

Chapter 9: Parent-Child Relationships

By Tim Condon

第5章“Fluent and Persisting Models”介绍了模型的概念。在本章中，您将学习如何在两个模型之间建立父子关系。您还将了解这些关系的目的，如何在Vapor中对它们进行建模以及如何将它们与路由一起使用。

注意：本章要求您已设置并配置PostgreSQL。按照第6章“Configuring a Database”中的步骤，在Docker中设置PostgreSQL并配置Vapor应用程序。

父子关系

父子关系描述了一个模型具有一个或多个模型的“所有权”的关系。它们也被称为一对一和一对多关系。

例如，如果您设计人与宠物之间关系的模型，一个人可以拥有一只或多只宠物。宠物只能拥有一个主人。在TIL应用程序中，用户将创建缩略词。用户（父级）可以有許多缩略词，缩略词（子级）只能由一个用户创建。

创建用户

为User类创建一个新文件，为UsersController创建一个新文件。在终端中，键入：

```
# 1
cd ~/vapor/TILApp
# 2
touch Sources/App/Models/User.swift
# 3
touch Sources/App/Controllers/UsersController.swift
# 4
vapor xcode -y
```

这是它的作用：

1. 切换到TIL应用程序所在的目录。
2. 创建一个新文件**User.swift**。
3. 创建一个新文件**UsersController.swift**。
4. 重新生成Xcode项目并打开它。

用户模型

在Xcode中，打开**User.swift**并为用户创建基本模型：

```
import Foundation
import Vapor
import FluentPostgreSQL

final class User: Codable {
    var id: UUID?
    var name: String
    var username: String

    init(name: String, username: String) {
        self.name = name
        self.username = username
    }
}
```

该模型包含两个String属性来保存用户名和可选名称。它还包含一个可选的id属性，用于存储数据库保存时指定的模型的ID。请注意，与Acronym模型不同，这次ID是UUID。因此，您必须导入Foundation。

接下来，通过在文件末尾添加以下内容，使User模型遵循Fluent的模型协议：

```
extension User: PostgreSQLUUIDModel {}
```

使用FluentPostgreSQL模型帮助器使遵循Model协议变得简单。因为模型的id属性是UUID，所以必须使用PostgreSQLUUIDModel而不是PostgreSQLModel。接下来，在文件的底部，使User遵循Content、Migration和Parameter协议：

```
extension User: Content {}  
extension User: Migration {}  
extension User: Parameter {}
```

最后，打开**configure.swift**并在migrations.add(model: Acronym.self, database: .psql)之后将User模型添加到migration列表：

```
migrations.add(model: User.self, database: .psql)
```

这会将新模型添加到migrations中，以便Fluent在下一个应用程序启动时准备好数据库中的表。

用户控制器

打开**UsersController.swift**并创建一个可以创建用户的新控制器：

```
import Vapor  
  
// 1  
struct UsersController: RouteCollection {  
    // 2  
    func boot(router: Router) throws {  
        // 3  
        let usersRoute = router.grouped("api", "users")  
        // 4  
        usersRoute.post(User.self, use: createHandler)  
    }  
  
    // 5  
    func createHandler(  
        _ req: Request,  
        user: User  
    ) throws -> Future<User> {  
        // 6  
        return user.save(on: req)  
    }  
}
```

这应该看起来很熟悉了;这是它的作用:

1. 定义遵循RouteCollection协议的新类型UsersController。
2. 根据RouteCollection的要求实现boot(router:;)。
3. 为路径/**api/users**创建新的路由组。
4. 注册createHandler(_ :user:)来处理对/**api/users**的POST请求。这使用POST辅助方法将请求body解码为用户对象。
5. 定义路由处理函数。
6. 保存从请求中解码的用户。

最后, 打开**routes.swift**并将以下内容添加到routes(:)末尾:

```
// 1
let usersController = UsersController()
// 2
try router.register(collection: usersController)
```

这是它的作用:

1. 创建UsersController实例。
2. 向路由器注册新的控制器实例以连接路由。

再次打开**UsersController.swift**并将以下内容添加到UsersController的末尾。 这些函数返回所有用户和单个用户的列表:

```
// 1
func getAllHandler(_ req: Request) throws -> Future<[User]> {
    // 2
    return User.query(on: req).all()
}

// 3
func getHandler(_ req: Request) throws -> Future<User> {
    // 4
    return try req.parameters.next(User.self)
}
```

这是它的作用:

1. 定义一个返回Future<[User]>的新路由处理程序getAllHandler(:)。
2. 使用Fluent查询返回所有用户。
3. 定义一个返回Future<User>的新路由处理程序getHandler(:)。
4. 返回请求参数指定的用户。

在boot(router:)末尾注册这两个路由处理程序：

```
// 1
usersRoute.get(use: getAllHandler)
// 2
usersRoute.get(User.parameter, use: getHandler)
```

这是它的作用：

1. 注册getAllHandler(_)以处理对 **/api/users/**的GET请求。
2. 注册getHandler(_)以处理对 **/api/users/<USER ID>**的GET请求。

