Chapter 14: Templating with Leaf

By Tim Condon

在本书的前一部分中，您学习了如何使用Vapor和Fluent创建API。然后，您学习了如何创建iOS客户端以使用API。在本节中，您将创建另一个客户端 - 一个网站。您将看到如何使用Leaf在Vapor应用程序中创建动态网站。

Leaf

Leaf是Vapor的模板语言。模板语言允许您将信息传递给页面，这样它就可以生成最终的HTML，而不必预先知道所有信息。例如，在TIL应用程序中，您不知道用户在部署应用程序时将创建的每个缩略词。模板化将让您轻松处理。

模板语言还允许您减少网页中的重复内容。您可以创建单个模板并设置明确显示特定缩略词的属性，而不是缩略词的多个页面。如果您决定更改显示缩略词的方式，则只需在一个位置更改代码，所有缩略词页面将显示新格式。

最后，模板语言允许您将模板嵌入到其他模板中。例如，如果您的网站上有导航，则可以创建一个生成导航代码的模板。您将导航模板嵌入到需要导航的所有模板中，而不是复制代码。

配置Leaf

要使用Leaf，您需要将其作为依赖项添加到项目中。使用第11章“Testing”中的TIL应用程序或本章的入门项目，打开**Package.swift**，用以下内容替换其内容：

```
// swift-tools-version:4.0
import PackageDescription

let package = Package(
  name: "TILApp",
  dependencies: [
    .package(
      url: "https://github.com/vapor/vapor.git",
      from: "3.0.0"),
    .package(
      url: "https://github.com/vapor/fluent-postgresql.git",
      from: "1.0.0"),
    .package(
      url: "https://github.com/vapor/leaf.git",
      from: "3.0.0")
  ],
  targets: [
    .target(name: "App",
            dependencies: ["FluentPostgreSQL",
                           "Vapor",
                           "Leaf"]),
    .target(name: "Run", dependencies: ["App"]),
    .testTarget(name: "AppTests", dependencies: ["App"]),
  ]
)
```

所做的更改是：

- 使TILApp package依赖于Leaf package。
- 使App target依赖于Leaf target以确保其正确链接。

默认情况下，Leaf希望模板位于**Resources/Views**目录中。在“终端”中，键入以下内容以创建这些目录：

```
mkdir -p Resources/Views
```

最后，您必须为网站创建新路由。创建一个新控制器以包含这些路由。在终端中，键入以下内容：

```
touch Sources/App/Controllers/WebsiteController.swift
```

配置完所有内容后，重新生成Xcode项目以开始使用Leaf。在终端中，键入以下内容：

```
vapor xcode -y
```

渲染页面

打开**WebsiteController.swift**，创建一个新类型来保存所有网站路由和返回index模板的路由：

```
import Vapor
import Leaf

// 1
struct WebsiteController: RouteCollection {
  // 2
  func boot(router: Router) throws {
    // 3
    router.get(use: indexHandler)
  }

  // 4
  func indexHandler(_ req: Request) throws -> Future<View> {
    // 5
    return try req.view().render("index")
  }
}
```

这是它的作用：

1. 声明遵循RouteCollection协议的新的WebsiteController类型。
2. 根据RouteCollection的要求实现boot(router:)。
3. 注册indexHandler(_:)以处理对路由器根路径的GET请求，即对 / 的请求。
4. 实现返回Future<View>的indexHandler(_:)。
5. 渲染**index**模板并返回结果。稍后您将了解req.view()。

Leaf从**Resources/Views**目录中用名为**index.leaf**的模板生成了一个页面。

请注意，render(_:)调用不需要文件扩展名。创建此文件并插入以下内容：

```
<!DOCTYPE html>
#// 1
<html lang="en">
<head>
  <meta charset="utf-8" />
  #// 2
  <title>Hello World</title>
</head>
<body>
  #// 3
  <h1>Hello World</h1>
</body>
</html>
```

