Building Go Web Applications and Microservices Using Gin

使用 Gin 構建 Go Web 應用程序和微服務

Introduction

In this tutorial, you will learn how to build traditional web applications and microservices in Go using the Gin framework. Gin is a framework which reduces boilerplate code that would normally go into building these applications. It also lends itself very well to creating reusable and extensible pieces of code.

This part of the tutorial will help you set up your project and build a simple application using Gin that will display a list of articles and the article details page.

介紹

在本教程中，您將學習如何使用 Gin 框架在 Go 中構建傳統 Web 應用程序和微服務。Gin 是一種可以減少通常用於構建這些應用程序的樣板代碼的框架。它也非常適合創建可重用和可擴展的代碼片斷。

本教程的這一部分將幫助您設置您的項目並使用 Gin 構建一個簡單的應用程序，該應用程序將顯示文章列表和文章詳細信息頁面。

Goals

By the end of this tutorial, you will:

Learn how to use Gin to build a web application,

Understand the parts of a web application written in Go, and

Learn how to use Semaphore Continuous Integration to test and build the application quickly and safely.

目標

在本教程結束時，您將：

學習如何使用 Gin 建立一個 Web 應用程序，和

瞭解用 Go 編寫的 Web 應用程序的各個部分，及

了解如何使用 Semaphore Continuous Integration 工具來快速安全地測試和構建應用程序。

Prerequisites

For this tutorial, you will need Go, Git, and curl installed on your machine.

You can find the full source of the tutorial in this repository, feel free to fork it:

https://github.com/demo-apps/go-gin-app

Note: If you don’t have curl, you can use any other tool that you would normally use to test API endpoints.

先決條件

對於本教程，您將需要 Go 並 curl 安裝在您的機器上。

您可以在此存儲庫中找到本教程的完整源代碼，可以隨意進行 fork：

https://github.com/demo-apps/go-gin-app

注意：如果您沒有 curl，則可以使用通常用於測試 API 端點的任何其他工具。

What is Gin?

Gin is a high-performance micro-framework that can be used to build web applications and microservices. It makes it simple to build a request handling pipeline from modular, reusable pieces. It does this by allowing you to write middleware that can be plugged into one or more request handlers or groups of request handlers.

什麼是 Gin？

Gin 是一種高性能的微型框架，可用於構建 Web 應用程序和微服務。它使得從模塊化，可重用的部分構建請求處理管道變得非常簡單。它通過允許您編寫可插入一個或多個請求處理程序或請求處理程序組的中間件來實現此目的。

Why Gin?

One of the best features of Go is that it’s built-in net/http library that allows you to create an HTTP server effortlessly. However, it is also less flexible and requires some boilerplate code to implement.

There is no built-in support in Go to handle routes based on a regular expression or a “pattern”. You can write code to add this functionality. However, as the number of your applications grows, it is quite likely that you will either repeat such code everywhere or create a library for reuse.

This is the crux of what Gin offers. It contains a set of commonly used functionalities, e.g. routing, middleware support, rendering, that reduce boilerplate code and make writing web applications simpler.

為什麼要使用 Gin？

Go 的最佳功能之一是其內置的 net/http 庫，可以讓您輕鬆地創建 HTTP 服務器。但是，它也不太靈活，需要一些樣板代碼才能實現。

Go 中沒有內置的支持來處理基於正則表達式或“模式”的路由。您可以編寫代碼來添加此功能。但是，隨著應用程序數量的增長，您很可能會在任何地方重複使用這些代碼，或者創建一個用於重用的庫。

這是 Gin 提供的關鍵。它包含一組常用的功能，例如路由，中間件支持，渲染，可以減少樣板代碼並簡化 Web 應用程序的編寫。

Designing the Application

Let’s take a quick look at how a request is processed in Gin. The control flow for a typical web application, API server or a microservice looks as follows:

設計應用程序

讓我們快速看一下在 Gin 中處理請求的方式。典型的 Web 應用程序，API 服務器或微服務的控制流程如下所示：

|  |
| --- |
| Request -> Route Parser -> [Optional Middleware] -> Route Handler -> [Optional Middleware] -> Response |

When a request comes in, Gin first parses the route. If a matching route definition is found, Gin invokes the route handler and zero or more middleware in an order defined by the route definition. We will see how this is done when we take a look at the code in a later section.

當請求進入時，Gin 先分析路由。如果找到匹配的路由定義，則 Gin 將按路由定義的順序調用路由處理程序和零個或多個中間件。當我們在後面的章節中查看代碼時，我們會看到這是如何完成的。

Application Functionality

The application we’ll build is a simple article manager. This application should:

* Let users register with a username and a password (non-logged in users only),
* Let users login with a username and a password (non-logged in users only),
* Let users log out (logged in users only),
* Let users create new articles (logged in users only),
* Display the list of all articles on the home page (for all users), and
* Display a single article on its own page (for all users).

In addition to this functionality, the list of articles and a single article should be accessible in the HTML, JSON and XML formats.

This will allow us to illustrate how Gin can be used to design traditional web applications, API servers, and microservices.

To achieve this, we will make use of the following functionalities offered by Gin:

* Routing — to handle various URLs,
* Custom rendering — to handle the response format, and
* Middleware — to implement authentication.

We’ll also write tests to validate that all the features work as intended.

應用功能

我們將建立的應用程序是一個簡單的文章管理。此應用程序包括以下功能：

* 讓用戶使用用戶名和密碼進行註冊（僅限未登錄的用戶），
* 讓用戶使用用戶名和密碼登錄（僅限未登錄的用戶），
* 讓用戶註銷（僅限登錄用戶），
* 讓用戶創建新文章（僅限登錄用戶），
* 顯示主頁上所有文章的列表（針對所有用戶）和
* 在其自己的頁面上顯示一篇文章（適用於所有用戶）。

除此功能外，文章列表和單個文章應該可以以 HTML，JSON 和 XML 格式訪問。

這將使我們能夠說明 Gin 如何用於設計傳統的 Web 應用程序，API 服務器和微服務。

為了達到這個目的，我們將使用由 Gin 提供的以下功能：

* 路由 - 處理各種 URL，
* 自定義渲染 - 處理響應格式和
* 中間件 - 實現認證。

我們還會編寫測試來驗證所有功能是否按預期工作。

Routing

Routing is one of the core features that all modern frameworks provide. Any web page or an API endpoint is accessed by a URL. Frameworks use routes to handle requests to these URLs. If a URL is http://www.example.com/some/random/route, the route will be /some/random/route.

Gin offers a fast router that’s easy to configure and use. Apart from handling specified URLs, Gin routers can also handle patterns and grouped URLs.

In our application, we will:

* Serve the index page at route / (HTTP GET request),
* Group user-related routes under the /u route,
  + Serve the login page at /u/login (HTTP GET request),
  + Process the login credentials at /u/login (HTTP POST request),
  + Log out at /u/logout (HTTP GET request),
  + Serve the registration page at /u/register (HTTP GET request),
  + Process the registration information at /u/register (HTTP POST request) ,
* Group article related routes under the /article route,
  + Serve the article creation page at /article/create (HTTP GET request),
  + Process the submitted article at /article/create (HTTP POST request), and
  + Serve the article page at /article/view/:article\_id (HTTP GET request). Take note of the :article\_id part in this route. The : at the beginning indicates that this is a dynamic route. This means that :article\_id can contain any value and Gin will make this value available in the route handler.

路由

路由是所有現代框架提供的核心功能之一。任何網頁或 API 端點都可以通過 URL 訪問。框架使用路由來處理對這些 URL 的請求。如果 URL 是 http://www.example.com/some/random/route，路由將是 /some/random/route。

Gin 提供了一個易於配置和使用的快速路由器。除了處理指定的 URL 之外，Gin 路由器還可以處理模式和分組 URL。

在我們的應用程序中，我們將：

* 在路由服務索引頁面 /（HTTP GET 請求），
* 路由下的用戶相關路由分組 /u，
  + 在 /u/login（HTTP GET 請求）服務登錄頁面，
  + 在 /u/login（HTTP POST 請求）處理登錄憑證，
  + 在 /u/logout（HTTP GET 請求）註銷，
  + 在 /u/register（HTTP GET 請求）服務註冊頁面，
  + 在 /u/register（HTTP POST 請求）處理註冊信息，
* 在路線下分組文章相關 /article 路線，
  + 在 /article/create（HTTP GET 請求）服務文章創建頁面，
  + 處理提交的文章 /article/create（HTTP POST 請求）和
  + 在 /article/view/:article\_id（HTTP GET 請求）服務文章頁面。記下 :article\_id 該路線中的部分。在 :一開始表示這是一個動態的路線。這意味著 :article\_id 可以包含任何值，並且 Gin 將使這個值在路由處理器中可用。

Rendering

A web application can render a response in various formats like HTML, text, JSON, XML or other formats. API endpoints and microservices typically respond with data, commonly in JSON format but also in any other desired format.

In the next section, we’ll see how we can render different types of responses without duplicating any functionality. We will primarily respond to a request with an HTML template. However, we will also define two endpoints that can respond with JSON or XML data.

渲染

Web 應用程序可以呈現各種格式的響應，如 HTML，文本，JSON，XML 或其他格式。API 端點和微服務通常以數據進行響應，通常以 JSON 格式進行響應，但也以任何其他所需的格式進行響應。

在下一節中，我們將看到如何在不重複任何功能的情況下呈現不同類型的響應。我們將主要通過 HTML 模板回應請求。但是，我們還將定義可以使用 JSON 或 XML 數據響應的兩個端點。

Middleware

In the context of a Go web application, middleware is a piece of code that can be executed at any stage while handling an HTTP request. It is typically used to encapsulate common functionality that you want to apply to multiple routes. We can use middleware before and/or after an HTTP request is handled. Some common uses of middleware include authorization, validation, etc.

If middleware is used before a request is handled, any changes it makes to the request will be available in the main route handler. This is handy if we want to implement some validations on certain requests. On the other hand, if the middleware is used after the route handler, it will have a response from the route handler. This can be used to modify the response from the route handler.

Gin allows us to write middleware that implements some common functionality that needs to be shared while handling multiple routes. This keeps the codebase small, separates concerns and improves code maintainability.

We want to ensure that some pages and actions, eg. creating an article, logging out, are available only to users who are logged in. We also want to ensure that some pages and actions, eg. registering, logging in, are available only to users who aren’t logged in.

If we were to put this logic in every route, it would be quite tedious, repetitive and error-prone. Luckily, we can create middleware for each of these tasks and reuse them in specific routes.

