2. 添加页面

本指南的任务是为我们的Phoenix项目添加两个新页面。一个是纯静态的页面，另一个将取URL路径的一部分作为输入，并将其传递到模板以供显示。在此过程中，我们将熟悉Phoenix项目的基本组件：路由、控制器、视图和模板。

当Phoenix为我们生成一个新的应用程序时，它会构建一个像这样的顶级目录结构：

```
├── _build
├── assets
├── config
├── deps
├── lib
│   └── hello
│   └── hello_web
│   └── hello.ex
│   └── hello_web.ex
├── priv
├── test
```

本指南中的大部分工作都在 `lib/hello_web` 目录中，该目录保存应用程序中与web相关的部分。展开后如下：

```
├── channels
│   └── user_socket.ex
├── controllers
│   └── page_controller.ex
├── templates
│   ├── layout
│   │   └── app.html.eex
│   └── page
│       └── index.html.eex
└── views
    ├── error_helpers.ex
    ├── error_view.ex
    ├── layout_view.ex
    └── page_view.ex
├── endpoint.ex
```

```
├── gettext.ex
├── router.ex
```

当前在控制器、模板和视图目录中的所有文件都用于创建我们在上一篇指南中看到的“Welcome to Phoenix!”页面。稍后我们将看到如何重用其中一些代码。在开发环境中运行时，代码更改将在新的web请求时自动重新编译。

我们应用程序的所有静态资源，如js、css和图像文件都存在于 `assets` 文件夹中，这些资源由webpack或其他前端构建工具构建到 `priv/static` 中。我们暂时不会在这里做任何修改，但是将来参考的时候知道在哪里查找是很好的。

```
├── assets
│   ├── css
│   │   └── app.css
│   ├── js
│   │   └── app.js
│   └── static
│   └── node_modules
│   └── vendor
```

还有一些与web无关的文件我们应该知道。我们的应用程序文件(启动Elixir应用程序及其管理树)位于 `lib/hello/application.ex` 。在 `lib/hello/repo` 中也有Ecto Repo用于与数据库交互。你可以在Ecto的指南中了解更多。

```
lib
├── hello
|   ├── application.ex
|   └── repo.ex
├── hello_web
|   ├── channels
|   ├── controllers
|   ├── templates
|   ├── views
|   ├── endpoint.ex
|   ├── gettext.ex
|   └── router.ex
```

我们的 `lib/hello_web` 目录包含与web相关的文件——路由、控制器、模板、通道等。我们的Elixir应用程序其余部分位于 `lib/hello` 中，这里的代码结构与其他Elixir应用程序一样。

准备足够了，让我们开始第一个Phoenix新页面吧！

新的路由（A New Route）

路由是将唯一的HTTP verb/path对映射到处理它们的controller/action对。Phoenix在新应用程序中为我们生成一个路由文件 `lib/hello_web/router.ex` 。该文件是我们将在这个小节工作的地方。

我们上一节 "Welcome to Phoenix!"页面的路由是这样的：

```
get "/", PageController, :index
```

让我们来消化一下这个路由要告诉我们的东西。访问http://localhost:4000/向根路径发出一个HTTP GET请求。所有像这样的请求都由 `HelloWeb.PageController` 模块的 `index` 函数处理，该模块被定义在 `lib/hello_web/controllers/page_controller.ex` 文件中。

当我们将浏览器指向http://localhost:4000/hello时，我们将要构建的页面将简单地显示“Hello World, from Phoenix!”

创建该页面首先要做的是为它定义一个路由。在文本编辑器打开 `lib/hello_web/router.ex` 。它目前应该是这样的：

```
defmodule HelloWeb.Router do
  use HelloWeb, :router

  pipeline :browser do
    plug :accepts, ["html"]
    plug :fetch_session
    plug :fetch_flash
    plug :protect_from_forgery
    plug :put_secure_browser_headers
  end

  pipeline :api do
    plug :accepts, ["json"]
```

```
  end

  scope "/", HelloWeb do
    pipe_through :browser

    get "/", PageController, :index
  end

  # Other scopes may use custom stacks.
  # scope "/api", HelloWeb do
  #   pipe_through :api
  # end
end
```

现在，我们将忽略pipelines和 `scope` 的作用，只关注添加路由。（如果你感兴趣，我们将在路由指南中介绍这些主题）

让我们向router中添加一条新路由，它将 `/hello` 的 `GET` 请求映射到即将创建的 `HelloWeb.HelloController` 的 `index` 动作。

```
get "/hello", HelloController, :index
```

`router.ex` 文件中的 `scope "/"` 块现在应该是这样的：

```
scope "/", HelloWeb do
  pipe_through :browser

  get "/", PageController, :index
  get "/hello", HelloController, :index
end
```

新的控制器（A New Controller）

Controllers是Elixir的modules，action是其定义的Elixir functions。actions的目的是收集任何数据并执行渲染所需的任何任务。我们的路由表明我们需要一个带有 `index/2` 动作的 `HelloWeb.HelloController` 模块。