这个文件的作用是：

1. 使用<head>和<body>声明基本的HTML 5页面。
2. 将页面标题设置为**Hello World** - 这是浏览器标签页中显示的标题。
3. 将正文设置为一个单独的**Hello World**标题。

> 注意：您可以使用您选择的任何文本编辑器创建**.leaf**文件，包括Xcode。如果使用Xcode，请选择**Editor ▸ Syntax Coloring ▸ HTML**以便正确突出显示元素和缩进支持。

您必须注册新建的WebsiteController控制器。打开**routes.swift**并将以下内容添加到routes(_:)末尾：

```
let websiteController = WebsiteController()
try router.register(collection: websiteController)
```

接下来，您必须注册Leaf服务。打开**configure.swift**并将以下内容添加到import Vapor下面的imports部分：

```
import Leaf
```

接着，在try services.register(FluentPostgreSQLProvider())之后，添加以下内容：

```
try services.register(LeafProvider())
```

使用常规的req.view()来获取渲染器可以让您轻松切换到不同的模板引擎。虽然这在运行应用程序时可能不太有用，但它对于测试却非常有用。

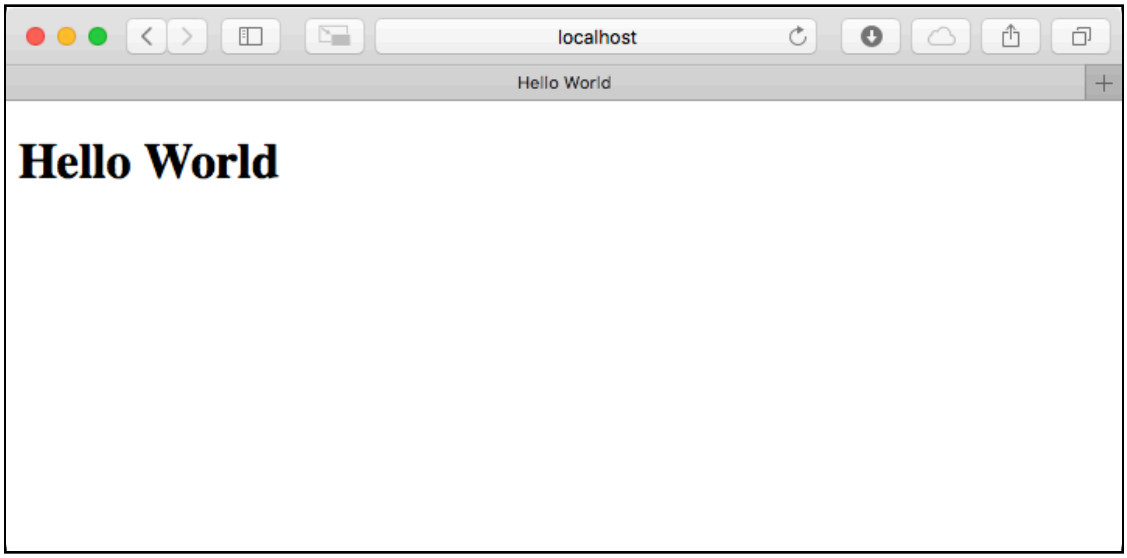
例如，它允许您使用测试渲染器生成纯文本以进行验证，而不是在测试用例中解析HTML输出。

req.view()要求Vapor提供遵循ViewRenderer协议的类型。 TemplateKit - Leaf构建的模块 - 提供PlaintextRenderer，Leaf提供LeafRenderer。在**configure.swift**中，将以下内容添加到configure(_:_:_:)的末尾：

```
config.prefer(LeafRenderer.self, for: ViewRenderer.self)
```

