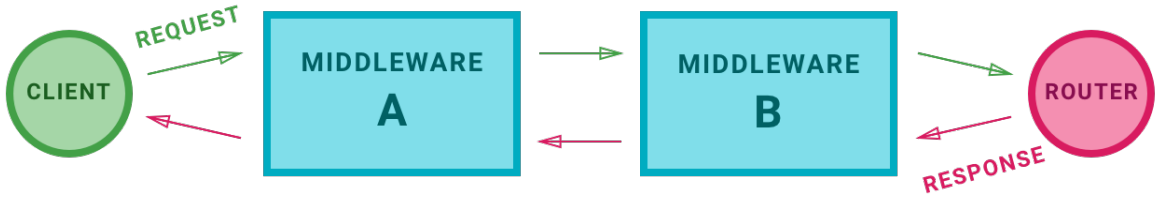
Chapter 28: Middleware

By Tanner Nelson

在构建应用程序的过程中，您经常会发现有必要将自己的步骤措施集成到请求管道中。实现此目的的最常见机制是使用一个或多个中间件。它们允许您执行以下操作：

- 记录传入的请求。
- 捕获错误并显示消息。
- 特定路由的流量速率限制。

中间件实例位于路由器和连接到服务器的客户端之间。这允许它们可以查看传入的请求，在到达您的控制器之前，并可能会发生改变。中间件实例可以选择通过生成自己的响应来提前返回，或者它可以将请求转发给链中的下一个响应者。最终的响应者总是你的路由器。当生成下一个响应者的响应时，中间件可以进行它认为必要的任何修改，或者选择将其原样转发回客户端。这意味着每个中间件实例都可以控制传入请求和传出响应。



正如您在上图中所看到的，应用程序中的第一个中间件实例 - 中间件A - 首先接收来自客户端的传入请求。然后，第一个中间件可以选择将此请求传递给下一个中间件 - 中间件B - 依此类推。最终，某些组件会生成响应，然后以相反的方向遍历中间件。请注意，这意味着第一个中间件最后会收到响应。

中间件协议相当简单，应该可以帮助您更好地理解上图：

```
public protocol Middleware {
  func respond(
    to request: Request,
    chainingTo next: Responder) throws -> Future<Response>
}
```

对于中间件A，request是来自客户端的传入数据，而next是中间件B。中间件A返回的异步响应则直接去到客户端。

对于中间件B，request是从中间件A传递来的请求。next是路由器。 中间件B返回的future响应则去到中间件 A.

Vapor的中间件

Vapor包含一些现成的中间件。本节将向您介绍可用的选项，以便您了解常用的中间件。

Error中间件

Vapor中最常用的中间件是ErrorMiddleware。它负责将同步或异步Swift错误转换为HTTP响应。未捕获的错误导致HTTP服务器立即关闭连接并打印内部错误日志。使用ErrorMiddleware可确保将您抛出的所有错误呈现为适当的HTTP响应。

在生产模式下，ErrorMiddleware将所有错误转换为不透明的500个内部服务器错误响应。这对于保持应用程序的安全性非常重要，因为错误可能包含敏感信息。您可以选择通过将错误类型遵循AbortError协议来提供不同的错误响应，从而允许您指定HTTP状态代码和错误消息。您也可以使用Abort，一种遵循AbortError协议的具体错误类型。例如：

```
throw Abort(.badRequest, "Something's not quite right.")
```

File中间件

另一种常见的中间件类型是FileMiddleware。此中间件提供应用程序目录中Public文件夹中的文件。当您使用Vapor创建可能需要静态文件（如图像或样式表）的前端网站时，这非常有用。

其他中间件

Vapor还提供了SessionsMiddleware，负责跟踪客户端连接的会话。其他软件包可能提供中间件以帮助它们集成到您的应用程序中。例如，Vapor的Authentication验证包中包含使用基本密码，简单的bearer tokens甚至JWT（JSON Web Tokens）保护路由的中间件。

Example: Todo API

现在您已经了解了各种类型的中间件的功能，您已经准备好了解如何配置它们以及如何创建自己的自定义中间件类型。

为此，您将实现基本的Todo列表API。此API有三个路由：

```
$ swift run Run routes
+--------+--------------+
| GET    | /todos       |
+--------+--------------+
| POST   | /todos       |
+--------+--------------+
| DELETE | /todos/:todo |
+--------+--------------+
```

