data to the list of products*

products = append(products, product)

})

*// register all pages to scrape*

for \_, pageToScrape := range pagesToScrape {

c.Visit(pageToScrape)

*// store the data to a CSV after extraction*

c.OnScraped(func(r \*colly.Response) {

*// open the CSV file*

file, err := os.Create("products.csv")

if err != nil {

log.Fatalln("Failed to create output CSV file", err)

}

defer file.Close()

*// initialize a file writer*

writer := csv.NewWriter(file)

*// write the CSV headers*

headers := []string{

"Url",

"Image",

"Name",

"Price",

}

writer.Write(headers)

*// write each product as a CSV row*

for \_, product := range products {

*// convert a Product to an array of strings*

record := []string{

product.Url,

product.Image,

product.Name,

product.Price,

}

*// add a CSV record to the output file*

writer.Write(record)

}

writer.Flush()

})

}

*// wait for Colly to visit all pages*

c.Wait()

}

You'll get the same products.csv file with all the product details as the output.

Enabling the parallel mode lets you achieve better scraping performance. However, depending on your scraping requirements, you may still need to change some code logic to prevent[race conditions](https://en.wikipedia.org/wiki/Race_condition). That's because most data structures in Go aren't thread-safe.

Great! You just learned the basics of parallel web scraping.

**Scraping Dynamic Content Websites With a Headless Browser in Go**

**A static-content website comes with all its content pre-loaded in the HTML pages returned by the server**. It means you can scrape data from a static-content website by simply parsing its HTML content.

On the other hand, **other websites rely on JavaScript for page rendering** or use it to perform API calls and retrieve data asynchronously. These websites are called **dynamic content websites**.

To render their content, you'll need a tool that can run JavaScript, such as a [headless browser](https://www.zenrows.com/blog/headless-browser-scraping). A headless browser provides browser capabilities and allows you to load a web page in a browser with no graphic user interface (GUI). You can instruct the headless browser to mimic user interactions, such as clicking, hovering, logging in, scrolling, and more.

The most popular headless browser library for Golang is [chromedp](https://www.zenrows.com/blog/chromedp" \t "_blank). Install it with the following command:

Terminal

go get -u github.com/chromedp/chromedp

Let's use chromedp to scrape data from the demo e-commerce website in a browser.

The chromedp [Nodes()](https://pkg.go.dev/github.com/chromedp/chromedp#Nodes) function in the code below enables you to instruct the headless browser to perform a query. Select the product HTML elements and store them in the nodes variable. Then, iterate over them and apply the [AttributeValue()](https://pkg.go.dev/github.com/chromedp/chromedp" \l "AttributeValue" \t "_blank) and [Text()](https://pkg.go.dev/github.com/chromedp/chromedp#Text) methods to get the target data.

Example

package main

import (

"context"

"fmt"

"log"

"github.com/chromedp/cdproto/cdp"

"github.com/chromedp/chromedp"

)

type Product struct {

Url, Image, Name, Price string

}

func main() {

var products []Product

*// initialize a chrome instance*

ctx, cancel := chromedp.NewContext(

context.Background(),

chromedp.WithLogf(log.Printf),

)

defer cancel()

*// navigate to the target web page and select the HTML elements of interest*

var nodes []\*cdp.Node

chromedp.Run(ctx,

chromedp.Navigate("https://www.scrapingcourse.com/ecommerce"),

chromedp.Nodes(".product", &nodes, chromedp.ByQueryAll),

)

*// scraping data from each node*

var url, image, name, price string

for \_, node := range nodes {

chromedp.Run(ctx,

chromedp.AttributeValue("a", "href", &url, nil, chromedp.ByQuery, chromedp.FromNode(node)),

chromedp.AttributeValue("img", "src", &image, nil, chromedp.ByQuery, chromedp.FromNode(node)),

chromedp.Text(".product-name", &name, chromedp.ByQuery, chromedp.FromNode(node)),

chromedp.Text(".price", &price, chromedp.ByQuery, chromedp.FromNode(node)),

)

product := Product{}

product.Url = url

product.Image = image

product.Name = name

product.Price = price

products = append(products, product)

}

fmt.Print(products)

*//... export logic*

}

The above code outputs the desired data:

Output

[

{

https://www.scrapingcourse.com/ecommerce/product/abominable-hoodie/,

https://www.scrapingcourse.com/ecommerce/wp-content/uploads/2024/03/mh09-blue\_main.jpg,

Abominable Hoodie,

$69.00

},

//... other products omitted for brevity

{

https://www.scrapingcourse.com/ecommerce/product/artemis-running-short/,

https://www.scrapingcourse.com/ecommerce/wp-content/uploads/2024/03/wsh04-black\_main.jpg,

Artemis Running Short,

$45.00

}

]

Performing web scraping in Go with Colly or chomedp is similar. The only difference is that chromedp runs the scraping instructions in a browser and allows you to scrape dynamic content, while Colly doesn't support browser functionalities for extracting dynamic content.

With chromedp, you can crawl dynamic websites and interact with them in a browser as a real user. It also means that your script is less likely to be detected as a bot, so chromedp makes it easy to [scrape a web page without getting blocked](https://www.zenrows.com/blog/web-scraping-without-getting-blocked).

**Other Web Scraping Libraries for Go**

Other great libraries for web scraping with Golang are:

* [**ZenRows**](https://www.zenrows.com/): A complete web scraping API that handles all anti-bot bypass for you. It comes with headless browser capabilities, CAPTCHA bypass, rotating proxies and more.
* [**GoQuery**](https://www.zenrows.com/blog/goquery): A Go library that offers a syntax and a set of features similar to [jQuery](http://jquery.com/). You can use it to perform web scraping just like in JQuery.
* [**Ferret**](https://github.com/MontFerret/ferret): A portable, extensible and fast web scraping system that aims to simplify data extraction from the web. Ferret allows users to focus on the data and is based on a unique declarative language.
* [**Selenium**](https://www.zenrows.com/blog/selenium-golang): One of the most well-known headless browsers suitable for scraping dynamic content. It doesn't offer official support, but there's a port for it in Go.

**Conclusion**

In this step-by-step Go tutorial, you've seen the building blocks for Golang web scraping.

Here's what you've learned:

* Perform basic data scraping with Golang using Colly.
* Achieve crawling logic to visit a whole website.
* The reason why you may need a Go headless browser solution.
* Scrape a dynamic-content website with chromedp.

**Scraping can become challenging because of the anti-scraping measures** implemented by several websites. Many libraries struggle to bypass these obstacles. A best practice to avoid these problems is to use a web scraping API, such as [ZenRows](https://www.zenrows.com/" \t "_blank). This solution bypasses all anti-bot systems with a single API call.