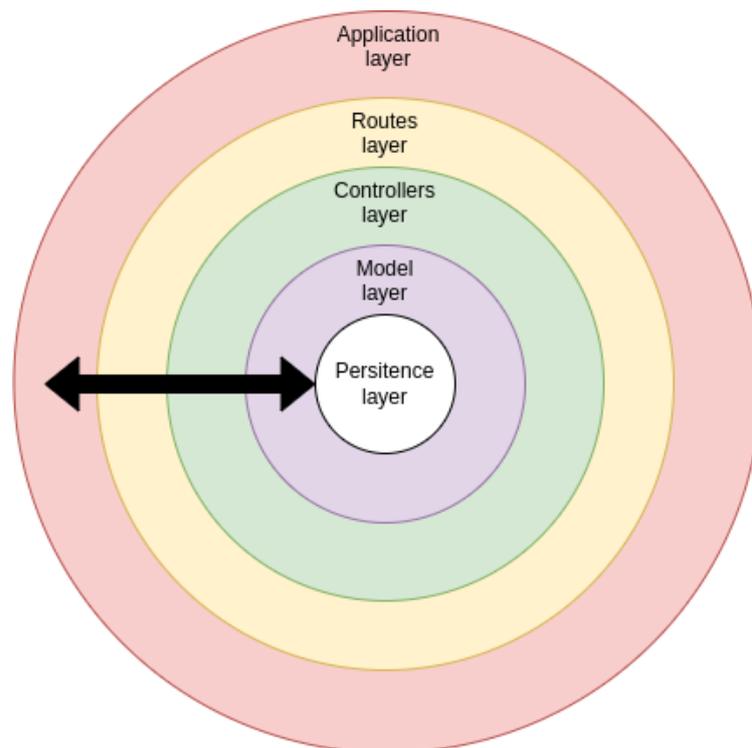


软件架构手册

作者	Germán Cocca
来源	来自于 FreeCodeCamp 网站
翻译	Linda (火龙果软件)
说明	本文首先介绍软件架构的主要概念，其次对于每个主题，将给出一个简短的理论介绍，最后分享一些代码示例，让您更清楚地了解这些示例的工作原理，下面请开始详细阅读下文吧！



目录

软件架构手册	1
一、 什么是软件架构?	3
二、 需要了解的重要软件架构概念	4
2.1 什么是客户端-服务器模型?	4
2.2 什么是 API?	5
2.3 什么是模块化?	6
三、 您的基础架构是怎样的?	6
四、 整体式架构.....	7
五、 微服务架构.....	8
5.1 什么是前端后端 (BFF) ?	10
5.2 如何使用负载均衡器和水平扩展.....	11
六、 您的基础架构所在的位置.....	13
6.1 本地托管	14
6.2 在云上托管.....	16
七、 需要了解的不同文件夹结构.....	19
7.1 多合一文件夹结构.....	20
7.2 图层文件夹结构.....	22
7.3 MVC 文件夹结构.....	30
八、 结论	37

大家好！在本手册中，您将了解软件架构这个广阔而复杂的领域。

当我第一次开始我的编码之旅时，我发现这是一个既令人困惑又令人生畏的领域。所以我会尽量避免你的困惑。

在本手册中，我将尝试为您提供一个简单、表面、易于理解的软件架构介绍。

我们将讨论软件世界中的架构，您应该了解的一些主要概念，以及当今使用最广泛的架构模式。

对于每个主题，我将给出一个简短的理论介绍。然后，我将分享一些代码示例，让您更清楚地了解这些示例的工作原理。让我们开始吧！

一、什么是软件架构？

根据[这个消息来源](#)：系统的软件架构表示与整体架构和行为相关的设计决策。

这很通用，对吧？是的。这正是过去在研究软件架构时让我感到困惑的原因。这是一个包含很多内容的话题，该术语用于谈论许多不同的事情。

我可以说的最简单的方式是，软件架构是指你在创建软件的过程中如何组织东西。这里的“东西”可以指：

- **实现详细信息**（即存储库的文件夹结构）
- **实现设计决策**（是使用服务器端还是客户端呈现？关系数据库还是非关系数据库？）
- 您选择的**技术**（您是否将 REST 或 GraphQL 用于您的 API？Python with Django 或 Node with Express for your 后端？）

- **系统设计**决策（例如您的**系统**是整体式还是分为微服务？）
- **基础架构**决策（您是在本地还是在云提供商上托管软件？）

这是许多不同的选择和可能性。更复杂的是，在这 5 个技术中，可以组合不同的模式。

这意味着，我可以拥有一个使用 REST 或 GraphQL 的整体式 API，一个托管在本地或云上的基于微服务的应用程序，等等。

为了更好地解释这种混乱，首先我们将解释一些基本的通用概念。然后，我们将介绍其中的一些划分，解释当今用于构建应用程序的最常见的架构模式或选择。

二、 需要了解的软件架构概念

2.1 什么是客户端-服务器模型？

客户端-服务器是一种模型，用于在资源或服务**提供程序**（服务器）和服务或资源请求者（**客户端**）之间构建应用程序的任务或工作负载。

简而言之，客户端是请求某种信息或执行操作的应用程序，服务器是根据客户端所做的事情发送信息或执行操作的程序。

客户端通常由在 Web 或移动应用程序上运行的前端应用程序表示（尽管也存在其他平台，后端应用程序也可以充当客户端）。服务器通常是后端应用程序。

为了用一个例子来说明这一点，假设你正在进入你最喜欢的社交网络。当您在浏览器上输入 URL 并按回车键时，您的浏览器将充当客户端应用程序并向社交网络服务器**发送请求**，社交网络服务器通过向您发送网站内容来**响应**。

如今，大多数应用程序都使用客户端-服务器模型。要记住的最重要的概念是客户端请求**服务器执行的资源或服务**。

另一个需要了解的重要概念是客户端和服务端是同一系统的一部分，但每个都是一个应用程序/程序。这意味着它们可以单独开发、托管和执行。

如果你不熟悉前端和后端之间的区别，[这里有一篇很酷的文章来解释它](#)。这是[另一篇](#)扩展客户端-服务器概念的文章。

2.2 什么是 API?

我们刚刚提到，客户端和服务端是相互通信以请求事物和响应事物的实体。这两个部分通常的通信方式是通过 API（应用程序编程接口）。

API 只不过是一组定义的规则，用于确定应用程序如何与另一个应用程序通信。这就像两部分之间的合同，上面写着“如果你发送 A，我会一直回复 B。如果你发送 C，我会一直回复 D...”等等。

有了这组规则，客户端确切地知道完成某项任务需要什么，服务端确切地知道当必须执行某个操作时客户端需要什么。

可以通过不同的方式实现 API。最常用的是 REST，SOAP 和 GraphQL。

关于 API 的通信方式，大多数情况下使用 HTTP 协议，内容以 JSON 或 XML 格式交换。

但是其他协议和内容格式是完全可能的。

如果您想扩展此主题，这里有[一篇不错的文章](#)供您阅读。

2.3 什么是模块化?