您将为此项目创建和配置两种不同的中间件类型：

1. LogMiddleware：记录传入请求的响应时间。
2. SecretMiddleware：通过要求一个密钥来防止未经许可访问私有路由。

Log middleware

您要创建的第一个中间件将记录传入的请求。它将显示每个请求的以下信息：

- 请求方法
- 请求路径
- 响应状态
- 生成响应需要多长时间

在Terminal中打开starter项目目录，并输入以下命令为其生成Xcode项目：

```
vapor xcode -y
```

Xcode打开后，导航到**Middleware/LogMiddleware.swift**。在那里你会找到一个空的LogMiddleware类。

暂时忽略TimeInterval扩展；你以后会用到的。

首先使LogMiddleware遵循Middleware协议。只有一个方法要求：respond(to:chainingTo:)。

目前，中间件只会记录传入请求的描述。用以下内容替换LogMiddleware：

```
final class LogMiddleware: Middleware {
  // 1
  let logger: Logger

  init(logger: Logger) {
    self.logger = logger
  }

  // 2
  func respond(
    to req: Request,
    chainingTo next: Responder) throws -> Future<Response> {
    // 3
    logger.info(req.description)
    // 4
    return try next.respond(to: req)
  }
}

// 5
extension LogMiddleware: ServiceType {
  static func makeService(
    for container: Container) throws -> LogMiddleware {
```

```
    // 6
    return try .init(logger: container.make())
  }
}
```

这是您刚刚添加的代码的分析：

1. 创建一个存储属性来保存Logger。
2. 实现Middleware协议要求。
3. 将请求的描述作为信息日志发送到Logger。
4. 将传入的请求转发给下一个响应者。
5. 允许LogMiddleware在您的应用程序中注册为服务。
6. 初始化LogMiddleware实例，使用容器创建必要的Logger。

现在您已经创建了自定义中间件，您需要将其注册到您的应用程序。打开**configure.swift**并在 // register custom service types here: 下面添加以下行：

```
services.register(LogMiddleware.self)
```

注册LogMiddleware后，您可以使用MiddlewareConfig进行集成。接下来，在var middleware = MiddlewareConfig()下面添加以下行：

```
middleware.use(LogMiddleware.self)
```

这样可以全局启用LogMiddleware。这里的顺序也很重要：由于LogMiddleware是在ErrorMiddleware之前添加的，它首先接收请求，最后接收响应。这确保LogMiddleware可以记录未经其他中间件修改的客户端原始请求，和发向客户端之前的最终响应。

最后，构建并运行您的应用程序，然后使用curl向**GET/todos**发出请求：

```
curl localhost:8080/todos
```

查看正在运行的应用程序的日志输出。你会看到类似的东西：

```
[ INFO ] GET /todos HTTP/1.1
Host: localhost:8080
User-Agent: curl/7.54.0
Accept: */*
<no body> (LogMiddleware.swift:15)
```

这是一个很好的开始！但是您还可以改进LogMiddleware以提供更加有用和可读的输出。 打开**LogMiddleware.swift**并使用以下方法替换respond(to:chainingTo:)的实现：

```
func respond(
  to req: Request,
  chainingTo next: Responder) throws -> Future<Response> {
  // 1
  let start = Date()
  return try next.respond(to: req).map { res in
    // 2
    self.log(res, start: start, for: req)
    return res
  }
}

// 3
func log(_ res: Response, start: Date, for req: Request) {
  let reqInfo = "\(req.http.method.string) \(req.http.url.path)"
  let resInfo = "\(res.http.status.code) " +
    "\(res.http.status.reasonPhrase)"
  // 4
  let time = Date()
    .timeIntervalSince(start)
    .readableMilliseconds
  // 5
  logger.info("\(reqInfo) -> \(resInfo) [\(time)]")
}
```

以下是新方法如何工作的分析：

1. 首先，创建一个开始时间。在完成任何额外工作之前执行此操作以获得最准确的响应时间测量。

2. 不是直接返回响应，map future结果，以便您可以访问Response对象。将此传递给log(_:start:for:)。

3. 此方法使用响应开始日期记录传入请求的响应。

4. 使用timeIntervalSince(_:)和文件底部TimeInterval的扩展生成可读时间。

5. 记录信息字符串

现在您已经更新了LogMiddleware，再次构建并运行，然后curl **GET/todos**。

```
curl localhost:8080/todos
```

如果检查应用程序的输出，您将看到一种新的，更简洁的输出格式。

```
[ INFO ] GET /todos -> 200 OK [1.9ms] (LogMiddleware.swift:32)
```