为此，我们创建一个新的 `lib/hello_web/controllers/hello_controller.ex` 文件，并使它看起来像下面这样：

```
defmodule HelloWeb.HelloController do
  use HelloWeb, :controller

  def index(conn, _params) do
    render(conn, "index.html")
  end
end
```

我们将在Controllers Guide保留 `use HelloWeb, :controller` 的讨论。现在，让我们专注于 `index/2` 动作。

所有控制器动作都带有两个参数。第一个是 `conn` ，该结构保存有关请求的大量数据。第二个是 `params` ，这是请求参数。在这里，我们没有使用 `params` ，而是通过添加 `_` 来避免编译器警告。

该动作的核心是 `render(conn, "index.html")` 。这告诉Phoenix去查找一个名为 `index.html.eex` 的模版并渲染它。Phoenix将在我们控制器目录后的那个目录中查找模板，也就是 `lib/hello_web/templates/hello` 。

> 注意：在这里也可以使用原子作为模板名称， `render(conn, :index)` ，但是将根据Accept headers来选择模板，例如 `”index.html”` 或 `"index.json"` 。

负责渲染的模块是视图，接下来我们将新建一个。

新的视图（A New View）

Phoenix视图有几个重要的工作。它们渲染模板。它们还充当来自控制器的原始数据的表示层，为在模板中使用做好准备。执行此转换的函数应该放在视图中。

例如，假设我们有一个数据结构，该数据结构用一个 `first_name` 字段和一个 `last_name` 字段表示用户，并且在模板中，我们要显示用户的全名。我们可以在模板中编写代码以将这些字段合并为一个全名，但是更好的方法是在视图中编写一个函数为我们完成，然后在模板中调用该函

数。这样做的结果是得到一个更干净，更清晰的模板。

为了给我们的 `HelloController` 渲染任何模板，我们需要一个 `HelloView` 。名称在这里很重要—视图和控制器名称的第一部分必须匹配。现在，让我们创建一个空的视图，并留下更详细的视图描述以供以后使用。创建 `lib/hello_web/views/hello_view.ex` 并使其如下所示：

```
defmodule HelloWeb.HelloView do
  use HelloWeb, :view
end
```

新的模版（A New Template）

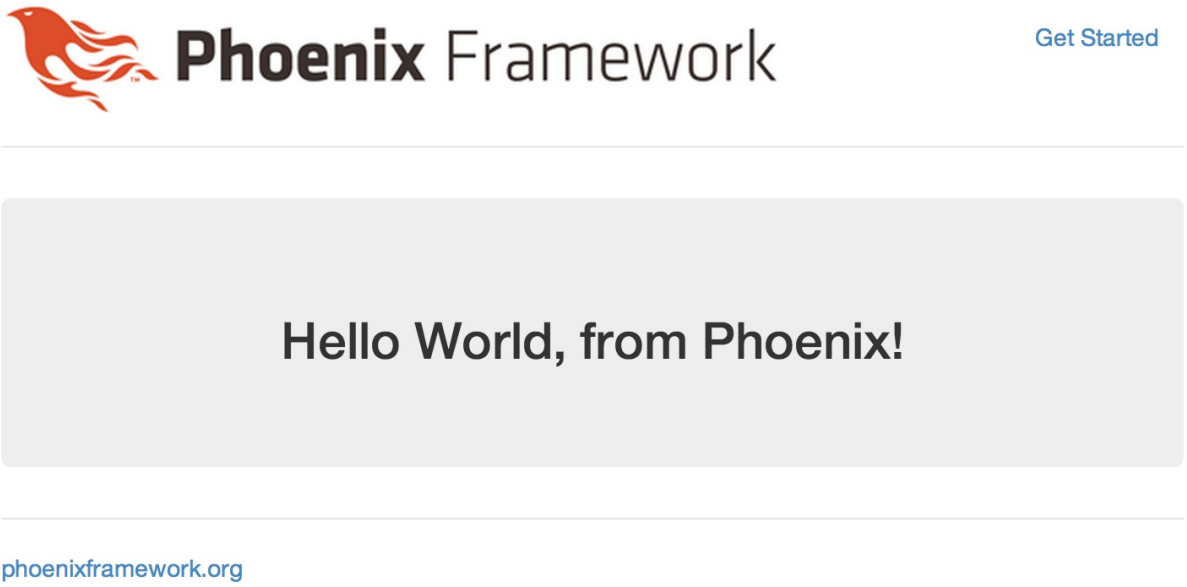
Phoenix模板就是可以把数据渲染其中的模版。Phoenix使用的标准模板引擎为 `EEx` （Embedded Elixir）。Phoenix增强了EEx，使其包括对值自动转义。这可以保护您免受跨站点脚本之类的安全漏洞的困扰，而无需您进行额外的工作。我们所有的模板文件都将具有 `.eex` 文件扩展名。