这告诉Vapor当要求ViewRenderer类型时就使用LeafRenderer。

构建并运行应用程序，记住选择**Run**方案，然后打开浏览器。输入URL **http:// localhost:8080**，您将收到从模板生成的页面：



注入变量

该模板目前只是一个静态页面，并不令人印象深刻！要使页面更具动态性，打开**index.leaf**并将<title>行更改为以下内容：

```
<title>#(title) | Acronyms</title>
```

这是使用Leaf的#( )函数提取名为title的参数。像许多Vapor一样，Leaf使用Codable来处理数据。

在**WebsiteController.swift**的底部，添加以下内容，以创建一个包含标题的新类型：

```
struct IndexContext: Encodable {
  let title: String
}
```

由于数据仅流向Leaf，因此您只需要遵循Encodable协议。IndexContext是视图的数据，类似于MVVM设计模式中的视图模型。接下来，更改indexHandler(_:)以将IndexContext传递给模板。用以下内容替换实现：

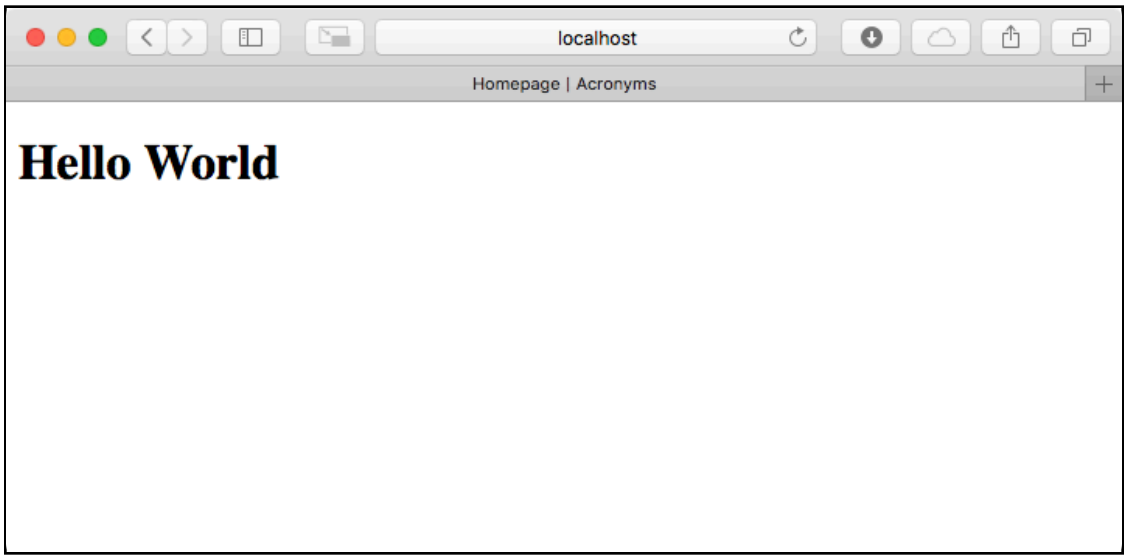
```
func indexHandler(_ req: Request) throws -> Future<View> {
  // 1
  let context = IndexContext(title: "Home page")
  // 2
  return try req.view().render("index", context)
}
```

这是新代码的作用：

1. 创建包含所需标题的IndexContext实例。

2. 将context传递给Leaf作为render(_:_:)函数的第二个参数。

构建并运行，然后在浏览器中刷新页面。你会看到更新的标题：



使用标签

TIL网站的主页应显示所有缩略词的列表。依然在**WebsiteController.swift**中，在IndexContext类型里的title下面添加一个新属性：

```
let acronyms: [Acronym]?
```

这是一个可选的缩略词数组;它的值可以是nil，因为数据库中可能没有缩略词。接下来，更改indexHandler(_:)以获取所有缩略词并将它们插入IndexContext中。