当我们谈论软件架构中的“模块化”时，我们指的是将大东西分成小块的做法。执行这种分解的做法是为了简化大型应用程序或代码库。

模块化具有以下优点：

- 它有利于划分关注点和功能，这有助于项目的可视化、理解和组织。
- 当项目被明确组织和细分时，它往往更容易维护，并且不容易出现错误和错误。
- 如果您的项目细分为许多不同的部分，则可以单独和独立地处理和修改每个部分，这通常非常有用。

我知道这听起来有点通用，但模块化或细分事物的做法是软件架构的重要组成部分。因此，只需将这个概念牢记在您的脑海中 - 当我们通过一些示例时，它会变得更加清晰和明显。

如果您想了解有关此主题的更多信息，我最近写了一篇[关于在 JS 中使用模块](#)的文章，您可能会觉得有用。

三、 您的基础架构是怎样的？

好了，现在让我们开始讨论好东西。我们将开始讨论组织软件应用程序的许多不同的方式，从如何组织项目背后的基础结构开始。

为了使所有这些不那么抽象，我们将使用一个假设的应用程序，我们称之为 Netflix。

附带评论：请记住，这个例子可能不是最现实的例子，我将假设/强迫情况以提出某些概念。

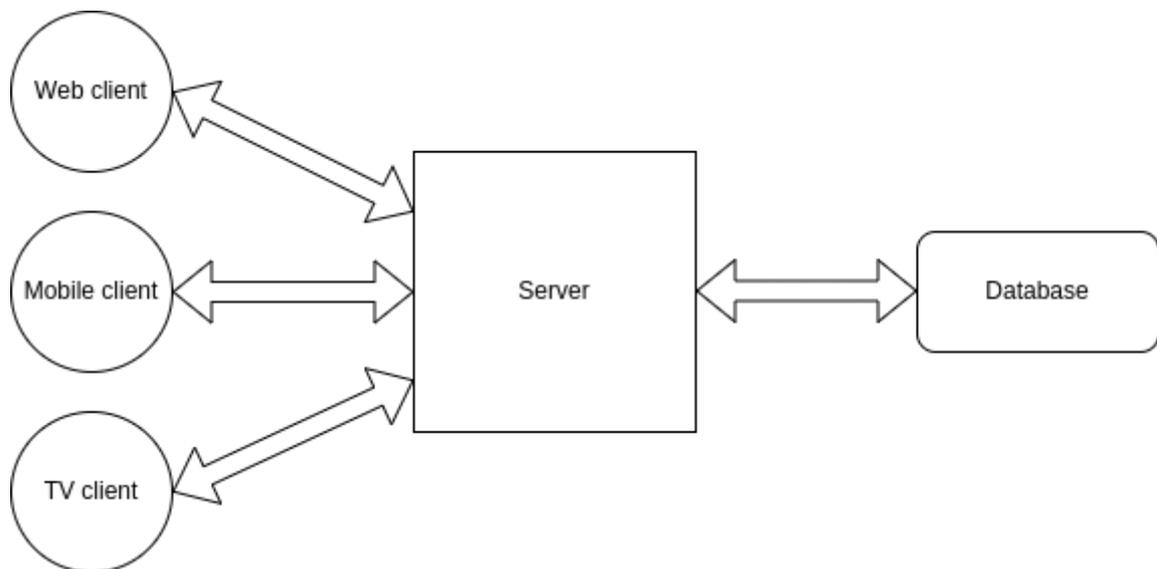
这里的想法是通过示例帮助您理解核心体系结构概念，而不是执行实际分析。

3.1 整体式架构

因此，Netflix 将是一个典型的视频流应用程序，用户可以在其中观看电影，连续剧，纪录片等。用户将能够在 Web 浏览器、移动应用程序和电视应用程序中使用该应用程序。

我们应用程序中包含的主要服务将是**身份验证**（以便人们可以创建帐户，登录等），**付款**（因此人们可以订阅和访问内容...因为你不认为这一切都是免费的，对吧？☹）和**流媒体**（所以人们可以实际观看他们所支付的内容）。

我们的架构的快速草图可能如下所示：



经典的整体式架构

在左侧，我们三个不同的前端应用程序，它们将充当此系统中的客户端。例如，它们可能是用 React 和 React-native 开发的。

我们有一个服务器，它将接收来自所有三个客户端应用程序的请求，在必要时与数据库通信，并相应地响应每个前端。后端可以用 Node 和 Express 开发，比方说。

这种体系结构称为**整体式架构**，因为有一个服务器应用程序负责系统的所有功能。在我们的例子中，如果用户想要进行身份验证、向我们付款或观看我们的一部电影，则所有请求都将发送到同一个服务器应用程序。

整体式设计的主要优点是其简单性。它的功能和所需的设置简单易懂，这就是为什么大多数应用程序以这种方式开始的原因。

3.2 微服务架构

所以事实证明，Netflix 完全在摇摆不定。我们刚刚发布了最新一季的“陌生暴徒”，这是一部关于青少年说唱歌手的精彩科幻系列，而我们的电影“Agent 404”（关于一个秘密特工潜入一家公司，冒充高级程序员，但实际上对代码一无所知）正在打破所有记录.....

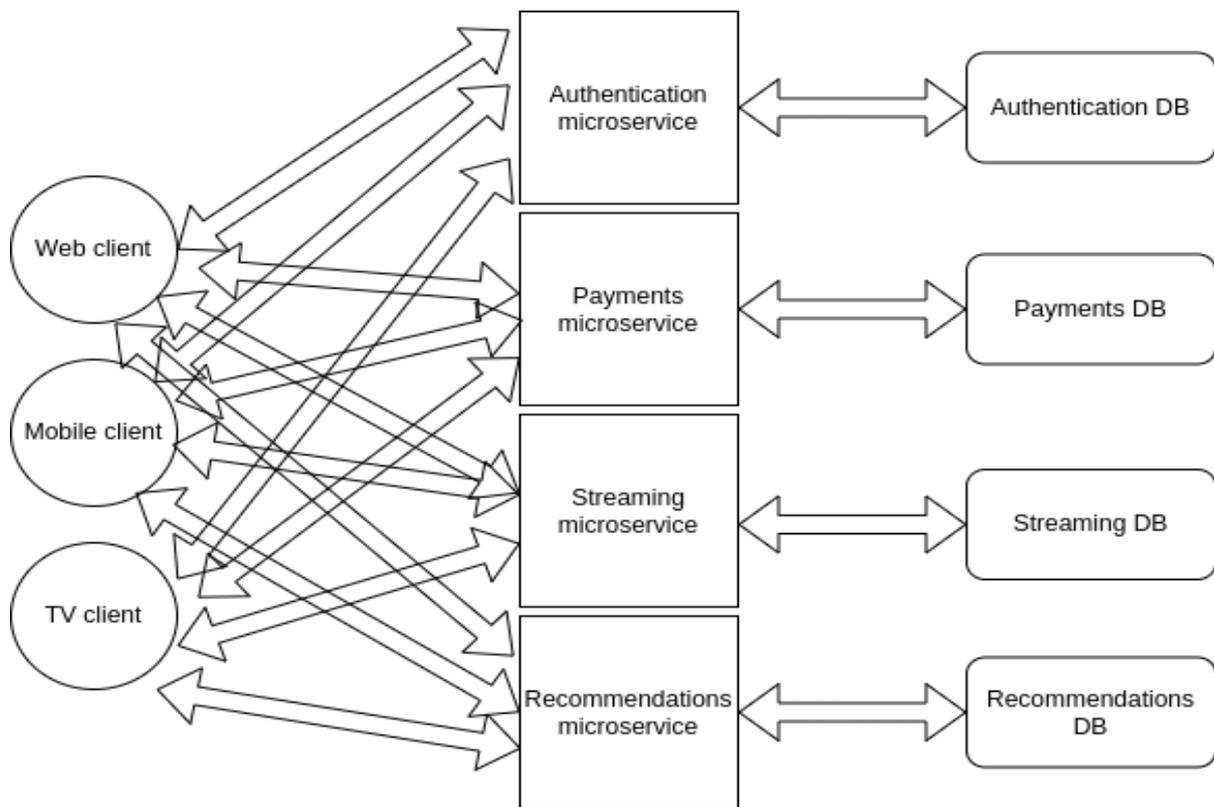
我们每个月都会收到来自世界各地的数以万计的新用户，这对我们的业务来说是件好事，但对我们的整体应用程序来说却不是那么好。

最近，我们一直在经历服务器响应时间的延迟，即使我们已经**垂直扩展**了服务器（在其中放置了更多的 RAM 和 GPU），但可怜的东西似乎无法承受它所承担的负载。