We will also create middleware that will be applied to all routes. This middleware (setUserStatus) will check whether a request is from an authenticated user or not. It will then set a flag that can be used in templates to modify the visibility of some of the menu links based on this flag.

中間件

在 Go Web 應用程序的上下文中，中間件是可以在處理 HTTP 請求的任何階段執行的一段代碼。它通常用於封裝您想要應用於多個路由的通用功能。我們可以在處理 HTTP 請求之前和 /或之後使用中間件。中間件的一些常見用途包括授權，驗證等。

如果在處理請求之前使用了中間件，那麼它對請求所做的任何更改都將在主要路由處理程序中可用。如果我們想對某些請求實施一些驗證，這很方便。另一方面，如果在路由處理程序之後使用中間件，它將具有來自路由處理程序的響應。這可以用來修改路由處理程序的響應。

Gin 允許我們編寫實現一些常見功能的中間件，這些功能需要在處理多條路由時進行共享。這使代碼庫變小，分離問題並提高代碼可維護性。

我們希望確保一些頁面和操作，例如。創建一篇文章並註銷，只能用於已登錄的用戶。我們還希望確保某些頁面和操作，例如。註冊，登錄，僅適用於未登錄的用戶。

如果我們將這種邏輯放在每個路由上，那將是相當繁瑣，重複和容易出錯的。幸運的是，我們可以為這些任務中的每一個創建中間件，並在特定路由中重用它們。

我們還將創建將應用於所有路由的中間件。該中間件（setUserStatus）將檢查請求是否來自經過驗證的用戶。然後它將設置一個標誌，可以在模板中使用該標誌來修改基於該標誌的某些菜單鏈接的可見性。

Installing the Dependency

This application will only use one external dependency: the Gin framework. We can install the latest version using the following command:

go get -u github.com/gin-gonic/gin

安裝依賴關係

此應用程序將只使用一個外部依賴項 - Gin 框架。我們可以使用以下命令安裝最新版本：

go get -u github.com/gin-gonic/gin

Creating Reusable Templates

Our application will display a web page using its template. However, there will be several parts such as the header, menu, sidebar, and footer, which will be common across all pages. Go allows us to create reusable template snippets that can be imported in other templates.

The header and the footer will be the common pieces that will be reused across all templates. We will also create the menu in its own template file which will be used by the header template. Finally, we’ll create the template for the index page which will import the header and the footer. All the template files will be placed in the templates directory within the project directory.

Let’s start by creating the template for the menu in templates/menu.html as follows:

創建可重用模板

我們的應用程序將使用其模板顯示網頁。但是，會有幾個部分，例如標題，菜單，側邊欄和頁腳，這些在所有頁面中都是常見的。Go 允許我們創建可以在其他模板中導入的可重用模板片段。

頁眉和頁腳將是將在所有模板中重複使用的常見部分。我們還將在其自己的模板文件中創建菜單，該模板文件將由標題模板使用。最後，我們將為導入頁眉和頁腳的索引頁創建模板。所有模板文件將被放置在 templates 項目目錄中的目錄中。

我們首先為菜單創建模板，templates/menu.html 如下所示

|  |
| --- |
| <!--menu.html-->  <nav class="navbar navbar-default">  <div class="container">  <div class="navbar-header">  <a class="navbar-brand" href="/">  Home  </a>  </div>  </div>  </nav> |

Initially, the menu contains only the link to the home page. We will add to this as we add more functionality to the application. The template for the header will be placed in the templates/header.html file as follows:

最初，菜單僅包含主頁的鏈接。隨著我們嚮應用程序添加更多功能，我們將添加到此。標題的模板將被放置在 templates/header.html 文件中，如下所示：

|  |
| --- |
| <!--header.html-->  <!doctype html>  <html>  <head>  <!--Use the title variable to set the title of the page-->  <title>{{ .title }}</title>  <meta name="viewport" content="width=device-width, initial-scale=1">  <meta charset="UTF-8">  <!--Use bootstrap to make the application look nice-->  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css" integrity="sha384-1q8mTJOASx8j1Au+a5WDVnPi2lkFfwwEAa8hDDdjZlpLegxhjVME1fgjWPGmkzs7" crossorigin="anonymous">  <script async src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/js/bootstrap.min.js" integrity="sha384-0mSbJDEHialfmuBBQP6A4Qrprq5OVfW37PRR3j5ELqxss1yVqOtnepnHVP9aJ7xS" crossorigin="anonymous"></script>  </head>  <body class="container">  <!--Embed the menu.html template at this location-->  {{ template "menu.html" . }} |

And the footer:

|  |
| --- |
| <!--footer.html-->  </body>  </html> |

As you can see, we are using the open-source Bootstrap framework. Most of this file is standard HTML. However, take note of two lines. The line containing <title>{{ .title }}</title> is used to dynamically set the title of the page using the .title variable that must be set inside the application. Secondly, the line containing {{ template "menu.html" . }} is used to import the menu template from the menu.html file. This is how Go lets you import one template in another.

The template for the footer contains static HTML. The template for the index page makes use of the header and the footer and displays a simple Hello Gin message:

正如你所看到的，我們正在使用開源的 Bootstrap 框架。這個文件大部分是標準的 HTML。但是，請注意兩行。包含的行 <title>{{ .title }}</title> 用於使用 .title 必須在應用程序內設置的變量動態設置頁面的標題。其次，包含的行 {{ template "menu.html" . }} 用於從 menu.html 文件中導入菜單模板 。這就是 Go 如何讓你在另一個模板中導入一個模板。

頁腳模板包含靜態 HTML。索引頁面的模板使用頁眉和頁腳，並顯示一個簡單的 Hello Gin 消息：

|  |
| --- |
| <!--index.html-->  <!--Embed the header.html template at this location-->  {{ template "header.html" .}}  <h1>Hello Gin!</h1>  <!--Embed the footer.html template at this location-->  {{ template "footer.html" .}} |

Like the index template, templates for other pages will reuse the templates for the header and the footer in a similar manner.

與索引模板一樣，其他頁面的模板也會以類似的方式重新使用頁眉和頁腳的模板。

Completing and Validating the Setup

Once you have created the templates, it’s time to create the entry file for your application. We’ll create the main.go file for this with the simplest possible web application that will use the index template. We can do this using Gin in four steps:

完成並驗證安裝程序

一旦創建了模板，就可以創建應用程序的入口文件了。我們將 main.go 用最簡單的 Web 應用程序為此創建文件，該應用程序將使用索引模板。我們可以用四個步驟使用 Gin：

1. Create the router

The default way to create a router in Gin is as follows:

1.創建路由器

在 Gin 中創建路由器的默認方式如下所示：

|  |
| --- |
| router := gin.Default() |

This creates a router that can be used to define the build of the application.

這將創建一個可用於定義應用程序構建的路由器。

2. Load the templates

Once you have created the router, you can load all the templates like this:

2.加載模板

一旦你創建了路由器，你可以像這樣加載所有的模板：

|  |
| --- |
| router.LoadHTMLGlob("templates/\*") |

This loads all the template files located in the templates folder. Once loaded, these don’t have to be read again on every request making Gin web applications very fast.

這會加載位於該 templates 文件夾中的所有模板文件。一旦加載，這些都不必在每次請求時都要重新讀取，這使得 Gin Web 應用程序非常快速。

3. Define the route handler

At the heart of Gin is how you divide the application into various routes and define handlers for each route. We will create a route for the index page and an inline route handler.

3.定義路由處理程序

Gin 的核心是如何將應用程序分成不同的路由並為每條路由定義處理程序。我們將為索引頁和內聯路由處理程序創建路由。

|  |
| --- |
| router.GET("/", func(c \*gin.Context) {  // Call the HTML method of the Context to render a template  c.HTML(  // Set the HTTP status to 200 (OK)  http.StatusOK,  // Use the index.html template  "index.html",  // Pass the data that the page uses (in this case, 'title')  gin.H{  "title": "Home Page",  },  )  }) |

The router.GET method is used to define a route handler for a GET request. It takes in as parameters the route (/) and one or more route handlers which are just functions.

The route handler has a pointer to the context (gin.Context) as its parameter. This context contains all the information about the request that the handler might need to process it. For example, it includes information about the headers, cookies, etc.

The Context also has methods to render a response in HTML, text, JSON and XML formats. In this case, we use the context.HTML method to render an HTML template (index.html). The call to this method includes additional data in which the value of title is set to Home Page. This is a value that the HTML template can make use of. In this case, we use this value in the <title> tag in the header’s template.

該 router.GET 方法用於為 GET 請求定義路由處理程序。它接受作為參數的路由（/）和一個或多個路由處理程序，它們只是函數。

路由處理程序有一個指向 context（gin.Context）作為其參數的指針。此上下文包含有關處理程序可能需要處理的請求的所有信息。例如，它包含有關標題，Cookie 等的信息。

Context 還具有以 HTML，文本，JSON 和 XML 格式呈現響應的方法。在這種情況下，我們使用該 context.HTML 方法來呈現 HTML 模板（index.html）。對此方法的調用包括其中值 title 設置為的附加數據 Home Page。這是 HTML 模板可以使用的值。在這種情況下，我們在標頭模板 <title> 的標籤中使用這個值。

4. Start the application

To start the application, you can use the Run method of the router:

4.啟動應用程序

要啟動應用程序，您可以使用 Run 路由器的方法：

|  |
| --- |
| router.Run() |

This starts the application on localhost and serves on the 8080 port by default.

The complete main.go file looks as follows:

這會默認啟動應用程序 localhost 並在 8080 端口上提供服務。

完整的 main.go 文件如下所示：

|  |
| --- |
| // main.go  package main  import (  "net/http"  "github.com/gin-gonic/gin"  )  var router \*gin.Engine  func main() {  // Set the router as the default one provided by Gin  router = gin.Default()  // Process the templates at the start so that they don't have to be loaded  // from the disk again. This makes serving HTML pages very fast.  router.LoadHTMLGlob("templates/\*")  // Define the route for the index page and display the index.html template  // To start with, we'll use an inline route handler. Later on, we'll create  // standalone functions that will be used as route handlers.  router.GET("/", func(c \*gin.Context) {  // Call the HTML method of the Context to render a template  c.HTML(  // Set the HTTP status to 200 (OK)  http.StatusOK,  // Use the index.html template  "index.html",  // Pass the data that the page uses (in this case, 'title')  gin.H{  "title": "Home Page",  },  )  })  // Start serving the application  router.Run()  } |

To execute the application from the command line, go to your application directory and execute the following command:

要從命令行執行應用程序，請轉到您的應用程序目錄並執行以下命令：

|  |
| --- |
| go build -o app |

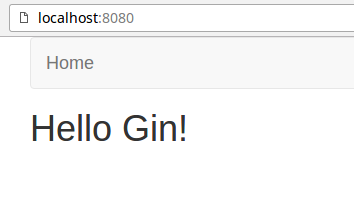
This will build your application and create an executable named app which you can run as follows:

這將構建您的應用程序並創建一個可執行的命令 app，您可以運行如下：

|  |
| --- |
| ./app |

If all goes well, you should be able to access your application at http://localhost:8080 and it should look like this:

如果一切順利，您應該可以訪問您的應用程序， http://localhost:8080 它應該如下所示：



The directory structure of your application at this stage should be as follows:

在這個階段你的應用程序的目錄結構應該如下所示：

|  |
| --- |
| ├── main.go  └── templates  ├── footer.html  ├── header.html  ├── index.html  └── menu.html |

Displaying the List of Articles

In this section, we’ll add the functionality to display the list of all articles on the index page.