使用以下内容再次替换实现：

```
func indexHandler(_ req: Request) throws -> Future<View> {
  // 1
  return Acronym.query(on: req)
    .all()
    .flatMap(to: View.self) { acronyms in
      // 2
      let acronymsData = acronyms.isEmpty ? nil : acronyms
      let context = IndexContext(
        title: "Home page",
        acronyms: acronymsData)
      return try req.view().render("index", context)
  }
}
```

这是它的作用：

1. 使用Fluent查询从数据库中获取所有缩略词。

2. 如果有缩略词的话，将它们赋值到IndexContext中，否则将变量设置为nil。 Leaf可以在模板中检查nil。

最后打开**index.leaf**并将<body>标签之间的部分代码更改为以下内容：

```
#// 1
<h1>Acronyms</h1>

#// 2
#if(acronyms) {
  #// 3
  <table>
    <thead>
      <tr>
        <th>Short</th>
        <th>Long</th>
      </tr>
    </thead>
    <tbody>
      #// 4
      #for(acronym in acronyms) {
```

```
          <tr>
            #// 5
            <td>#(acronym.short)</td>
            <td>#(acronym.long)</td>
          </tr>
        }
      </tbody>
    </table>
  #// 6
  } else {
    <h2>There aren't any acronyms yet!</h2>
  }
```

这是新代码的作用：

1. 声明一个“Acronyms”的新标题。

2. 使用Leaf的#if( )标签查看是否设置了acronyms变量。 #if( )可以验证变量的可空性， 对布尔值也有效，甚至可以计算表达式。

3. 如果设置了acronyms变量，则创建一个HTML表。该表有一个标题行 - <thead> - 有两列，**Short**和**Long**。

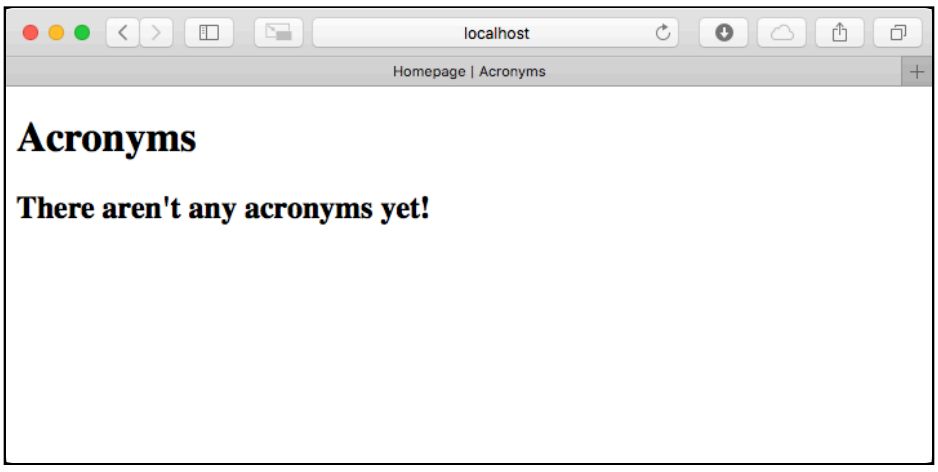
4. 使用Leaf的#for( )标签循环遍历所有缩略词。这与Swift的for循环类似。