此外，我们一直在系统中开发新功能（例如读取用户偏好并推荐适合用户个人资料的电影的推荐工具），**我们的代码库开始看起来庞大且非常复杂。**

深入分析这个问题，我们发现占用最多资源的功能是流媒体，而身份验证和支付等其他服务并不代表很大的负载。

为了解决这个问题，我们将实现一个**微服务架构**，如下所示：



第一个微服务实施

因此，如果你对这一切不熟悉，你可能会想“微服务到底是什么”，对吧？好吧，我们可以将其定义为将服务器端功能划分为许多仅负责一个或几个特定功能的小型服务器的概念。

按照我们的示例，以前我们只有一个服务器负责所有功能（整体架构）。实施微服务后，我们将有一个服务器负责身份验证，另一个负责支付，另一个负责流式传输，最后一个负责推荐。

当用户想要登录时，客户端应用程序将与身份验证服务器通信，当用户想要付款时，客户端应用程序将与支付服务器通信，当用户想要观看某些内容时，客户端应用程序将与流媒体服务器通信。

所有这些通信都是通过 API 进行的，就像使用常规单片服务器一样（或通过 [Kafka](#) 或 [RabbitMQ](#) 等其他通信系统）。唯一的区别是，现在我们有不同的服务器负责不同的操作，而不是一个执行所有操作的服务器。

这听起来有点复杂，确实如此，但微服务为我们提供了以下好处：

- 您可以**根据需要缩放特定服务**，而不是一次缩放整个后端。按照我们的示例，当我们开始遇到性能问题时，我们垂直扩展了整个服务器 - 但实际上请求更多资源的功能只是流媒体。现在我们将流媒体功能分离到单个服务器中，我们可以仅扩展该服务器，而其他的则无需运行，只要它们保持正常工作。
- 功能将更加**松散耦合**，这意味着我们将能够独立开发和部署它们。
- 每个服务器的代码库将更小、**更简单**。这对于从一开始就与我们合作的开发人员来说很好，对于新开发人员来说也更容易、更快速地理解。

微服务是一种设置和管理更复杂的架构，这就是为什么它只对非常大的项目有意义。大多数项目将从整体式架构开始，仅出于性能原因才迁移到微服务。

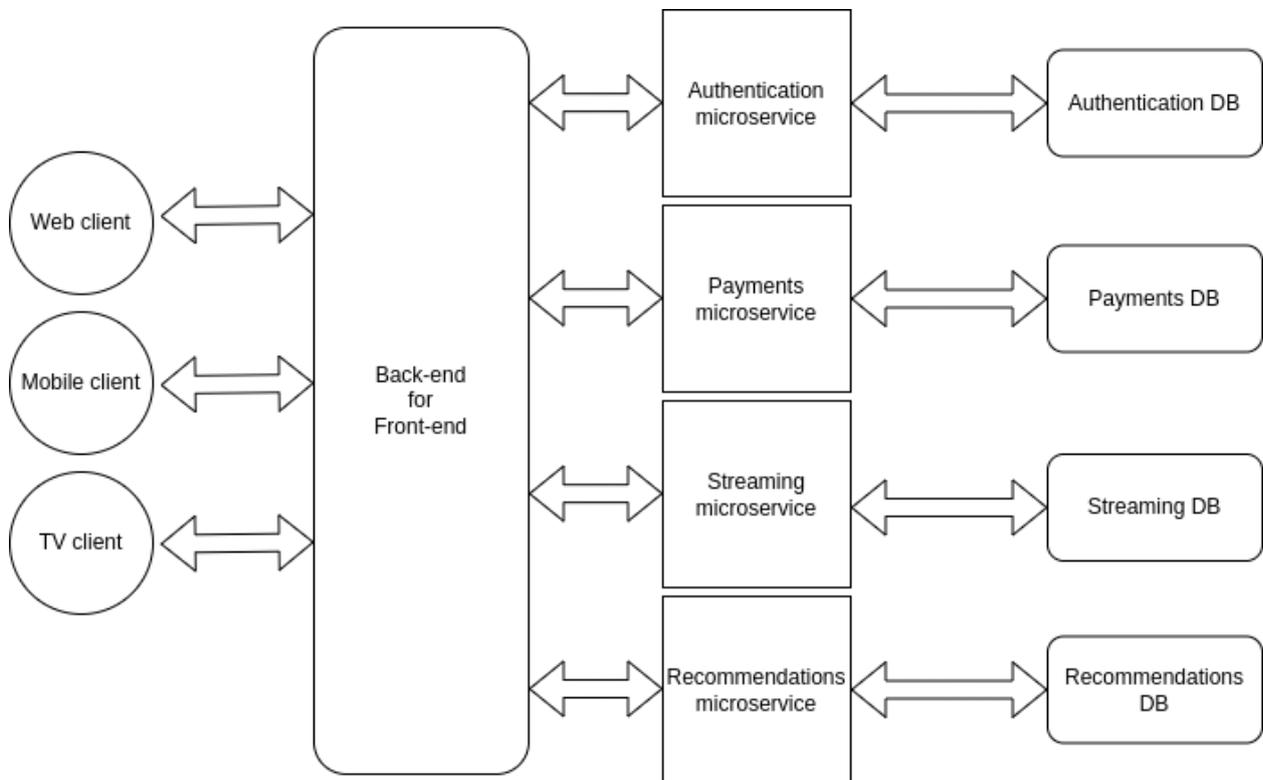
如果你想了解更多关于微服务的信息，[这里有一个非常好的解释](#)。

3.3 什么是前端后端（BFF）？

实现微服务时出现的一个问题是与前端应用程序的通信变得更加复杂。现在我们有许多服务器负责不同的事情，这意味着前端应用程序需要跟踪该信息，以了解向谁发出请求。

通常，通过在前端应用和微服务之间实现中间层来解决此问题。该层将接收所有前端请求，将它们重定向到相应的微服务，接收微服务响应，然后将响应重定向到相应的前端应用。

BFF 模式的好处是，我们可以获得微服务架构的好处，而不会使与前端应用程序的通信过于复杂。



BFF 实施

这里有一个[视频](#)，解释了 BFF 模式，如果你想了解更多。

3.4 如何使用负载均衡器和水平扩展

因此，我们的流媒体应用程序以指数级的速度不断增长和增长。我们全球有数百万用户 24/7 全天候观看我们的电影，并且比我们预期的要快，我们再次开始遇到性能问题。

我们再次发现流媒体服务是压力最大的服务，我们已经尽可能地**垂直扩展**了该服务器。将该服务进一步细分为更多微服务是没有意义的，因此我们决定**水平扩展**该服务。

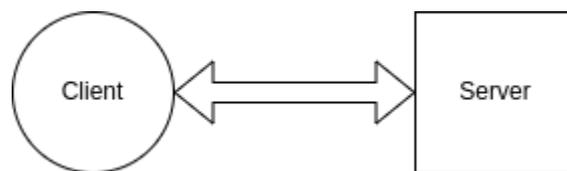
在我们提到**垂直扩展**意味着向单个服务器/计算机添加更多资源 (RAM、磁盘空间、GPU 等)

之前。另一方面，**水平扩展**意味着设置更多服务器来执行相同的任务。

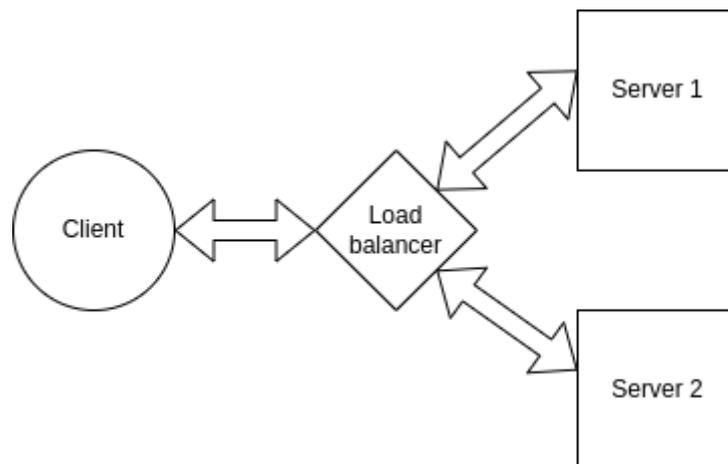
我们现在不再只有一个服务器负责流式传输，而是有三个。然后，客户端执行的请求将在这三台服务器之间平衡，以便所有服务器都处理可接受的负载。