顯示文章列表

在本節中，我們將添加功能來顯示索引頁面上所有文章的列表。

Setting Up the Route

In the previous section, we created the route and the route definition in the main.go file itself. As the application grows, it would make sense to move the routes definitions in its own file. We’ll create a function initializeRoutes() in the routes.go file and call this function from the main() function to set up all the routes. Instead of defining the route handler inline, we’ll define them as separate functions.

After making these changes, the routes.go file will contain the following:

設置路由

在前一節中，我們在 main.go 文件本身中創建了路由和路由定義 。隨著應用程序的增長，將路徑定義移動到其自己的文件中是有意義的。我們將 initializeRoutes() 在該 routes.go 文件中創建一個函數，並從該 main()函數調用該函數來設置所有路由。我們將定義它們作為單獨的函數，而不是定義內聯的路由處理程序。

進行這些更改後，該 routes.go 文件將包含以下內容：

|  |
| --- |
| // routes.go  package main  func initializeRoutes() {  // Handle the index route  router.GET("/", showIndexPage)  } |

Since we’ll be displaying the list of articles on the index page, we don’t need to define any additional routes after we’ve refactored the code.

The main.go file should contain the following code:

由於我們將在索引頁面上顯示文章列表，因此在重構代碼後，我們不需要定義任何其他路由。

該 main.go 文件應該包含以下代碼：

|  |
| --- |
| // main.go  package main  import "github.com/gin-gonic/gin"  var router \*gin.Engine  func main() {  // Set the router as the default one provided by Gin  router = gin.Default()  // Process the templates at the start so that they don't have to be loaded  // from the disk again. This makes serving HTML pages very fast.  router.LoadHTMLGlob("templates/\*")  // Initialize the routes  initializeRoutes()  // Start serving the application  router.Run()  } |

Designing the Article Model

We will keep the article structure simple with just three fields – Id, Title and Content. This can be represented with a struct as follows:

設計文章模型

我們將保持簡單的文章結構只有三個字段屬性 - Id，Title 和 Content。這可以用結構體 struct 表示如下：

|  |
| --- |
| type article struct {  ID int `json:"id"`  Title string `json:"title"`  Content string `json:"content"`  } |

Most applications will use a database to persist the data. To keep things simple, we will keep the list of articles in memory and will initialize the list with two hard-coded articles as follows:

大多數應用程序將使用數據庫來保存數據。為了簡單起見，我們將保留內存中的文章列表，並用兩個硬編碼文章初始化列表，如下所示：

|  |
| --- |
| var articleList = []article{  article{ID: 1, Title: "Article 1", Content: "Article 1 body"},  article{ID: 2, Title: "Article 2", Content: "Article 2 body"},  } |

We will place the above code in a new file named models.article.go. At this stage, we need a function that will return the list of all articles. We will name this function getAllArticles() and place it in the same file. We will also write a test for it. This test will be named TestGetAllArticles and will be placed in the models.article\_test.go file.

Let’s start by creating the unit test (TestGetAllArticles) for the getAllArticles() function. After creating this unit test, the models.article\_test.go file should contain the following code:

我們將把上面的代碼放在一個名為的新文件中 models.article.go。在這個階段，我們需要一個函數返回所有文章的列表。我們將命名這個函數 getAllArticles() 並將它放在同一個文件中。我們也會為它寫一個測試。該測試將被命名 TestGetAllArticles 並將被放置在 models.article\_test.go 文件中。

我們首先創建函數的單元測試（TestGetAllArticles） getAllArticles()。創建此單元測試後，該 models.article\_test.go 文件應包含以下代碼：

|  |
| --- |
| // models.article\_test.go  package main  import "testing"  // Test the function that fetches all articles  func TestGetAllArticles(t \*testing.T) {  alist := getAllArticles()  // Check that the length of the list of articles returned is the  // same as the length of the global variable holding the list  if len(alist) != len(articleList) {  t.Fail()  }  // Check that each member is identical  for i, v := range alist {  if v.Content != articleList[i].Content ||  v.ID != articleList[i].ID ||  v.Title != articleList[i].Title {  t.Fail()  break  }  }  } |

This unit test uses the getAllArticles() function to fetch the list of all articles. This test first makes sure that the article list fetched by this function and the article list present in the global variable articleList are identical. It then loops over the article list to verify that each article is identical. The test fails if either of these two checks fail.

Once we have written the test, we can proceed to write the actual code. The models.article.go file should contain the following code:

該單元測試使用該 getAllArticles() 函數來獲取所有文章的列表。此測試首先確保此函數獲取的文章列表和全局變量 articleList 中存在的文章列表完全相同。然後它遍歷文章列表以驗證每篇文章是否相同。如果這兩項檢查中的任何一項失敗，則測試失敗。

一旦我們編寫了測試，我們就可以繼續編寫實際的代碼。該 models.article.go 文件應該包含以下代碼：

|  |
| --- |
| // models.article.go  package main  type article struct {  ID int json:"id"  Title string json:"title"  Content string json:"content"  }  // For this demo, we're storing the article list in memory  // In a real application, this list will most likely be fetched  // from a database or from static files  var articleList = []article{  article{ID: 1, Title: "Article 1", Content: "Article 1 body"},  article{ID: 2, Title: "Article 2", Content: "Article 2 body"},  }  // Return a list of all the articles  func getAllArticles() []article {  return articleList  } |

Creating the View Template

Since the list of articles will be displayed on the index page, we don’t need to create a new template. However, we do need to change the index.html template to replace the current content with the list of articles.

To make this change, we’ll assume that the list of articles will be passed to the template in a variable named payload. With this assumption, the following snippet should show the list of all articles:

創建視圖模板

由於文章列表將顯示在索引頁面上，因此我們不需要創建新模板。但是，我們確實需要更改 index.html 模板以將當前內容替換為文章列表。

為了做出這個改變，我們假定文章列表將被傳遞給名為變量的模板 payload。有了這個假設，下面的代碼片段應該顯示所有文章的列表：

|  |
| --- |
| {{range .payload }}  <!--Create the link for the article based on its ID-->  <a href="/article/view/{{.ID}}">  <!--Display the title of the article -->  <h2>{{.Title}}</h2>  </a>  <!--Display the content of the article-->  <p>{{.Content}}</p>  {{end}} |

This snippet will loop over all items in the payload variable and display the title and the content of each article. The above snippet will also link to each article. However, since we have not yet defined route handlers for displaying individual articles, these links won’t work as expected.

The updated index.html file should contain the following code:

該片段將循環 payload 顯示變量中的所有項目，並顯示每篇文章的標題和內容。上面的代碼片段也會鏈接到每篇文章。但是，由於我們尚未定義用於顯示單篇文章的路由處理程序，因此這些鏈接無法按預期工作。

更新後的 index.html 文件應包含以下代碼：

|  |
| --- |
| <!--index.html-->  <!--Embed the header.html template at this location-->  {{ template "header.html" .}}  <!--Loop over the payload variable, which is the list of articles-->  {{range .payload }}  <!--Create the link for the article based on its ID-->  <a href="/article/view/{{.ID}}">  <!--Display the title of the article -->  <h2>{{.Title}}</h2>  </a>  <!--Display the content of the article-->  <p>{{.Content}}</p>  {{end}}  <!--Embed the footer.html template at this location-->  {{ template "footer.html" .}} |

Specifying the Requirement for the Route Handler With a Unit Test

Before we create the handler for the index route, we will create a test to define the expected behavior of this route handler. This test will check for the following conditions:

1. The handler responds with an HTTP status code of 200,
2. The returned HTML contains a title tag containing the text Home Page.

The code for the test will be placed in the TestShowIndexPageUnauthenticated function in the handlers.article\_test.go file. We will place helper functions used by this function in the common\_test.go file.

The content of handlers.article\_test.go is as follows:

用單元測試指定路由處理程序的要求

在為索引路由創建處理程序之前，我們將創建一個測試來定義此路由處理程序的預期行為。該測試將檢查以下情況：

1. 處理程序以 200 的 HTTP 狀態碼進行響應，
2. 返回的 HTML 包含一個包含文本的標題標籤 Home Page。

測試代碼將被放置在文件中的 TestShowIndexPageUnauthenticated 函數中 handlers.article\_test.go。我們將把這個函數使用的幫助函數放在 common\_test.go 文件中。

內容 handlers.article\_test.go 如下：

|  |
| --- |
| // handlers.article\_test.go  package main  import (  "io/ioutil"  "net/http"  "net/http/httptest"  "strings"  "testing"  )  // Test that a GET request to the home page returns the home page with  // the HTTP code 200 for an unauthenticated user  func TestShowIndexPageUnauthenticated(t \*testing.T) {  r := getRouter(true)  r.GET("/", showIndexPage)  // Create a request to send to the above route  req, \_ := http.NewRequest("GET", "/", nil)  testHTTPResponse(t, r, req, func(w \*httptest.ResponseRecorder) bool {  // Test that the http status code is 200  statusOK := w.Code == http.StatusOK  // Test that the page title is "Home Page"  // You can carry out a lot more detailed tests using libraries that can  // parse and process HTML pages  p, err := ioutil.ReadAll(w.Body)  pageOK := err == nil && strings.Index(string(p), "<title>Home Page</title>") > 0  return statusOK && pageOK  })  } |

The content of common\_test.go is as follows:

內容 common\_test.go 如下：

|  |
| --- |
| package main  import (  "net/http"  "net/http/httptest"  "os"  "testing"  "github.com/gin-gonic/gin"  )  var tmpArticleList []article  // This function is used for setup before executing the test functions  func TestMain(m \*testing.M) {  //Set Gin to Test Mode  gin.SetMode(gin.TestMode)  // Run the other tests  os.Exit(m.Run())  }  // Helper function to create a router during testing  func getRouter(withTemplates bool) \*gin.Engine {  r := gin.Default()  if withTemplates {  r.LoadHTMLGlob("templates/\*")  }  return r  }  // Helper function to process a request and test its response  func testHTTPResponse(t \*testing.T, r \*gin.Engine, req \*http.Request, f func(w \*httptest.ResponseRecorder) bool) {  // Create a response recorder  w := httptest.NewRecorder()  // Create the service and process the above request.  r.ServeHTTP(w, req)  if !f(w) {  t.Fail()  }  }  // This function is used to store the main lists into the temporary one  // for testing  func saveLists() {  tmpArticleList = articleList  }  // This function is used to restore the main lists from the temporary one  func restoreLists() {  articleList = tmpArticleList  } |

To implement this test, we have written some helper functions. These will also help us reduce boilerplate code when we write additional tests to test similar functionality.