5. 为每个缩略词创建一行。使用Leaf的#( )函数来提取变量。由于所有内容都是Encodable，您可以使用点符号来访问缩略词的属性，就像Swift一样！

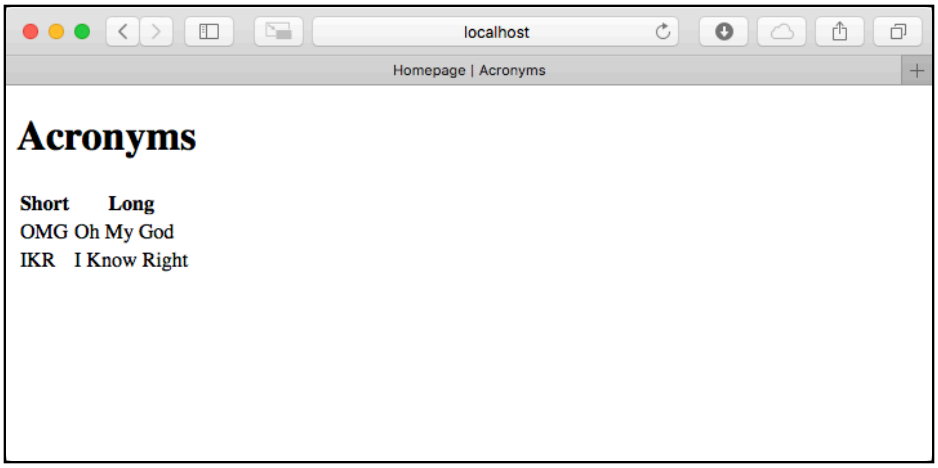
6. 如果没有缩略词，请打印一条合适的信息。

构建并运行，然后在浏览器中刷新页面。

如果数据库中没有缩略词，您将看到正确的消息：



如果数据库中有缩略词，您将在表格中看到它们：



缩略词详细页面

现在，您需要一个页面来显示每个缩略词的详细信息。在**WebsiteController.swift**的末尾，创建一个新类型来保存此页面的context：

```
struct AcronymContext: Encodable {
  let title: String
  let acronym: Acronym
  let user: User
}
```

此AcronymContext包含页面标题，缩略词本身以及创建该缩略词的用户。在indexHandler(_:)下面为缩略词详细信息页面创建以下路由处理程序：

```
// 1
func acronymHandler(_ req: Request) throws -> Future<View> {
  // 2
  return try req.parameters.next(Acronym.self)
    .flatMap(to: View.self) { acronym in
      // 3
      return acronym.user
        .get(on: req)
        .flatMap(to: View.self) { user in
          // 4
          let context = AcronymContext(
            title: acronym.short,
            acronym: acronym,
            user: user)
          return try req.view().render("acronym", context)
      }
  }
}
```

以下是此路由处理程序的作用：

1. 声明一个新的路由处理程序acronymHandler(_:)，并返回Future <View>。
2. 从请求的parameters中提取缩略词并解包结果。
3. 获取创建缩略词的用户并解包结果。
4. 创建包含相应详细信息的AcronymContext实例，并使用**acronym.leaf**模板渲染页面。

最后在boot(router:)底部注册路由：

```
router.get("acronyms", Acronym.parameter, use: acronymHandler)
```

这会为**/acronyms/<ACRONYM ID>**注册acronymHandler路由，类似于API。在**Resources/Views**目录中创建**acronym.leaf**模板文件，打开新文件并添加以下内容：

```
<!DOCTYPE html>
#// 1
<html lang="en">
<head>
  <meta charset="utf-8" />
  #// 2
  <title>#(title) | Acronyms</title>
</head>
<body>
  #// 3
  <h1>#(acronym.short)</h1>
  #// 4
  <h2>#(acronym.long)</h2>

  #// 5
  <p>Created by #(user.name)</p>
</body>
</html>
```