这种请求分发通常由称为**负载均衡器**的东西执行。负载均衡器充当我们服务器的**反向代理**，在客户端请求到达服务器之前拦截客户端请求，并将该请求重定向到相应的服务器。

虽然典型的客户端-服务器连接可能如下所示：



这就是我们以前拥有的使用负载均衡器，我们可以在多个服务器之间分发客户端请求：

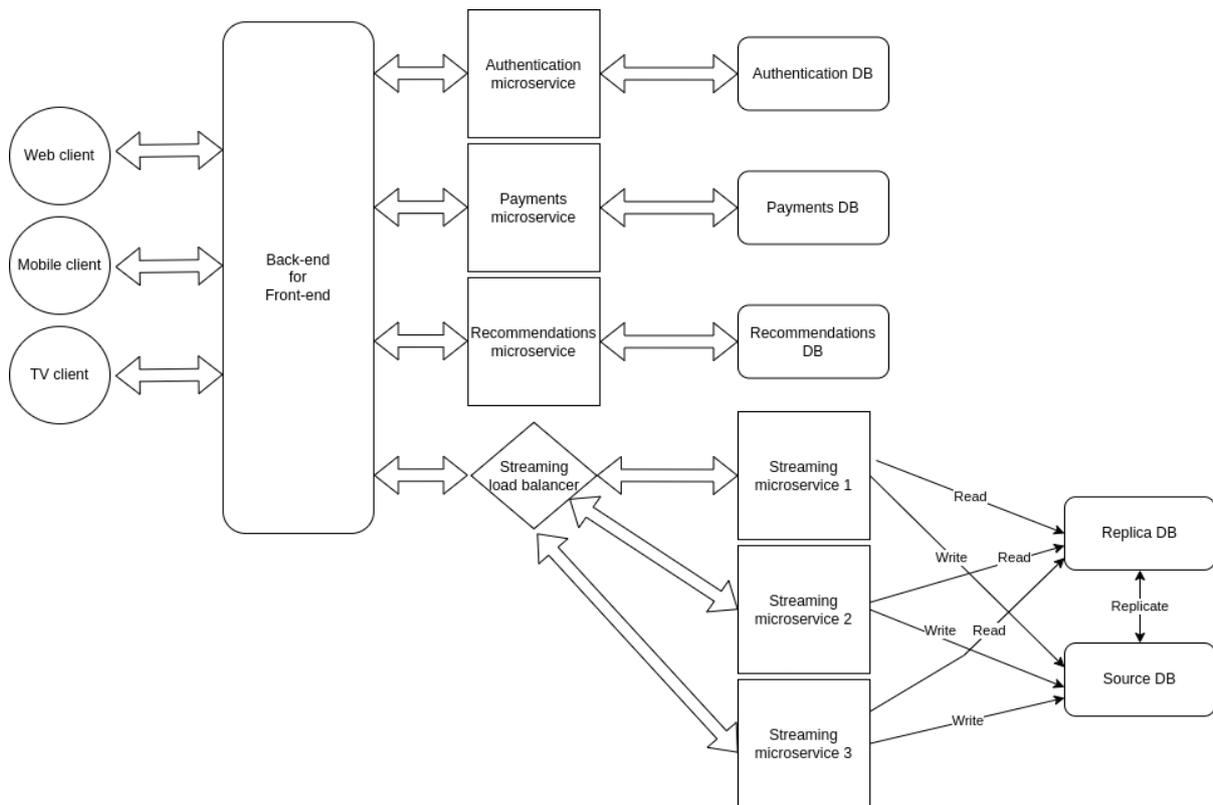


这就是我们现在想要的您应该知道，水平扩展也可以与服务器一样使用数据库。实现此目的的一种方法是使用源-副本模型，在该模型中，一个特定的源数据库将接收所有写入查询，并沿一个或多个副本数据库复制其数据。副本数据库将接收并响应所有读取查询。

数据库复制的优点是：

- 更好的性能：此模型提高了性能，允许并行处理更多查询。
- 可靠性和可用性：如果您的某个数据库服务器因任何原因被破坏或无法访问，数据仍保留在其他数据库中。

因此，在实施负载均衡器、水平扩展和数据库复制后，我们的架构可能如下所示：



我们的水平扩展架构这里有一个[很棒的负载均衡器视频解释](#)，如果你有兴趣了解更多信息。

附带评论：当我们谈论微服务、负载均衡器和扩展时，我们可能总是在谈论后端应用程序。

对于前端应用程序，它们大多总是作为整体式开发，尽管还有一个奇怪有趣的东西叫做[微前端](#)。

四、您的基础架构所在的位置

现在我们已经对如何组织应用程序基础结构有了基本的想法，接下来要考虑的是我们将把所有这些东西放在哪里。

正如我们将要看到的，在决定在何处以及如何托管应用程序时，主要有三个选项：内部部署、传统服务器提供商或云。

4.1 本地托管

内部部署意味着您拥有运行应用的硬件。过去，这曾经是最传统的托管应用程序的方式。公司过去有专门的服务器房间和专门用于硬件设置和维护的团队。

此选项的好处是公司可以完全控制硬件。不好是它需要空间、时间和金钱。

想象一下，如果你想水平扩展某个服务器，那将意味着购买更多的设备，设置它，不断监督它，修复任何损坏的东西.....如果您以后需要缩小该服务器的规模，那么通常您在购买这些东西后无法退货。

对于大多数公司来说，拥有本地服务器意味着将大量资源投入到与公司目标没有直接关系的任务上。



我们如何想象我们在 Netflix 的服务器机房



它是如何结束的

在本地服务器上仍然有意义的一种情况是处理非常微妙或私人的信息。例如，考虑运行发电厂的软件或私人银行信息。其中许多组织决定在本地使用服务器作为完全控制其软件和硬件的一种方式。

4.2 传统服务器提供商

对于大多数公司来说，更舒适的选择是传统的服务器提供商。这些公司拥有自己的服务器，他们只是租用它们。您决定项目需要哪种硬件，并按月支付费用（或根据其他条件支付一定金额）。