Secret middleware

既然您已经学会了如何创建中间件并在全局范围内应用它，那么您将学习如何将中间件应用于特定路由。

两个Todo List API路由可以对数据库进行更改：

- **POST /todos**
- **DELETE /todos/:id**

如果这是一个公共API，您需要使用中间件用密钥来保护这些路由。这正是SecretMiddleware将要做的。

打开**Middleware/SecretMiddleware.swift**并使用以下代码替换SecretMiddleware的类定义：

```
final class SecretMiddleware: Middleware {
  // 1
  let secret: String

  init(secret: String) {
    self.secret = secret
  }

  // 2
  func respond(
    to request: Request,
    chainingTo next: Responder) throws -> Future<Response> {
    // 3
    guard
      request.http.headers.firstValue(name: .xSecret) == secret
    else {
      // 4
      throw Abort(
        .unauthorized,
        reason: "Incorrect X-Secret header.")
    }
    // 5
    return try next.respond(to: request)
  }
}
```

以下是SecretMiddleware如何工作的分析：

1. 创建一个存储属性以保存密钥。
2. 实现Middleware协议要求。
3. 根据配置的密钥检查传入请求中的X-Secret标头。
4. 如果标头值不匹配，则抛出未经授权的HTTP状态的错误。
5. 如果标头匹配，则正常传递给链中的下一个中间件。

现在，您只需要将SecretMiddleware遵循ServiceType协议，以便它可以在您的应用程序中用作服务。

在SecretMiddleware实现之后添加以下代码。

```
extension SecretMiddleware: ServiceType {
  static func makeService(
    for worker: Container) throws -> SecretMiddleware {
    // 1
    let secret: String
    switch worker.environment {
    // 2
    case .development: secret = "foo"
    default:
      // 3
      guard let envSecret = Environment.get("SECRET") else {
        let reason = """
          No $SECRET set on environment. \
          Use "export SECRET=<secret>"
          """
        throw Abort(
          .internalServerError,
          reason: reason)
      }
      secret = envSecret
    }
    // 4
    return SecretMiddleware(secret: secret)
  }
}
```

以下是此代码的工作分析：

1. 创建一个本地变量来存储配置的密钥。
2. 如果当前是开发环境，只需使用“**foo**”作为密钥。
3. 如果当前不是开发环境，则尝试从环境变量**$SECRET**键中获取密钥。

4. 使用配置的密钥初始化SecretMiddleware的实例。

是时候注册新的中间件了。打开**configure.swift**并在注释// register custom service types下面添加以下内容。

```
services.register(SecretMiddleware.self)
```

您已经创建并注册了SecretMiddleware，现在您可以使用它来保护所需的路由了。打开**routes.swift**并使用以下代码替换**POST**和**DELETE**路由：

```
// 1
router.group(SecretMiddleware.self) { secretGroup in
  // 2
  secretGroup.post("todos", use: todoController.create)
  secretGroup.delete(
    "todos",
    Todo.parameter,
    use: todoController.delete)
}
```

这是它的作用：

1. 创建一个由SecretMiddleware包装的新路由组。

2. 在新创建的路由组中注册**POST**和**DELETE**路由，而不是在全局路由器中。

构建并运行应用程序，然后在RESTed中创建新请求。配置请求如下：

- **URL**: http://localhost:8080/todos
- **method**: POST

添加名称和值的参数：

- **title**: This is a test TODO!

单击发送请求并注意响应：

```
{
    "error": true,
    "reason": "Incorrect X-Secret header."
}
```

中间件正在保护路由！如果您尝试查询**GET/todos**，您会发现它仍然有效。

将**X-Secret: foo**添加到RESTed中的headers部分并再次发送请求。现在你会注意到响应已经改变了。中间件允许此请求通过控制器，因为现在它具有适当的标头了。

然后去哪儿?

中间件对于创建大型Web应用程序非常有用。它允许您全局的应用限制和变换，或仅应用于少数不相关的路由、可重用的组件。在本章中，您学习了如何创建一个全局LogMiddleware，它显示有关应用程序的所有传入请求的信息。然后，您创建了SecretMiddleware，它可以保护选定路由免受公共访问。

有关使用中间件的更多信息，请务必查看Vapor的API文档：

- `Middleware`: https://api.vapor.codes/vapor/latest/Vapor/Protocols/Middleware.html
- `MiddlewareConfig`: https://api.vapor.codes/vapor/latest/Vapor/Structs/MiddlewareConfig.html