The TestMain function sets Gin to use the test mode and calls the rest of the test functions. The getRouter function creates and returns a router in a manner similar to the main application. The saveLists() function saves the original article list in a temporary variable. This temporary variable is used by the restoreLists() function to restore the article list to its initial state after a unit test is executed.

Finally, the testHTTPResponse function executes the function passed in to see if it returns a boolean true value — indicating a successful test, or not. This function helps us avoid duplicating the code needed to test the response of an HTTP request.

To check the HTTP code and the returned HTML, we’ll do the following:

1. Create a new router,
2. Define a route to use the same handler that the main app uses (showIndexPage),
3. Create a new request to access this route,
4. Create a function that processes the response to test the HTTP code and HTML, and
5. Call testHTTPResponse() with this new function to complete the test.

為了實現這個測試，我們寫了一些幫助函數。當我們編寫額外的測試來測試類似的功能時，這些也將幫助我們減少樣板代碼。

該 TestMain 函數設置 Gin 使用測試模式並調用其餘的測試功能。該 getRouter 函數以類似於主應用程序的方式創建並返回路由器。該 saveLists() 函數將原始文章列表保存在一個臨時變量中。該 restoreLists() 函數使用該臨時變量在執行單元測試後將文章列表恢復到初始狀態。

最後，該 testHTTPResponse 函數執行傳入的函數以查看它是否返回布爾值 true 值 - 表示測試成功或不成功。此功能可幫助我們避免重複測試 HTTP 請求響應所需的代碼。

要檢查 HTTP 代碼和返回的 HTML，我們將執行以下操作：

1. 創建一個新的路由器，
2. 定義一個使用主應用程序使用的相同處理程序的路由（showIndexPage），
3. 創建一個訪問此路線的新請求，
4. 創建一個處理響應以測試 HTTP 代碼和 HTML 的函數
5. testHTTPResponse() 用這個新功能調用完成測試。

Creating the Route Handler

We will create all route handlers for article related functionality in the handlers.article.go file. The handler for the index page, showIndexPage performs the following tasks:

創建路由處理程序

我們將在 handlers.article.go 文件中為文章相關功能創建所有路由處理程序 。索引頁面的處理程序 showIndexPage 執行以下任務：

1. Fetches the list of articles

This can be done using the getAllArticles function defined previously:

1.獲取文章列表

這可以使用 getAllArticles 之前定義的函數來完成：

|  |
| --- |
| articles := getAllArticles() |

2. Renders the index.html template passing it the article list

This can be done using the code below:

2.呈現 index.html 通過文章列表的模板

這可以使用下面的代碼完成：

|  |
| --- |
| c.HTML(  // Set the HTTP status to 200 (OK)  http.StatusOK,  // Use the index.html template  "index.html",  // Pass the data that the page uses  gin.H{  "title": "Home Page",  "payload": articles,  },  ) |

The only difference from the version in the previous section is that we’re passing the list of articles which will be accessed in the template by the variable named payload.

The handlers.article.go file should contain the following code:

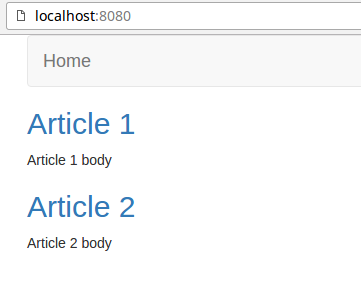
與前一節中唯一的不同之處在於，我們傳遞了將通過名為變量的模板訪問的文章列表 payload。

該 handlers.article.go 文件應該包含以下代碼：

|  |
| --- |
| // handlers.article.go  package main  import (  "net/http"  "github.com/gin-gonic/gin"  )  func showIndexPage(c \*gin.Context) {  articles := getAllArticles()  // Call the HTML method of the Context to render a template  c.HTML(  // Set the HTTP status to 200 (OK)  http.StatusOK,  // Use the index.html template  "index.html",  // Pass the data that the page uses  gin.H{  "title": "Home Page",  "payload": articles,  },  )  } |

If you now build and run your application and visit http://localhost:8080 in a browser, it should look like this:

如果您現在構建並運行應用程序並 http://localhost:8080 在瀏覽器中訪問，它應該如下所示：



These are the new files added in this section:

這些是本節添加的新文件：

|  |
| --- |
| ├── common\_test.go  ├── handlers.article.go  ├── handlers.article\_test.go  ├── models.article.go  ├── models.article\_test.go  └── routes.go |

Displaying a Single Article

In the last section, while we displayed a list of articles, the links to the articles didn’t work. In this section, we’ll add handlers and templates to display an article when it is selected.

顯示單個文章

在最後一節中，雖然我們顯示了一篇文章列表，但文章的鏈接不起作用。在本節中，我們將添加處理程序和模板以在選中文章時顯示文章。

Setting Up the Route

We can set up a new route to handle requests for a single article in the same manner as in the previous route. However, we need to account for the fact that while the handler for all articles would be the same, the URL for each article would be different. Gin allows us to handle such conditions by defining route parameters as follows:

設置路由

我們可以建立一條新路由，以與前一路由相同的方式處理單篇文章的請求。但是，我們需要說明的是，雖然所有文章的處理程序都是相同的，但每篇文章的 URL 都會有所不同。Gin 允許我們通過如下定義路由參數來處理這些情況：

|  |
| --- |
| router.GET("/article/view/:article\_id", getArticle) |

This route will match all requests matching the above path and will store the value of the last part of the route in the route parameter named article\_id which we can access in the route handler. For this route, we will define the handler in a function named getArticle.

The updated routes.go file should contain the following code:

該路由將匹配與上述路徑相匹配的所有請求，並將路由的最後一部分的值存儲在 article\_id 我們可以在路由處理程序中訪問的路由參數中。對於這條路由，我們將在名為的函數中定義處理程序 getArticle。

更新後的 routes.go 文件應包含以下代碼：

|  |
| --- |
| // routes.go  package main  func initializeRoutes() {  // Handle the index route  router.GET("/", showIndexPage)  // Handle GET requests at /article/view/some\_article\_id  router.GET("/article/view/:article\_id", getArticle)  } |

Creating the View Templates

We need to create a new template at templates/article.html to display the content of a single article. This can be created in a manner similar to the index.html template. However, instead of the payload variable containing the list of articles, in this case it will contain a single article.

You can see the content of article.html in the Github repository.

創建視圖模板

我們需要創建一個新模板 templates/article.html 來顯示單篇文章的內容。這可以以與 index.html 模板類似的方式創建 。但是，而不是 payload 包含文章列表的變量，在這種情況下，它將包含一篇文章。

您可以在 Github 存儲庫中看到 article.html 的內容。

Specifying the Requirement for the Route Handler With a Unit Test

The test for the handler of this route will check for the following conditions:

1. The handler responds with an HTTP status code of 200,
2. The returned HTML contains a title tag containing the title of the article that was fetched.

The code for the test will be placed in the TestArticleUnauthenticated function in the handlers.article\_test.go file. We will place helper functions used by this function in the common\_test.go file.

用單元測試指定路由處理程序的要求

對這條路線的處理程序的測試將檢查以下條件：

1. 處理程序以 200 的 HTTP 狀態碼進行響應，
2. 返回的 HTML 包含一個標題標籤，其中包含所提取文章的標題。

測試代碼將被放置在文件中的 TestArticleUnauthenticated 函數中 handlers.article\_test.go。我們將把這個函數使用的幫助函數放在 common\_test.go 文件中。

Creating the Route Handler

The handler for the article page, getArticle performs the following tasks:

創建路由處理程序

文章頁面的處理程序 getArticle 執行以下任務：

1. Extracts the ID of the article to display

To fetch and display the right article, we first need to extract its ID from the context. This can be extracted as follows:

1.提取要顯示的文章的 ID

要獲取並顯示正確的文章，我們首先需要從上下文中提取其 ID。這可以提取如下：

|  |
| --- |
| c.Param("article\_id") |

where c is the Gin Context which is a parameter to any route handler when using Gin.

其中 c 是 Gin 上下文的一個參數。

2. Fetches the article

This can be done using the getArticleByID() function defined in the models.article.go file:

2.獲取文章

這可以使用文件中 getArticleByID() 定義的函數 完成 models.article.go：

|  |
| --- |
| article, err := getArticleByID(articleID) |

The getArticleByID function (in models.article.go) is as follows:

的 getArticleByID（在功能上 models.article.go）如下：

|  |
| --- |
| func getArticleByID(id int) (\*article, error) {  for \_, a := range articleList {  if a.ID == id {  return &a, nil  }  }  return nil, errors.New("Article not found")  } |

This function loops through the article list and returns the article whose ID matches the ID passed in. If no matching article is found it returns an error indicating the same.