此选项的优点在于，您无需再担心任何与硬件相关的内容。提供商负责它，作为一家软件公司，您只担心您的主要目标，即软件。

另一个很酷的事情是，扩大或缩小规模既简单又无风险。如果您需要更多硬件，则需要付费。如果您不再需要它，只需停止付款即可。

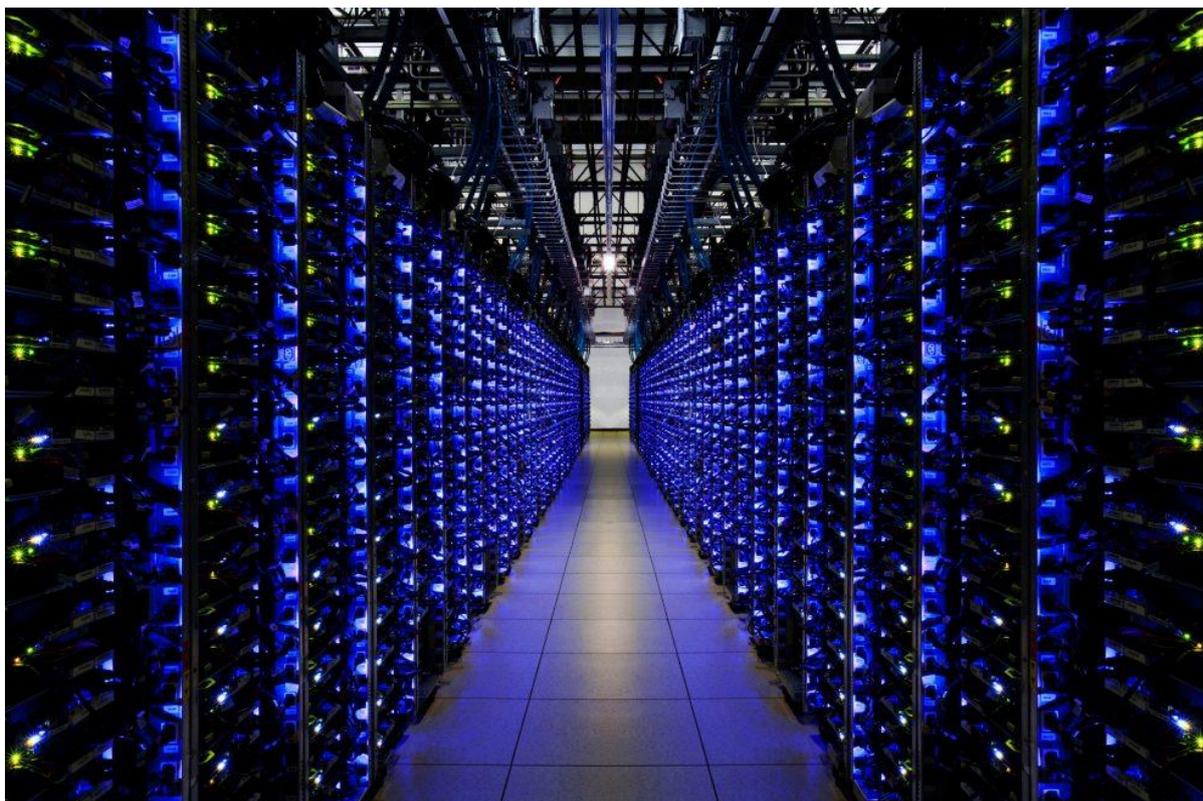
众所周知的服务器提供商的一个示例是[托管程序](#)。

4.3 在云上托管

如果你已经接触技术一段时间了，你可能不止一次听到“云”这个词。起初，这听起来像是抽象而神奇的东西，但实际上它背后的只不过是亚马逊、谷歌和微软等公司拥有的巨大数据中心。

在某些时候，这些公司发现他们拥有他们没有一直使用的计算能力。由于无论您是否使用它，所有这些硬件仍然代表着成本，因此明智的做法是将这种计算能力商业化给其他人。

这就是云计算。使用 AWS (Amazon Web Services), **Google Cloud** 或 Microsoft **Azure** 等不同服务，我们能够在这些公司的数据中心托管我们的应用程序，并利用所有这些计算能力。



“云”实际上可能是什么样子的

在了解云服务时，请务必注意，您可以通过许多不同的方式使用它们：

4.3.1 传统的

第一种方法是以类似于使用传统服务器提供商的方式使用它们。您可以选择所需的硬件类型，并按月支付确切的费用。

4.3.2 弹性的

第二种方法是利用大多数提供商提供的“弹性”计算。“弹性”意味着应用程序的硬件容量将根据应用的使用情况自动增长或缩小。

例如，您可以从具有 8GB RAM 和 500GB 磁盘空间的服务器开始。如果您的服务器开始收到越来越多的请求，并且这些容量不再足以提供良好的性能，则系统可以自动执行垂直或水平扩展。

很棒的事情是您可以事先配置所有这些，而不必担心再次使用它。当服务器自动扩展和缩减时，您只需为使用的资源付费。

4.3.2 无服务器

使用云计算的另一种方法是使用无服务器体系结构。

遵循此模式，您将不会有接收所有请求并响应它们的服务器。相反，您将有单独的函数映射到访问点（类似于 API 端点）。

这些函数将在每次收到请求时执行，并执行您为其编程的任何操作（连接到数据库、执行 CRUD 操作或您可以在常规服务器中执行的任何其他操作）。

无服务器架构的优点在于，您可以忘记所有关于服务器维护和扩展的信息。您只需在需要时执行函数，并且每个函数都会根据需要自动扩展和缩减。

作为客户，您只需为函数执行的次数和每次执行的处理时间付费。

如果您想了解更多信息，下面是[无服务器模式的说明](#)。

4.3.4 许多其他服务

您可能会看到弹性和无服务器服务如何为设置软件基础架构提供非常简单方便的替代方案。

除了与服务器相关的服务外，云提供商还提供大量其他解决方案，例如关系和非关系数据库、文件存储服务、缓存服务、身份验证服务、机器学习和数据处理服务、监控和性能分析等。一切都托管在云中。

通过 [Terraform](#) 或 [AWS Cloud 形成](#) 等工具，我们甚至可以将基础设施设置为代码。这意味着我们可以编写一个脚本，在几分钟内在云上设置服务器、数据库以及我们可能需要的任何其他内容。

从工程的角度来看，这是令人兴奋的，对于我们开发人员来说非常方便。如今，云计算提供了一套非常完整的解决方案，可以轻松地从小型项目适应地球上最大的数字产品。这就是为什么现在越来越多的软件项目选择将其基础架构托管在云中的原因。

如前所述，最常用和最知名的云提供商是 [AWS](#)，[Google Cloud](#) 和 [Azure](#)。虽然还有其他选择，如 [IBM](#)，[DigitalOcean](#) 和 [Oracle](#)。

这些提供商中的大多数都提供相同类型的服务，尽管它们可能具有不同的名称。例如，无服务器函数在 AWS 上称为“lambdas”，在 Google 云上称为“云函数”。

五、 需要了解的不同文件夹结构

好的，到目前为止，我们已经了解了架构如何引用基础结构组织和托管。现在让我们看看一些代码以及架构如何引用文件夹结构和代码模块化。

5.1 多合一文件夹结构

为了说明为什么文件夹结构很重要，让我们构建一个虚拟的示例 API。我们将有一个兔子的  模拟数据库，API 将对其执行 [CRUD](#) 操作。我们将使用 Node 和 Express 构建它。

这是我们的第一种方法，完全没有文件夹结构。我们的存储库将由文件夹和文件组成。node
modulesapp.jspackage-lock.jsonpackage.json

```
> node_modules
JS app.js
{} package-lock.json
{} package.json
```

在我们的 app.js 文件中，我们将拥有我们的小服务器、模拟数据库和两个端点：

```
// App.js
const express = require('express');

const app = express()
const port = 7070

// Mock DB
const db = [
  { id: 1, name: 'John' },
  { id: 2, name: 'Jane' },
  { id: 3, name: 'Joe' },
  { id: 4, name: 'Jack' },
  { id: 5, name: 'Jill' },
  { id: 6, name: 'Jak' },
  { id: 7, name: 'Jana' },
  { id: 8, name: 'Jan' },
  { id: 9, name: 'Jas' },
  { id: 10, name: 'Jasmine' },
]

/* Routes */
```

```
app.get('/rabbits', (req, res) => {
  res.json(db)
})

app.get('/rabbits/:idx', (req, res) => {
  res.json(db[req.params.idx])
})

app.listen(port, () => console.log(`[server]: Server is running at http://localhost:${port}`))
```

如果我们测试端点，我们将看到它们工作得很好：

```
http://localhost:7070/rabbits

# [
#   {
#     "id": 1,
#     "name": "John"
#   },
#   {
#     "id": 2,
#     "name": "Jane"
#   },
#   {
#     "id": 3,
#     "name": "Joe"
#   },
#   ....
# ]

###

http://localhost:7070/rabbits/1
```

```
# {  
#   "id": 2,  
#   "name": "Jane"  
# }
```