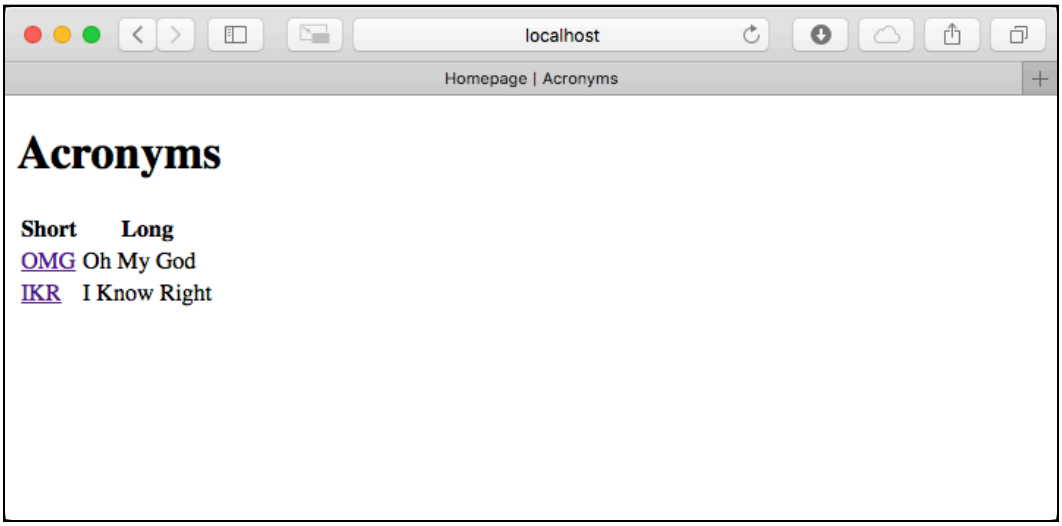
这是这个模板的作用：

1. 声明一个类似**index.leaf**的HTML5页面。
2. 将标题设置为传入的值。
3. 在<h1>标题中打印缩略词的short属性。
4. 在<h2>标题中打印缩略词的long属性。
5. 在<p>块中打印缩略词的用户。

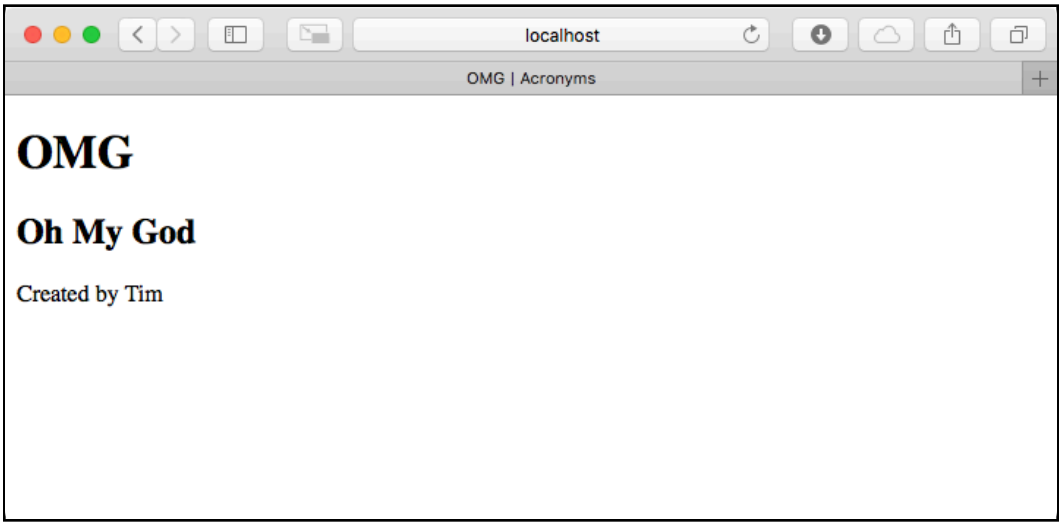
最后，更改**index.leaf**，以便您可以导航到该页面。用以下代码替换表格中的第一列（<td>#(acronym.short)</td>）

```
<td><a href="/acronyms/#(acronym.id)">#(acronym.short)</a></td>
```

这将缩略词的short属性包装在HTML的<a>标签中，这是一个链接。该链接将每个缩略词的URL设置为上面注册的路由。构建并运行，然后在浏览器中刷新页面：



你会看到每个缩略词的short形式现在都是一个链接。单击链接，浏览器将导航到缩略词的详细页面：



然后去哪儿?

本章介绍了Leaf并向您展示了如何开始构建动态网站。本节的下一章将向您展示如何将模板嵌入到其他模板中，以及美化您的应用程序并从网站创建缩略词。