該函數遍歷文章列表並返回其 ID 與傳入的 ID 相匹配的文章。如果找不到匹配的文章，則返回一個指示相同的錯誤。

3. Renders the article.html template passing it the article

This can be done using the code below:

3.呈現 article.html 將文章傳遞給它的模板

這可以使用下面的代碼完成：

|  |
| --- |
| c.HTML(  // Set the HTTP status to 200 (OK)  http.StatusOK,  // Use the article.html template  "article.html",  // Pass the data that the page uses  gin.H{  "title": article.Title,  "payload": article,  },  ) |

The updated handlers.article.go file should contain the following code:

更新後的 handlers.article.go 文件應包含以下代碼：

|  |
| --- |
| // handlers.article.go  package main  import (  "net/http"  "strconv"  "github.com/gin-gonic/gin"  )  func showIndexPage(c \*gin.Context) {  articles := getAllArticles()  // Call the HTML method of the Context to render a template  c.HTML(  // Set the HTTP status to 200 (OK)  http.StatusOK,  // Use the index.html template  "index.html",  // Pass the data that the page uses  gin.H{  "title": "Home Page",  "payload": articles,  },  )  }  func getArticle(c \*gin.Context) {  // Check if the article ID is valid  if articleID, err := strconv.Atoi(c.Param("article\_id")); err == nil {  // Check if the article exists  if article, err := getArticleByID(articleID); err == nil {  // Call the HTML method of the Context to render a template  c.HTML(  // Set the HTTP status to 200 (OK)  http.StatusOK,  // Use the index.html template  "article.html",  // Pass the data that the page uses  gin.H{  "title": article.Title,  "payload": article,  },  )  } else {  // If the article is not found, abort with an error  c.AbortWithError(http.StatusNotFound, err)  }  } else {  // If an invalid article ID is specified in the URL, abort with an error  c.AbortWithStatus(http.StatusNotFound)  }  } |

If you now build and run your application and visit http://localhost:8080/article/view/1 in a browser, it should look like this:

如果您現在構建並運行應用程序並 http://localhost:8080/article/view/1 在瀏覽器中訪問 ，它應該如下所示：



The new files added in this section are as follows:

本節添加的新文件如下所示：

|  |
| --- |
| └── templates  └── article.html |

Responding With JSON/XML

In this section, we will refactor the application a bit so that, depending on the request headers, our application can respond in HTML, JSON or XML format.

使用 JSON / XML 進行響應

在本節中，我們將重構應用程序，以便根據請求標頭，我們的應用程序可以使用 HTML，JSON 或 XML 格式進行響應。

Creating a Reusable Function

So far, we’ve been using the HTML method of Gin’s context to render directly from route handlers. This is fine when we always want to render HTML. However, if we want to change the format of the response based on the request, we should refactor this part out into a single function that takes care of the rendering. By doing this, we can let the route handler focus on validation and data fetching.

A route handler has to do the same kind of validation, data fetching and data processing irrespective of the desired response format. Once this part is done, this data can be used to generate a response in the desired format. If we need an HTML response, we can pass this data to the HTML template and generate the page. If we need a JSON response, we can convert this data to JSON and send it back. Likewise for XML.

We’ll create a render function in main.go that will be used by all the route handlers. This function will take care of rendering in the right format based on the request’s Accept header.

In Gin, the Context passed to a route handler contains a field named Request. This field contains the Header field which contains all the request headers. We can use the Get method on Header to extract the Accept header as follows:

創建可重用函數

到目前為止，我們一直使用 HTMLGin 上下文的方法直接從路由處理程序中進行渲染。當我們總想渲染 HTML 時，這很好。但是，如果我們想根據請求更改響應的格式，我們應該將此部分重構為一個負責渲染的函數。通過這樣做，我們可以讓路由處理程序專注於驗證和數據獲取。

路由處理程序必須執行相同類型的驗證，數據讀取和數據處理，而不考慮所需的響應格式。完成此部分後，可以使用此數據以所需格式生成響應。如果我們需要 HTML 響應，我們可以將這些數據傳遞給 HTML 模板並生成頁面。如果需要 JSON 響應，我們可以將這些數據轉換為 JSON 並發送回去。對於 XML 也是如此。

我們將創建一個 render 函數，main.go 供所有路由處理程序使用。這個函數將負責基於請求 Accept 頭以正確的格式進行渲染。

在 Gin 中，Context 傳遞給路由處理程序包含一個名為的字段 Request。該字段包含 Header 包含所有請求標題的字段。我們可以使用該 Get 方法 Header 來提取 Accept 標題，如下所示：

|  |
| --- |
| // c is the Gin Context  c.Request.Header.Get("Accept") |

* If this is set to application/json, the function will render JSON,
* If this is set to application/xml, the function will render XML, and
* If this is set to anything else or is empty, the function will render HTML.

The complete render function is as follows:

* 如果設置為 application/json，該函數將呈現 JSON，
* 如果設置為 application/xml，該函數將呈現 XML 和
* 如果這設置為其他任何內容或為空，則該函數將呈現 HTML。

完整的 render 功能如下：

|  |
| --- |
| // Render one of HTML, JSON or CSV based on the 'Accept' header of the request  // If the header doesn't specify this, HTML is rendered, provided that  // the template name is present  func render(c \*gin.Context, data gin.H, templateName string) {  switch c.Request.Header.Get("Accept") {  case "application/json":  // Respond with JSON  c.JSON(http.StatusOK, data["payload"])  case "application/xml":  // Respond with XML  c.XML(http.StatusOK, data["payload"])  default:  // Respond with HTML  c.HTML(http.StatusOK, templateName, data)  }  } |

Modifying the Requirement for the Route Handlers With a Unit Test

Since we are now expecting JSON and XML responses if the respective headers are set, we should add tests to the handlers.article\_test.go file to test these conditions. We will add tests to:

1. Test that the application returns a JSON list of articles when the Accept header is set to application/json
2. Test the application returns an article in XML format when the Accept header is set to application/xml

These will be added as functions named TestArticleListJSON and TestArticleXML.

使用單元測試修改路由處理程序的要求

由於我們現在預計會設置 JSON 和 XML 響應，因此我們應該在 handlers.article\_test.go 文件中添加測試以測試這些條件。我們將添加測試到：

1. 測試應用程序在 Accept 標題設置為時返回文章的 JSON 列表 application/json
2. 測試應用程序在 Accept 標題設置為時返回 XML 格式的文章 application/xml

這些將被添加為名為 TestArticleListJSON 和 TestArticleXML 函數。

Updating the Route Handlers

The route handlers don’t really need to change much as the logic for rendering in any format is pretty much the same. All that needs to be done is use the render function instead of rendering using the c.HTML methods.

For example, the showIndexPage route handler will change from

更新路由處理程序

路由處理程序並不需要太多改變，因為以任何格式呈現的邏輯幾乎都是相同的。所有需要做的就是使用 render 函數而不是使用 c.HTML 方法進行渲染。

例如，showIndexPage 路由處理程序將從中更改

|  |
| --- |
| func showIndexPage(c \*gin.Context) {  articles := getAllArticles()  // Call the HTML method of the Context to render a template  c.HTML(  // Set the HTTP status to 200 (OK)  http.StatusOK,  // Use the index.html template  "index.html",  // Pass the data that the page uses  gin.H{  "title": "Home Page",  "payload": articles,  },  )  } |

to

至

|  |
| --- |
| func showIndexPage(c \*gin.Context) {  articles := getAllArticles()  // Call the render function with the name of the template to render  render(c, gin.H{  "title": "Home Page",  "payload": articles}, "index.html")  } |

Retrieving the List of Articles in JSON Format

To see our latest updates in action, build and run your application. Then execute the following command:

以 JSON 格式檢索文章列表

要查看我們的最新動態更新，請構建並運行您的應用程序。然後執行以下命令：

|  |
| --- |
| curl -X GET -H "Accept: application/json" http://localhost:8080/ |

This should return a response as follows:

這應該返回一個響應，如下所示：

|  |
| --- |
| [{"id":1,"title":"Article 1","content":"Article 1 body"},{"id":2,"title":"Article 2","content":"Article 2 body"}] |

As you can see, our request got a response in the JSON format because we set the Accept header to application/json.

正如你所看到的，我們的請求得到了 JSON 格式的響應，因為我們設置了 Accept 標頭 application/json。

Retrieving an Article in XML Format

Let’s now get our application to respond with the details of a particular article in the XML format. To do this, first, start your application as mentioned above. Now execute the following command:

以 XML 格式檢索文章

現在讓我們的應用程序以 XML 格式回覆特定文章的細節。要做到這一點，首先要開始你的應用程序，如上所述。現在執行以下命令：

|  |
| --- |
| curl -X GET -H "Accept: application/xml" http://localhost:8080/article/view/1 |

This should return a response as follows:

這應該返回一個響應，如下所示：

|  |
| --- |
| <article><ID>1</ID><Title>Article 1</Title><Content>Article 1 body</Content></article> |

As you can see, our request got a response in the XML format because we set the Accept header to application/xml.

正如您所看到的，我們的請求以 XML 格式得到響應，因為我們將 Accept 標頭設置 為 application/xml。

Testing the Application

Since we’ve been using tests to create specifications for our route handlers and models, we should constantly be running them to ensure that the functions work as expected. Let’s now run the tests that we have written and see the results. In your project directory, execute the following command:

測試應用程序

由於我們一直在使用測試來為我們的路由處理程序和模型創建規範，因此我們應該不斷運行它們以確保功能按預期工作。現在讓我們運行我們編寫的測試並查看結果。在您的項目目錄中，執行以下命令：

|  |
| --- |
| go test -v |

Executing this command should result in something similar to this:

執行這個命令應該會輸出如下結果：

|  |
| --- |
| === RUN TestShowIndexPageUnauthenticated  [GIN] 2016/06/14 - 19:07:26 | 200 | 183.315µs | | GET /  --- PASS: TestShowIndexPageUnauthenticated (0.00s)  === RUN TestArticleUnauthenticated  [GIN] 2016/06/14 - 19:07:26 | 200 | 143.789µs | | GET /article/view/1  --- PASS: TestArticleUnauthenticated (0.00s)  === RUN TestArticleListJSON  [GIN] 2016/06/14 - 19:07:26 | 200 | 51.087µs | | GET /  --- PASS: TestArticleListJSON (0.00s)  === RUN TestArticleXML  [GIN] 2016/06/14 - 19:07:26 | 200 | 38.656µs | | GET /article/view/1  --- PASS: TestArticleXML (0.00s)  === RUN TestGetAllArticles  --- PASS: TestGetAllArticles (0.00s)  === RUN TestGetArticleByID  --- PASS: TestGetArticleByID (0.00s)  PASS  ok github.com/demo-apps/go-gin-app 0.084s |

As can be seen in this output, this command runs all the tests that we have written and, in this case, indicates that our application is working as we intend it to. If you take a close look at the output, you’ll notice that Go made HTTP requests in the course of testing the route handlers.