那么这有什么问题呢？没什么，实际上，它工作得很好。只有当代码库变得更大、更复杂，并且我们开始向 API 添加新功能时，才会出现问题。

与我们之前在解释整体架构时所讨论的类似，一开始将所有内容都放在一个地方既好又容易。但是，当事情开始变得越来越大和复杂时，这是一种令人困惑且难以遵循的方法。

遵循模块化原则，更好的主意是为我们需要执行的不同职责和操作使用不同的文件夹和文件。

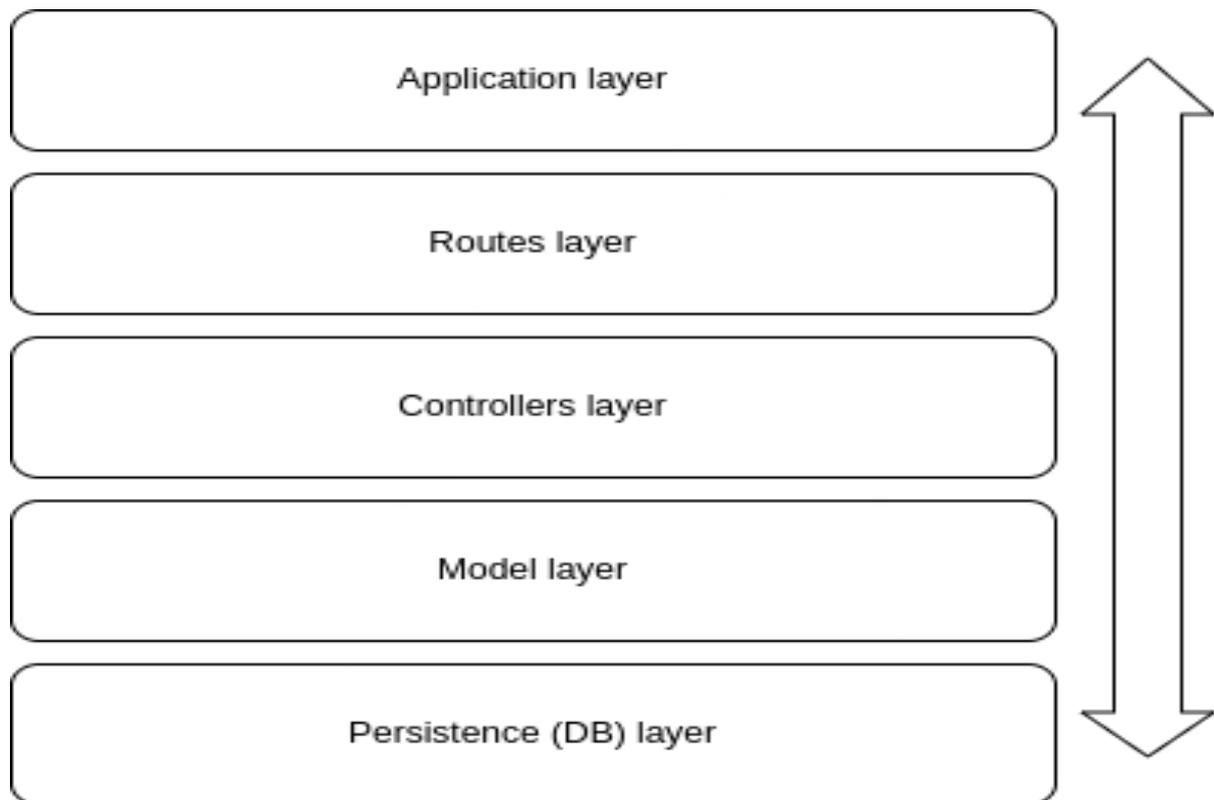
为了更好地说明这一点，让我们向 API 添加新功能，看看如何在层架构的帮助下采用模块化方法。

5.2 图层文件夹结构

层体系结构是关于将关注点和责任划分到不同的文件夹和文件中，并且只允许在某些文件夹和文件之间进行直接通信。

您的项目应该有多少层，每个层应该具有什么名称以及应该处理什么操作都是讨论的问题。因此，让我们看看我认为对于我们的示例来说，什么是好方法。

我们的应用程序将有五个不同的层，这些层将按以下方式排序：



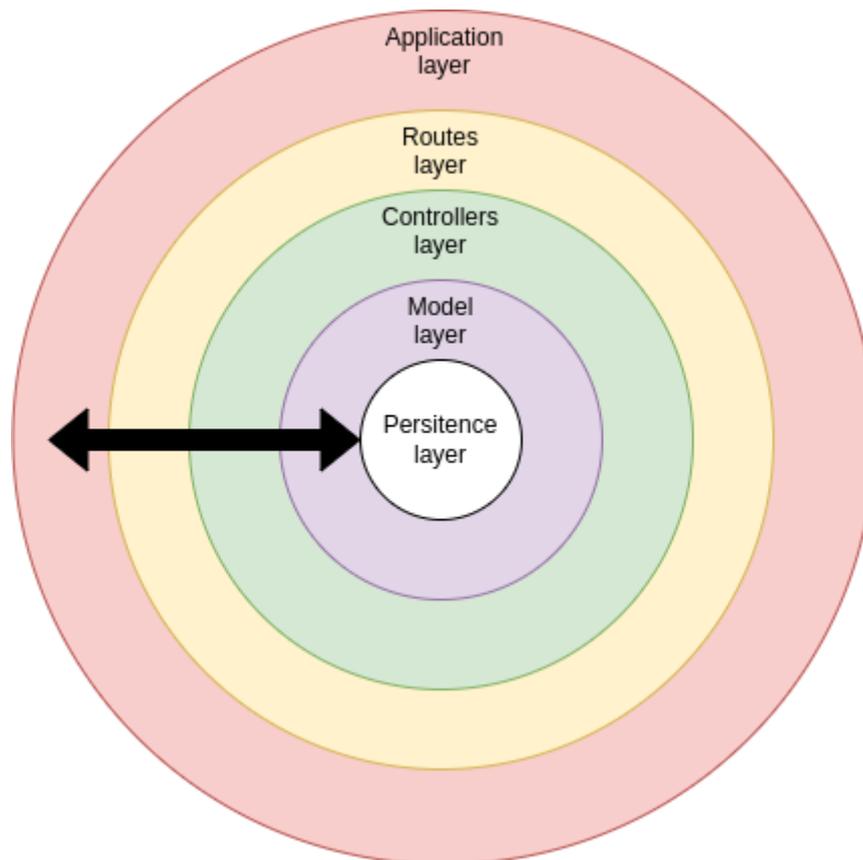
应用层

- 应用层将具有服务器的基本设置和与路由的连接（下一层）。
- 路由层将具有所有路由的定义以及与控制器的连接（下一层）。
- 控制器层将具有我们想要在每个端点中执行的实际逻辑以及与模型层的连接（下一层，您明白了.....
- 模型层将保存与我们的模拟数据库交互的逻辑。
- 最后，持久性层是我们数据库所在的位置。

您可以看到这种方法更加结构化，并且有明确的关注点划分。这听起来像很多样板。但是在设置它之后，这种架构将使我们能够清楚地知道每个东西在哪里，以及哪些文件夹和文件负责我们的应用程序执行的每个操作。

要记住的一件重要的事情是，在这些架构中，层之间有一个定义的通信流，必须遵循该通信流才能有意义。

这意味着请求首先必须经过第一层，然后是第二层，然后是第三层，依此类推。任何请求都不应该跳过层，因为这会扰乱架构的逻辑以及它为我们提供的组织和模块化的好处。



描绘我们架构的另一种方式

现在让我们看看一些代码。使用层体系结构，我们的文件夹结构可能如下所示：

```
└─ db
  └─ JS db.js
  └─ > node_modules
  └─ rabbits
    └─ controllers
      └─ JS rabbits.controllers.js
    └─ models
      └─ JS rabbits.models.js
    └─ routes
      └─ JS rabbits.routes.js
  └─ JS app.js
  └─ {} package-lock.json
  └─ {} package.json
```