模板的作用域是视图，而视图的作用域是控制器。Phoenix创建了一个 `lib/hello_web/templates` 目录，我们可以在其中放置所有这些内容。最好为组织创建命名空间，因此对于我们的 `hello` 页面，这意味着我们需要在 `lib/hello_web/templates` 下创建 `hello` 目录，然后在里面创建 `index.html.eex` 文件。

现在开始吧。创建 `lib/hello_web/templates/hello/index.html.eex` 并使其如下所示：

```
<div class="phx-hero">
  <h2>Hello World, from Phoenix!</h2>
</div>
```

现在我们有了路由，控制器，视图和模板，我们应该能够将浏览器指向http://localhost:4000/hello并看到来自Phoenix的问候！（如果您在中途停止了服务器，则重新启动服务器的任务是 `mix phx.server` 。）



关于我们刚刚所做的事情，有几件有趣的事情要注意。进行这些更改后，我们无需停止并重新启动服务器。是的，Phoenix有代码热更新功能！同样，即使我们的 `index.html.eex` 文件仅包含一个div标签，但我们获得的页面都是完整的HTML文档。我们的index模板将被渲染到应用程序布局— `lib/hello_web/templates/layout/app.html.eex` 中。如果打开它，您将看到如下所示的一行：

```
<%= render @view_module, @view_template, assigns %>
```

这就是在HTML发送给浏览器之前将我们的模板渲染到布局中。

关于代码热更新的说明，某些编辑器的自动linters可能会阻止代码热更新工作。如果对您不起作用，请参阅这里的讨论。

另一个新页面（Another New Page）

让我们为应用程序增加一点复杂性。我们将添加一个新页面，该页面将识别一部分URL，将其标记为“messenger”，并将其通过控制器传递到模板中，以便我们的messenger可以打个招呼。

正如我们上次所做的那样，我们要做的第一件事就是创建一个新的路由。

新的路由（A New Route）

在本练习中，我们将重复使用刚创建的 `HelloController` 并添加一个新的 `show` 动作。我们将在最后一个路由下方添加一行，如下所示：

```
scope "/", HelloWeb do
  pipe_through :browser

  get "/", PageController, :index
  get "/hello", HelloController, :index
  get "/hello/:messenger", HelloController, :show
end
```

注意，我们将原子 `:messenger` 放置在路径中。Phoenix将采用URL中该位置出现的任何值，并将带有指向该值的键 `messenger` 以 `Map` 形式传递给控制器。

例如，如果将浏览器指向：http://localhost:4000/hello/Frank，则“:messenger”的值将为“Frank”。

新的动作（A New Action）

`HelloWeb.HelloController` `show` 动作将处理对我们新路由的请求。我们已经拥有控制器在 `lib/hello_web/controllers/hello_controller.ex` 中，因此我们所需要做的就是编辑该文件并向其中添加show动作。这次，我们需要在传递给动作的参数map中保留其中一项，以便我们可以将其（the messenger）传递给模板。为此，我们将以下show函数添加到控制器中：

```
def show(conn, %{"messenger" => messenger}) do
  render(conn, "show.html", messenger: messenger)
end
```

这里有几件事要注意。我们对传递给show函数的参数进行模式匹配，以便将 `messenger` 变量绑定到我们 `:messenger` 在URL中放置的值。例如，如果我们的URL是http://localhost:4000/hello/Frank，则messenger变量将绑定到 `Frank` 。

在该 `show` 动作的主体内，我们还将第三个参数传递给render函数，一个键/值对，其中 `:messenger` 是键， `messenger` 变量作为值传递。

> 注意：如果操作主体除了需要绑定的messenger变量之外，还需要访问绑定到params变量的完整参数映射，则可以这样定义 `show/2` ：

```
def show(conn, %{"messenger" => messenger} = params) do
  ...
end
```

最好记住，params映射的键将始终是字符串，并且等号不表示赋值，而是模式匹配断言。

新的模版（A New Template）

对于这个难题的最后一部分，我们需要一个新的模板。由于它是针对 `HelloController` 的 `show` 动作的，因此它将被放进 `lib/hello_web/templates/hello` 目录并被命名为 `show.html.eex` 。它看起来令人惊讶地类似于我们的 `index.html.eex` 模板，只是我们需要显示messenger的名称。

为此，我们将使用特殊的EEx标记执行Elixir表达式— `<%= %>` 。请注意，初始标签的等号如下： `<%=` 。这意味着将执行这些标签之间的所有Elixir代码，并且结果值将替换该标签。如果缺少等号，则仍将执行代码，但该值不会出现在页面上。

这是模板应该看起来的样子：

```
<div class="phx-hero">
  <h2>Hello World, from <%= @messenger %>!</h2>
</div>
```

我们的messenger显示为 `@messenger` 。在这种情况下，这不是模块属性。它是一种特殊的元编程语法，代表 `assign.messenger` 。这样的结果是在视觉上好得多，并且在模板中更容易使用。

大功告成。如果将浏览器指向此处：http:// localhost:4000/hello/Frank，您应该会看到一个如下所示的页面：

随便玩一下。无论您在 `/hello/` 后面放了什么它都会作为您的messenger出现在页面上。