從這個輸出中可以看出，這個命令運行我們編寫的所有測試，在這種情況下，表明我們的應用程序正在按照我們的意圖工作。如果仔細觀察輸出，您會注意到 Go 在測試路由處理程序的過程中發出了 HTTP 請求。

Continuous Integration for Go on Semaphore

在 Semaphore 上進行 GO 持續集成

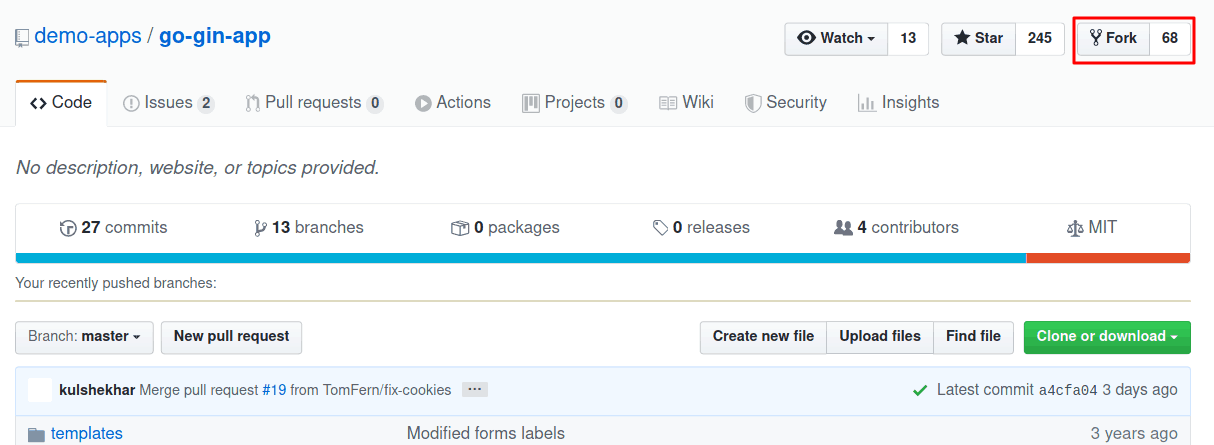
Continuous Integration (CI) can test and build the application for us in a fast and clean environment. When we are ready to publish, Continuous Delivery (CD) can make the releases, secure in the knowledge that the code passed all our tests.

First, we have to get all the code in GitHub:

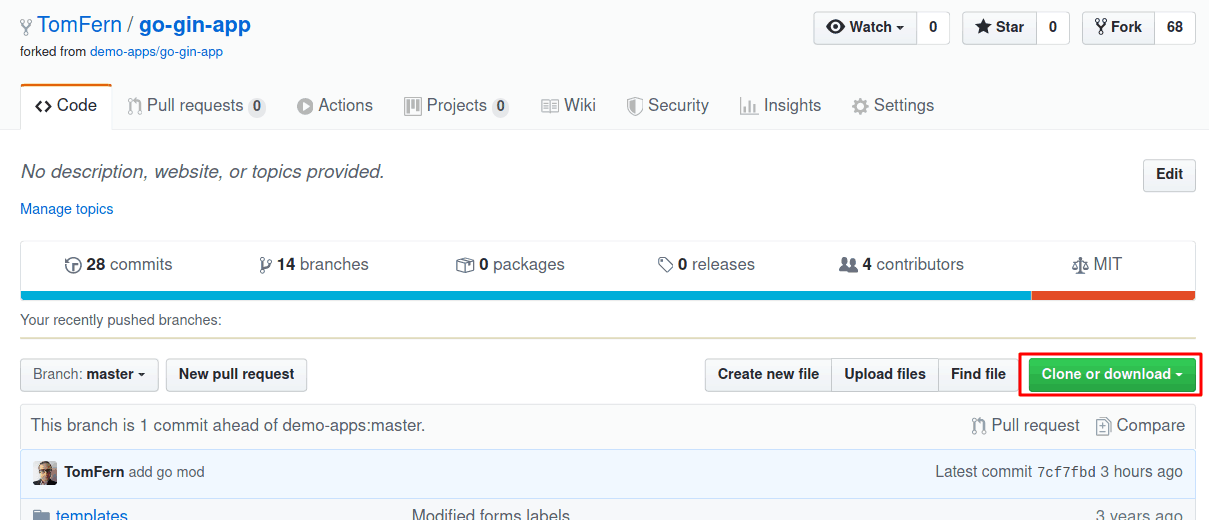
* If you created the application from scratch:
  + Create a GitHub repository.
  + Push all your code:

|  |
| --- |
| $ git init  $ git remote add YOUR\_REPOSITORY\_URL  $ git add -A  $ git commit -m "initial commit"  $ git push origin master |

* If you prefer to work with a ready-to-use example:
  + Go to the example repository.
  + Fork by using the Fork button on the top right corner:



* Use the Clone or download button to see the URL of your new repository:



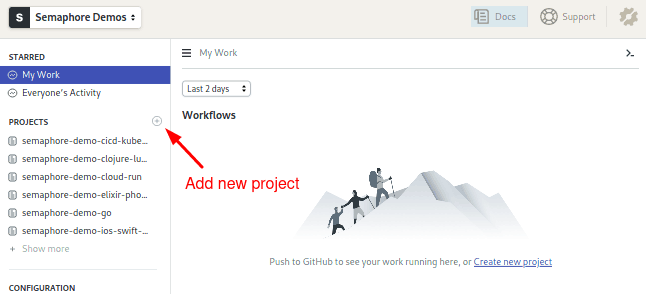
Finally, clone the code to your machine:

|  |
| --- |
| $ git clone YOUR\_REPOSITORY\_URL |

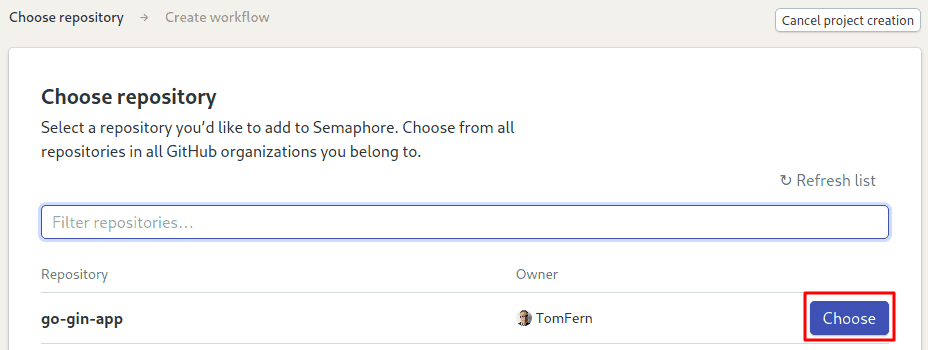
Add Semaphore to Your Project

Adding CI/CD to your project is completely free and takes only a few minutes:

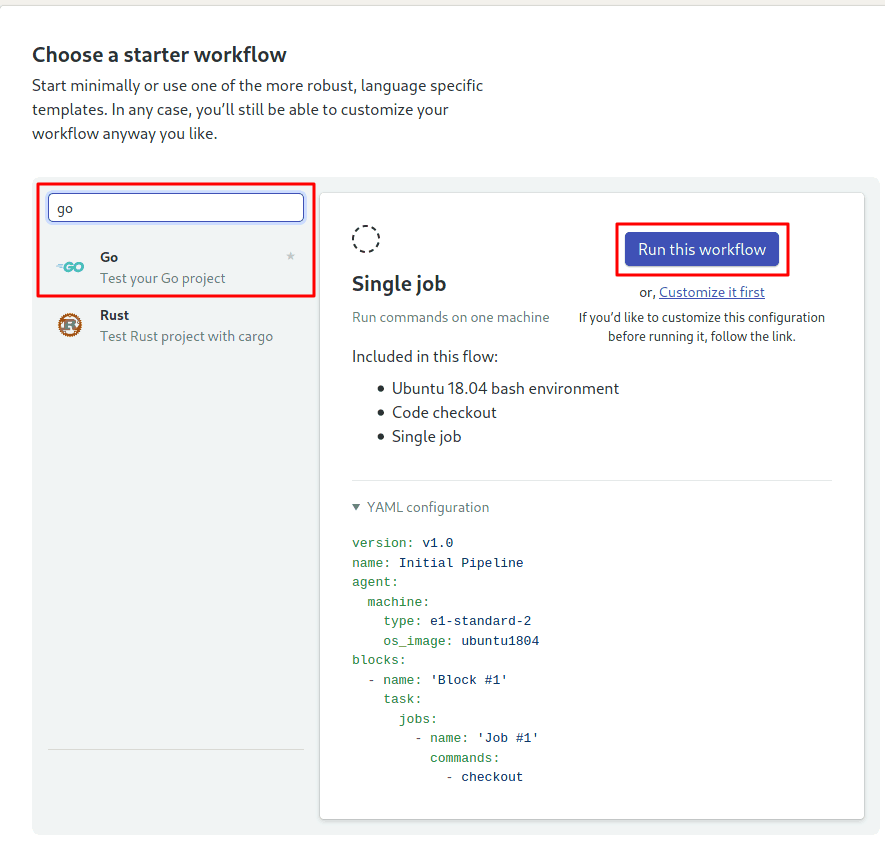
* Sign up for a Semaphore account using the Try it FREE button on the top-right corner.
* On the left navigation menu, use the + (plus sign) icon next to Projects:



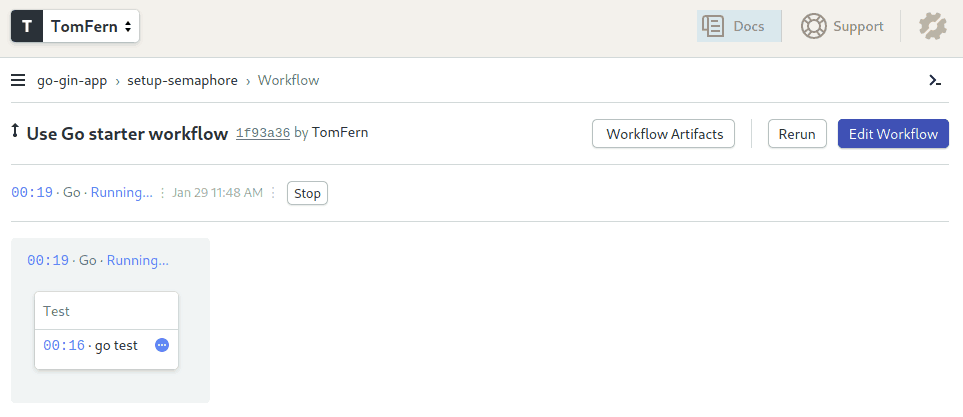
* Find you project and click on Choose:



* Select the Go starter workflow and click on Run this Workflow:



* A CI pipeline will be created and start immediately:



That’s it, on every push the CI pipeline will test and build the application.

Improving the Pipeline

The starter CI pipeline should work seamlessly without any additional setup. We can, however, make some improvements:

* Dependencies have to be downloaded every time. We can use a cache to keep them and speed up things.
* We are not using any module management at all. We need to may of locking which version of Gin we want to build against.
* Testing and building are on the same job. We should split it in different jobs so it’s easier for us to later add more tests.

Module Management in Go

Like most languages, Go has several competing module management mechanisms. We’ll use go modules which were introduced on Go 1.11 and is now an official part of the language.