- 我们有一个名为的新文件夹将保存我们的数据库文件。 db
- 另一个名为该文件夹将保存与该实体相关的路由，控制器和模型。 rabbits
- app.js 设置我们的服务器并连接到路由。

```
• // App.js
• const express = require('express');
•
• const rabbitRoutes =
  require('./rabbits/routes/rabbits.routes')
•
• const app = express()
• const port = 7070
•
• /* Routes */
• app.use('/rabbits', rabbitRoutes)
•
• app.listen(port, () => console.log(`[server]: Server is
  running at http://localhost:${port}`))
•
```

- rabbits.routes.js 保存与此实体相关的每个端点，并将它们链接到相应的控制器（我们想要在请求命中该端点时执行的功能）。

```
• // rabbits.routes.js
```

```
• const express = require('express')
• const bodyParser = require('body-parser')
•
• const jsonParser = bodyParser.json()
•
• const { listRabbits, getRabbit, editRabbit, addRabbit,
  deleteRabbit } =
  require('../controllers/rabbits.controllers')
•
• const router = express.Router()
•
• router.get('/', listRabbits)
•
• router.get('/:id', getRabbit)
•
• router.put('/:id', jsonParser, editRabbit)
•
• router.post('/', jsonParser, addRabbit)
•
• router.delete('/:id', deleteRabbit)
•
• module.exports = router
•
```

- rabbits.controllers.js 保存与每个终结点对应的逻辑。在这里，我们对函数应该作为输入的内容、应该执行什么进程以及应该返回什么进行编程的地方。☺ 此外，每个控制器链接到相应的模型函数（将执行与数据库相关的操作）。

```
• // rabbits.controllers.js
• const { getAllItems, getItem, editItem, addItem, deleteItem }
  = require('../models/rabbits.models')
•
• const listRabbits = (req, res) => {
•   try {
•     const resp = getAllItems()
•     res.status(200).send(resp)
•
•   } catch (err) {
•     res.status(500).send(err)
•   }
• }
```

```
•   }
• }
•
• const getRabbit = (req, res) => {
•   try {
•     const resp = getItem(parseInt(req.params.id))
•     res.status(200).send(resp)
•
•   } catch (err) {
•     res.status(500).send(err)
•   }
• }
•
• const editRabbit = (req, res) => {
•   try {
•     const resp = editItem(req.params.id, req.body.item)
•     res.status(200).send(resp)
•   } catch (err) {
•     res.status(500).send(err)
•   }
• }
•
• const addRabbit = (req, res) => {
•   try {
•     console.log( req.body.item )
•     const resp = addItem(req.body.item)
•     res.status(200).send(resp)
•   } catch (err) {
•     res.status(500).send(err)
•   }
• }
•
• const deleteRabbit = (req, res) => {
•   try {
•     const resp = deleteItem(req.params.idx)
•     res.status(200).send(resp)
•   } catch (err) {
•     res.status(500).send(err)
•   }
• }
```

```
•  
• module.exports = { listRabbits, getRabbit, editRabbit,  
  addRabbit, deleteRabbit }  
•
```

- rabbits.models.js 是我们定义将对数据库执行 CRUD 操作的函数的地方。每个函数表示不同类型的操作（读取一个、全部读取、编辑、删除等）。此文件是连接到我们数据库的文件。

```
• // rabbits.models.js  
• const db = require('../db/db')  
•  
• const getAllItems = () => {  
•   try {  
•     return db  
•   } catch (err) {  
•     console.error("getAllItems error", err)  
•   }  
• }  
•  
• const getItem = id => {  
•   try {  
•     return db.filter(item => item.id === id)[0]  
•   } catch (err) {  
•     console.error("getItem error", err)  
•   }  
• }  
•  
• const editItem = (id, item) => {  
•   try {  
•     const index = db.findIndex(item => item.id === id)  
•     db[index] = item  
•     return db[index]  
•   } catch (err) {  
•     console.error("editItem error", err)  
•   }  
• }  
•  
• const addItem = item => {
```

```
•   try {  
•     db.push(item)  
•     return db  
•   } catch (err) {  
•     console.error("addItem error", err)  
•   }  
• }  
•  
• const deleteItem = id => {  
•   try {  
•     const index = db.findIndex(item => item.id === id)  
•     db.splice(index, 1)  
•     return db  
•     return db  
•   } catch (err) {  
•     console.error("deleteItem error", err)  
•   }  
• }  
•  
• module.exports = { getAllItems, getItem, editItem, addItem,  
  deleteItem }  
•
```

- 最后, 托管我们的模拟数据库。在实际项目中, 这是实际数据库连接可能所在的位置。db.js

```
• // db.js  
• const db = [  
•   { id: 1, name: 'John' },  
•   { id: 2, name: 'Jane' },  
•   { id: 3, name: 'Joe' },  
•   { id: 4, name: 'Jack' },  
•   { id: 5, name: 'Jill' },  
•   { id: 6, name: 'Jak' },  
•   { id: 7, name: 'Jana' },  
•   { id: 8, name: 'Jan' },  
•   { id: 9, name: 'Jas' },  
•   { id: 10, name: 'Jasmine' },  
• ]  
•  
• module.exports = db
```

•

```
module.exports = db
```

如我们所见，此体系结构下有更多的文件夹和文件。但因此，我们的代码库更加结构化和清晰。一切都有自己的位置，不同文件之间的通信都明确定义。

这种组织极大地促进了新功能的添加、代码修改和错误修复。

一旦您熟悉了文件夹结构并知道在哪里可以找到每个内容，您就会发现处理这些较短和较小的文件非常方便，而不必滚动浏览一两个将所有内容放在一起的大文件。

我也支持为您的应用程序中的每个主要实体（在我们的例子中是兔子）提供一个文件夹。

这样可以更清楚地了解每个文件的相关内容。

假设我们现在也想添加新功能来添加/编辑/删除猫和狗。我们将为每个文件夹创建新文件夹，每个文件夹都有自己的路由、控制器和模型文件。这个想法是将关注点分开，并将每件事放在自己的位置。 🐱🐶

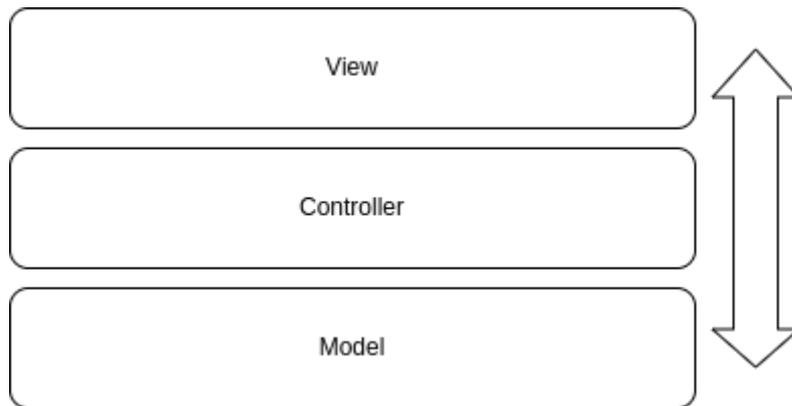
5.3 MVC 文件夹结构

MVC 是一种架构模式，代表**模型视图控制器**。我们可以说 MVC 架构就像是层架构的简化，也结合了应用程序的前端（UI）。

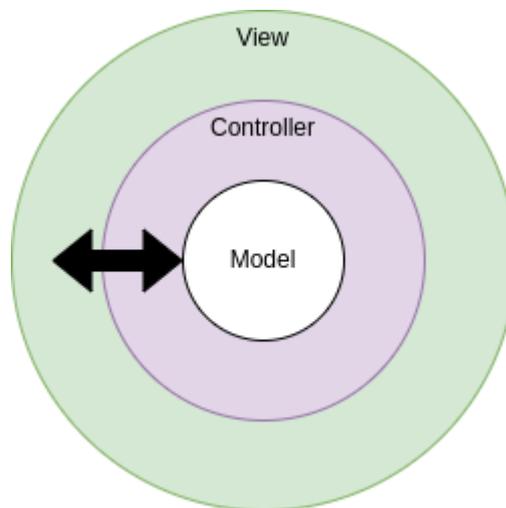
在此体系结构下，我们只有三个主要层：

- 视图层将负责呈现 UI。
- 控制器层将负责定义路由和每个路由的逻辑。

- 模型层将负责与我们的数据库进行交互。



与以前相同，每一层将仅与下一层交互，因此我们有一个明确定义的通信流。



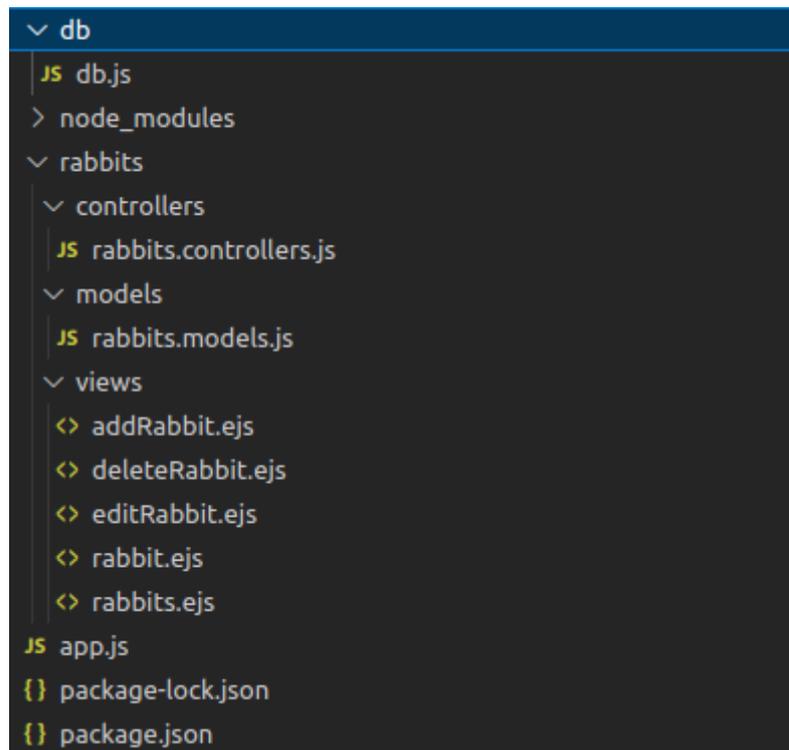
描绘我们构建的另一种方式

有许多框架允许你实现开箱即用的 MVC 架构（例如 [Django](#) 或 [Ruby on Rails](#)）。要使用 Node 和 Express 来做到这一点，我们需要一个像 [EJS](#) 这样的模板引擎。

如果你不熟悉模板引擎，它们只是一种轻松渲染 HTML 的方法，同时利用变量、循环、条件等编程功能（非常类似于我们在 React 中使用 JSX 所做的）。

正如我们将在一秒钟内看到的，我们将为我们要呈现的每种页面创建 EJS 文件，并且从每个控制器中，我们将呈现这些文件作为我们的响应，并将相应的响应作为变量传递给它们。

我们的文件夹结构将如下所示：



- 看到我们摆脱了之前的大部分文件夹并保留了和文件夹。dbcontrollersmodels
- 我们添加了与要呈现的每个页面/响应相对应的文件夹。views
- db.js 和文件保持完全相同。models.js
- 我们的看起来像这样：app.js

```

• // App.js
• const express = require("express");
• var path = require('path');
•
• const rabbitControllers =
  require("./rabbits/controllers/rabbits.controllers")
•
• const app = express()
• const port = 7070
•
• // Ejs config
• app.set("view engine", "ejs")
• app.set('views', path.join(__dirname, './rabbits/views'))
  
```

```

•
• /* Controllers */
• app.use("/rabbits", rabbitControllers)
•
• app.listen(port, () => console.log(`[server]: Server is
  running at http://localhost:${port}`))
•

```

- rabbits.controllers.js 更改以定义路由、连接到相应的模型函数以及为每个请求呈现相应的视图。看到在 render 方法中，我们将请求响应作为参数传递给视图。😊

```

• // rabbits.controllers.js
• const express = require('express')
• const bodyParser = require('body-parser')
•
• const jsonParser = bodyParser.json()
•
• const { getAllItems, getItem, editItem, addItem, deleteItem }
  = require('../models/rabbits.models')
•
• const router = express.Router()
•
• router.get('/', (req, res) => {
•   try {
•     const resp = getAllItems()
•     res.render('rabbits', { rabbits: resp })
•
•   } catch (err) {
•     res.status(500).send(err)
•   }
• })
•
• router.get('/:id', (req, res) => {
•   try {
•     const resp = getItem(parseInt(req.params.id))
•     res.render('rabbit', { rabbit: resp })
•
•   } catch (err) {

```

```
•     res.status(500).send(err)
•   }
• })
•
• router.put('/:id', jsonParser, (req, res) => {
•   try {
•     const resp = editItem(req.params.id, req.body.item)
•     res.render('editRabbit', { rabbit: resp })
•
•
•   } catch (err) {
•     res.status(500).send(err)
•   }
• })
•
• router.post('/', jsonParser, (req, res) => {
•   try {
•     const resp = addItem(req.body.item)
•     res.render('addRabbit', { rabbits: resp })
•
•
•   } catch (err) {
•     res.status(500).send(err)
•   }
• })
•
• router.delete('/:id', (req, res) => {
•   try {
•     const resp = deleteItem(req.params.idx)
•     res.render('deleteRabbit', { rabbits: resp })
•
•
•   } catch (err) {
•     res.status(500).send(err)
•   }
• })
•
• module.exports = router
•
```

- 最后，在视图文件中，我们将收到的变量作为参数并将其呈现为 HTML。

```
• <!-- Rabbits view -->
```

```
• <!DOCTYPE html>
• <html lang="en">
•   <body>
•     <header>All rabbits</header>
•     <main>
•       <ul>
•         <% rabbits.forEach(function(rabbit) { %>
•           <li>
•             Id: <%= rabbit.id %>
•             Name: <%= rabbit.name %>
•           </li>
•         <% }) %>
•       </ul>
•     </main>
•   </body>
• </html>
•
```

```
<!-- Rabbit view -->
<!DOCTYPE html>
<html lang="en">
  <body>
    <header>Rabbit view</header>
    <main>
      <p>
        Id: <%= rabbit.id %>
        Name: <%= rabbit.name %>
      </p>
    </main>
  </body>
</html>
```

现在我们可以转到浏览器，点击 <http://localhost:7070/rabbits> 并得到：

All rabbits

- Id: 1 Name: John
- Id: 2 Name: Jane
- Id: 3 Name: Joe
- Id: 4 Name: Jack
- Id: 5 Name: Jill
- Id: 6 Name: Jak
- Id: 7 Name: Jana
- Id: 8 Name: Jan
- Id: 9 Name: Jas
- Id: 10 Name: Jasmine

<http://localhost:7070/rabbits/>

Rabbit view

Id: 2 Name: Jane

And that's MVC!



六、 结论

我希望所有这些例子都能帮助你理解当我们提到软件世界中的“架构”时，我们在谈论什么。

正如我在开头所说，这是一个庞大而复杂的话题，通常包含许多不同的内容。

在这里，我们介绍了基础架构模式和系统、托管选项和云提供商，最后介绍了一些可以在项目中使用的常见且有用的文件夹结构。

我们已经了解了垂直和水平扩展、整体式应用程序和微服务、弹性和无服务器云计算.....很多事情。但这只是冰山一角！因此，请继续自己学习和研究。