* Pull the semaphore-setup branch from your repository. This branch was automatically created by Semaphore during setup:

|  |
| --- |
| $ git pull origin semaphore-setup  $ git checkout semaphore-setup |

* Initialize the module manifest, use your repository URL below (omit https://):

|  |
| --- |
| $ go mod init github.com/YOUR\_REPOSITORY\_URL |

* Add the Gin dependency. The following command adds the development version of Gin:

|  |
| --- |
| $ go get github.com/gin-gonic/gin@master |

* Push the updates to the repository:

|  |
| --- |
| $ git add go.mod go.sum  $ git commit -m "add go modules"  $ git push origin setup-semaphore |

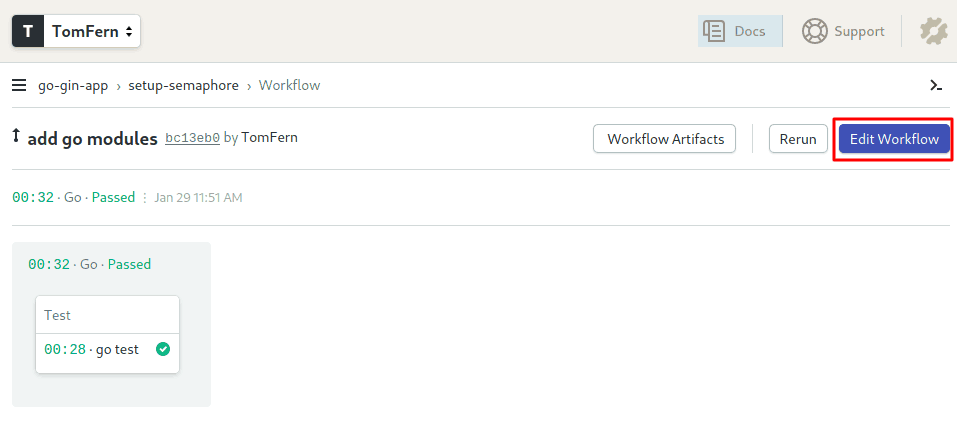
When adding a dependency, we can be selective on which version to use:

to use a branch: go get github.com/gin-gonic/gin@master

to use a tag: go get github.com/gin-gonic/gin@1.15.0

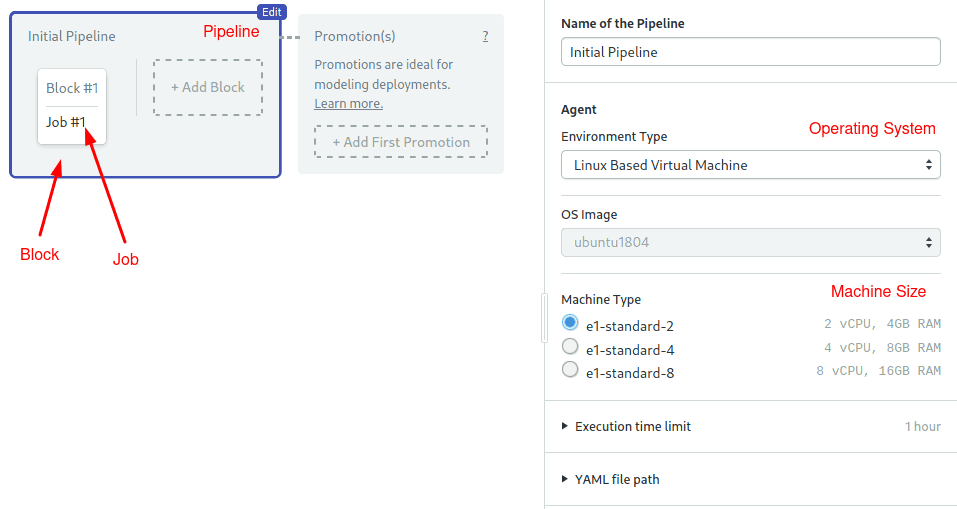
to pick a specific commit: go get github.com/gin-gonic/gin@07c0f05f244589ff4a02320a01ad0a1fc102cbd5

Use the Edit Workflow button to open the Workflow Builder:



The main elements of the builder are:

* Pipeline: A pipeline has a specific objective, e.g. testing. Pipelines are made of blocks that are executed from left to right in an agent.
* Agent: The agent is the virtual machine that powers the pipeline. We have three machine types to choose from. The machine runs an optimized Ubuntu 18.04 image with build tools for many languages.
* Block: blocks group jobs that can be executed in parallel. Jobs in a block usually have similar commands and configurations. Once all jobs in a block complete, the next block begins.
* Job: jobs define the commands that do the work. They inherit their configuration from their parent block.

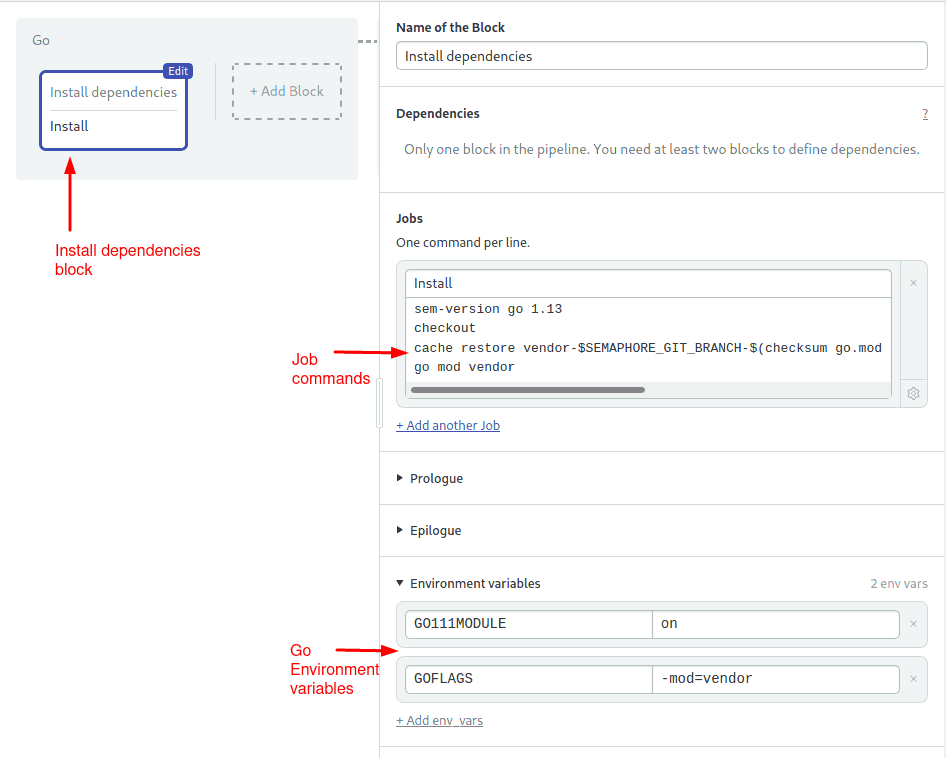


We’ll make an improved version of the first block:

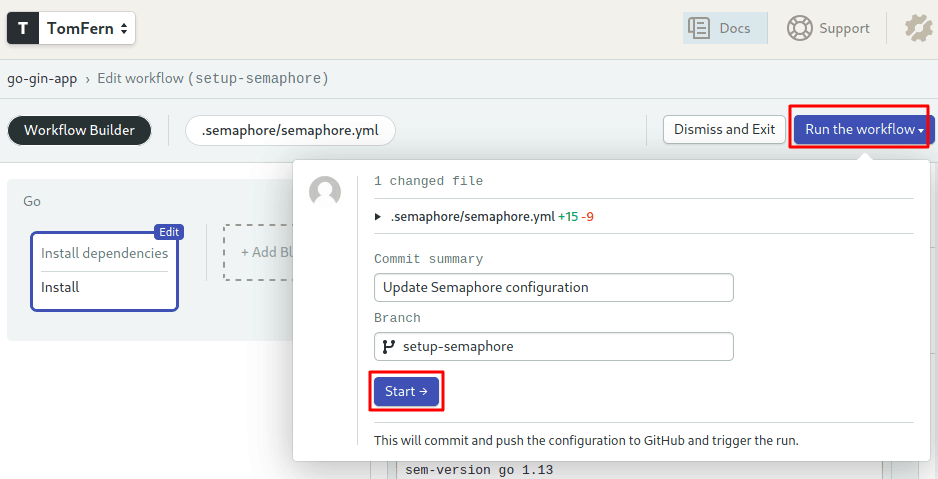
* Click on the first block and change its name to: “Install dependencies”.
* Below you’ll find the job, change its name to “Install” and type the following commands in the box:

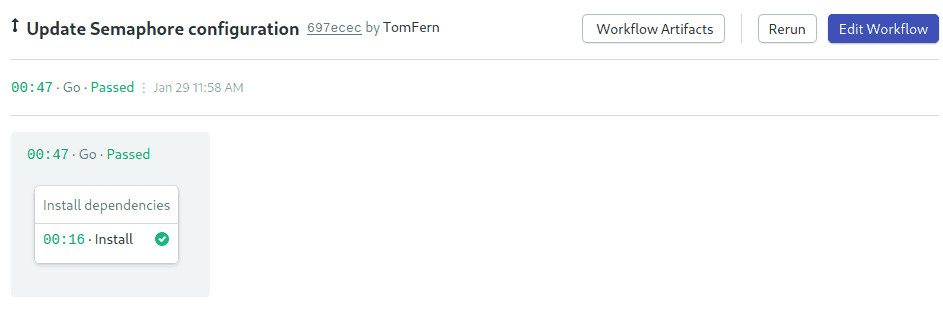
|  |
| --- |
| sem-version go 1.13  checkout  cache restore vendor-$SEMAPHORE\_GIT\_BRANCH-$(checksum go.mod),vendor-$SEMAPHORE\_GIT\_BRANCH,vendor-master  go mod vendor  cache store vendor-$SEMAPHORE\_GIT\_BRANCH-$(checksum go.mod),vendor-$SEMAPHORE\_GIT\_BRANCH,vendor-master vendor |

* Open the Environment Variables section. Create two variables using +Add env\_vars. These variables tell Go to use enable the modules feature and to store dependencies on the vendor/ directory instead of in your $HOME:
  + GO111MODULE = on
  + GOFLAGS = -mod=vendor



* Click on Run the Workflow on the top-right corner and then Start:





We’ve modified the block so it only downloads the Go dependencies:

* sem-version: a Semaphore built-in command to manage programming language versions. Semaphore supports most Go versions.
* checkout: another built-in command, checkout clones the repository and changes the current directory.
* go mod vendor: this a Go command that downloads the dependencies into the vendor directory, so they can be cached.
* cache: the cache commands provides read and write access to Semaphore’s cache, a project-wide storage for the jobs. Cache is usually able to figure out which files to stores, but we can use the following syntax to force a particular directory or file:

|  |
| --- |
| # store in the cache  cache store KEY1,KEY2,KEY3 DIRECTORY  # restore files and directories  cache restore KEY1,KEY2,KEY3 |

The first time that the jobs run, Go will download the dependencies and Semaphore will store them in the cache. For all the following runs, Semaphore will restore them and Go won’t need to download them again, thus speeding up the process considerably.

Testing with Semaphore

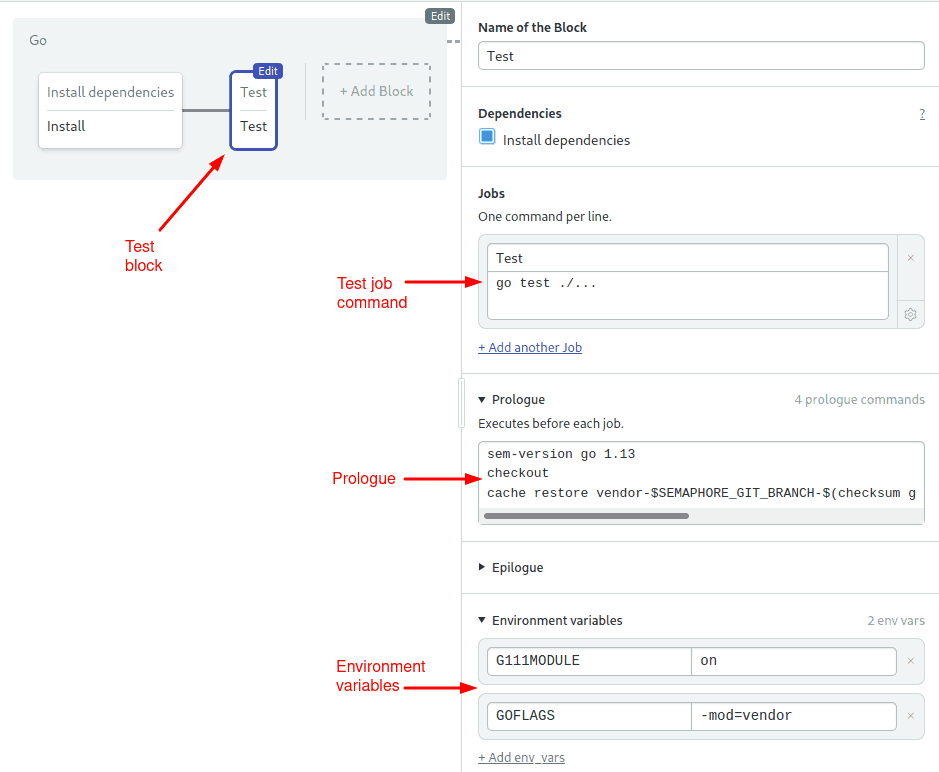
We expect our CI pipeline to be able to test the project and build a binary. We’ll add two more blocks for that:

* Click on Edit Workflow
* Use the +Add Block dotted line button to create a new block. Name the block “Test”
* Open the Prologue section and type the following commands. The commands are executed before all jobs in the block:

|  |
| --- |
| sem-version go 1.13  checkout  cache restore vendor-$SEMAPHORE\_GIT\_BRANCH-$(checksum go.mod),vendor-$SEMAPHORE\_GIT\_BRANCH,vendor-master  go mod vendor |

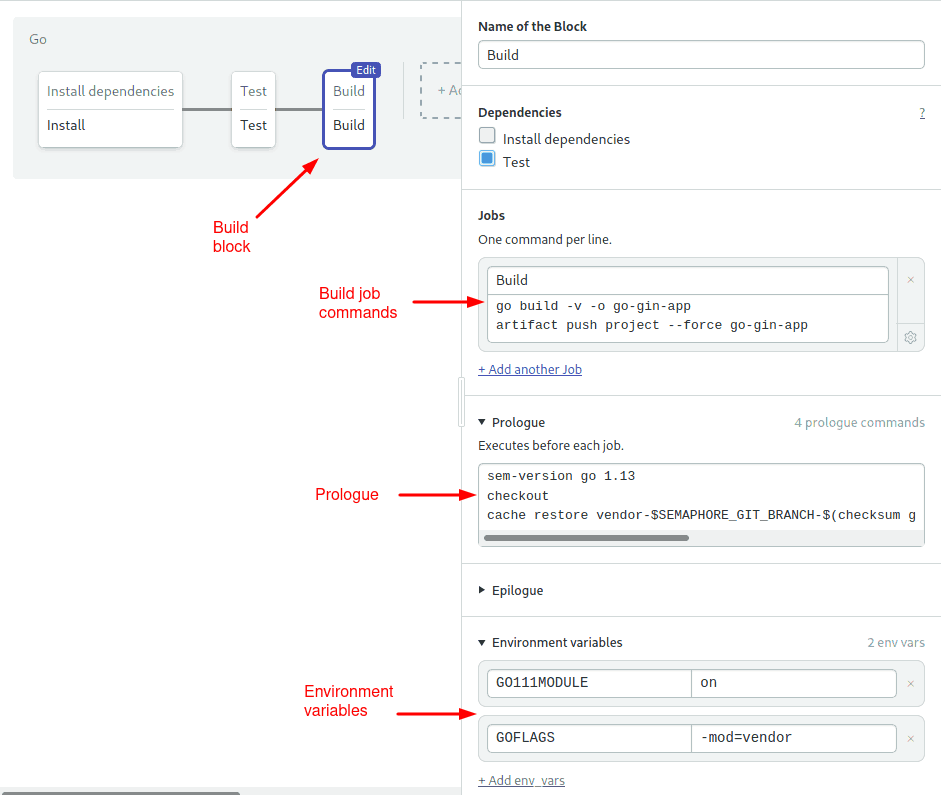
* Open the Environment Variables section and create the same variables as we did before: GO111MODULE and GOFLAGS.
* Set the name of the job to “Test” and type the following command:

|  |
| --- |
| go test ./... |



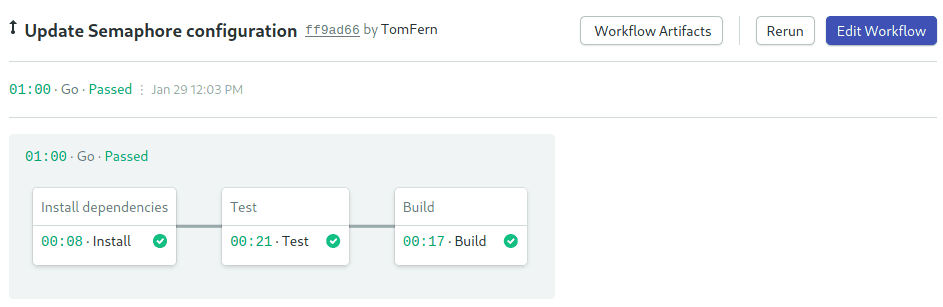
* Add the third block, let’s call it “Build”
* Repeat the Prologue and Environment Variables steps as before.
* Set the name of the job to “Build”
* Type the following commands in the box:

|  |
| --- |
| go build -v -o go-gin-app  artifact push project --force go-gin-app |

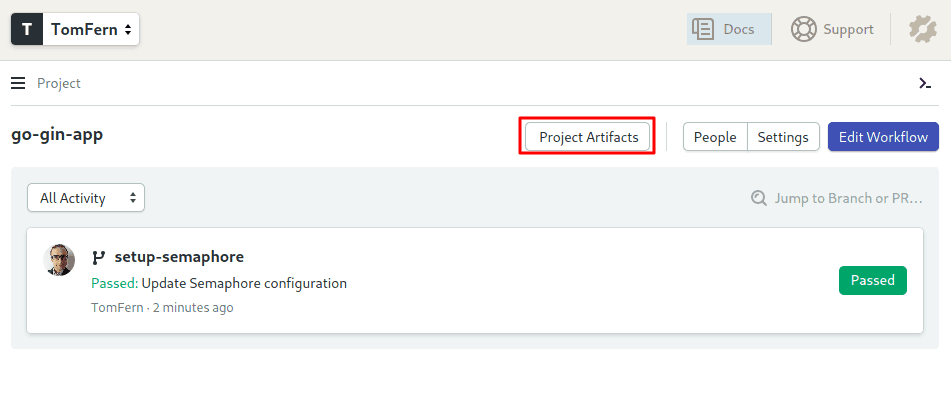


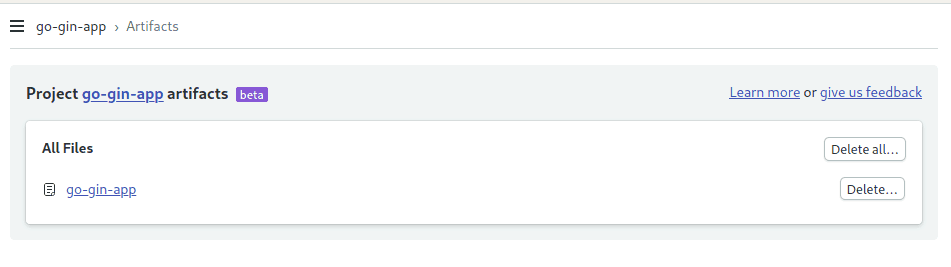
* Click on Run the Workflow and then Start.

The updated pipeline is now complete:



The artifact command we used in the build job uploads the binary into the project’s storage. To access it, use the Project Artifact button:





Semaphore has three separate artifact stores: Job, Workflow, and Project. For more information, check the artifacts doc.

Conclusion

In this tutorial, we created a new web application using Gin and gradually added more functionality. We used tests to build robust route handlers and saw how we can reuse the same code to render a response in multiple formats with minimal effort.

The code for the entire application is available in this Github repository.

Gin is easy to get started with—coupled with Go’s built-in functionality, its features make building high quality, well-tested web applications and microservices a breeze. If you have any questions or comments, feel free to post them below.

結論

在本教程中，我們使用 Gin 創建了一個新的 Web 應用程序，並逐漸添加了更多功能。我們使用測試來構建健壯的路由處理程序，並瞭解如何以最小的努力重複使用相同的代碼以多種格式呈現響應。

所有應用程序的代碼：https://github.com/demo-apps/go-gin-app

Gin 很容易入門 - 與 Go 的內置功能相結合，其功能使構建高質量，經過充分測試的 Web 應用程序和微服務變得輕而易舉。如果您有任何問題或意見，請隨時在下面發佈。