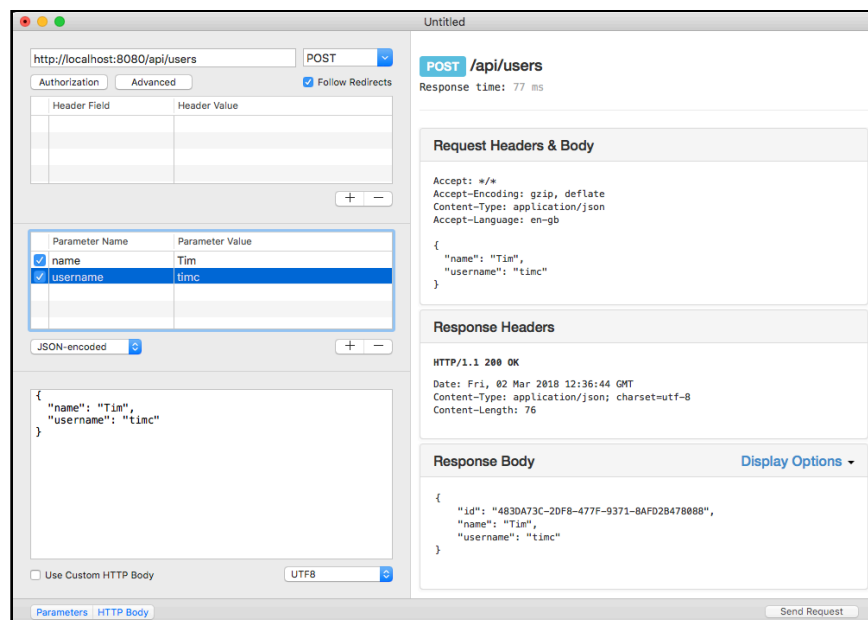
构建并运行应用程序，然后在RESTed中创建新请求。配置请求如下：

- **URL:** http://localhost:8080/api/users
- **method:** POST
- **Parameter encoding:** JSON-encoded

添加两个带有名称和值的参数：

- **name:** your name
- **username:** a username of your choice

发送请求，您将在响应中看到已保存的用户：



建立关系

在Vapor中父子关系的建模与数据库如何建模关系是相匹配的。由于用户拥有每个缩略词，因此您可以在缩略词中添加对用户的引用。这允许Fluent能更加有效地搜索数据库。

要获取用户的所有缩略词，请检索包含该用户引用的所有缩略词。要获取缩略词的用户，请使用该缩略词中的用户引用。

打开 **Acronym.swift** 并在`var long: String`之后添加一个新属性：

```
var userID: User.ID
```

这会向模型添加`User.ID`类型的属性。这是由`PostgreSQLUUIDModel`定义的`typealias`，它解析为`UUID`。请注意，此类型不是可选的，因此缩略词必须包含用户。

用以下内容替换初始化函数以反映这一点：

```
init(short: String, long: String, userID: User.ID) {  
    self.short = short  
    self.long = long  
    self.userID = userID  
}
```

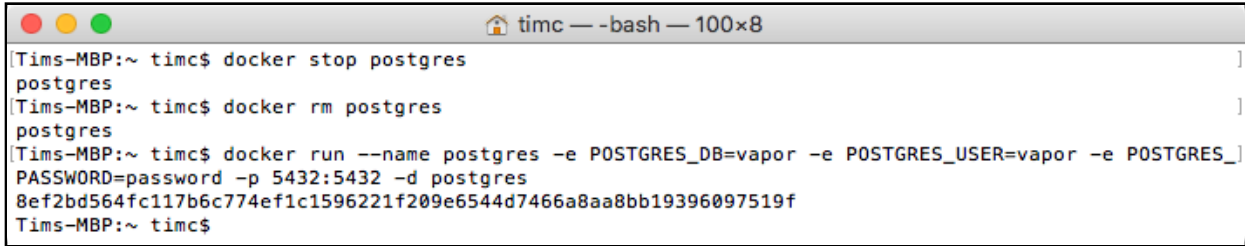
这就是建立关系所需要做的一切！在运行应用程序之前，需要重置数据库。Fluent已经运行了Acronym migration，但现在该表有一个新列。要将新列添加到表中，必须删除数据库，以便Fluent再次运行migration。在Xcode中停止应用程序，然后在终端中输入：

```
# 1  
docker stop postgres  
# 2  
docker rm postgres  
# 3  
docker run --name postgres -e POSTGRES_DB=vapor \  
-e POSTGRES_USER=vapor -e POSTGRES_PASSWORD=password \  
-p 5432:5432 -d postgres
```

这是它的作用：

1. 停止正在运行的Docker容器**postgres**。这是当前运行数据库的容器。
2. 移除Docker容器**postgres**，以删除任何现有数据。

3. 启动一个运行PostgreSQL的新Docker容器。有关更多信息，请参阅第6章的“Configuring a Database”。



```
timc — -bash — 100x8
[Tims-MBP:~ timc$ docker stop postgres
postgres
[Tims-MBP:~ timc$ docker rm postgres
postgres
[Tims-MBP:~ timc$ docker run --name postgres -e POSTGRES_DB=vapor -e POSTGRES_USER=vapor -e POSTGRES_
PASSWORD=password -p 5432:5432 -d postgres
8ef2bd564fc117b6c774ef1c1596221f209e6544d7466a8aa8bb19396097519f
Tims-MBP:~ timc$
```

注意：新migrations也可以更改表，以便在更改模型时不会丢失生产数据。第23章，“Database/API Versioning and Migration”涵盖了这一点。

在Xcode中构建并运行应用程序，执行migrations。打开RESTed并按照本章前面的步骤创建用户。确保复制返回的ID。

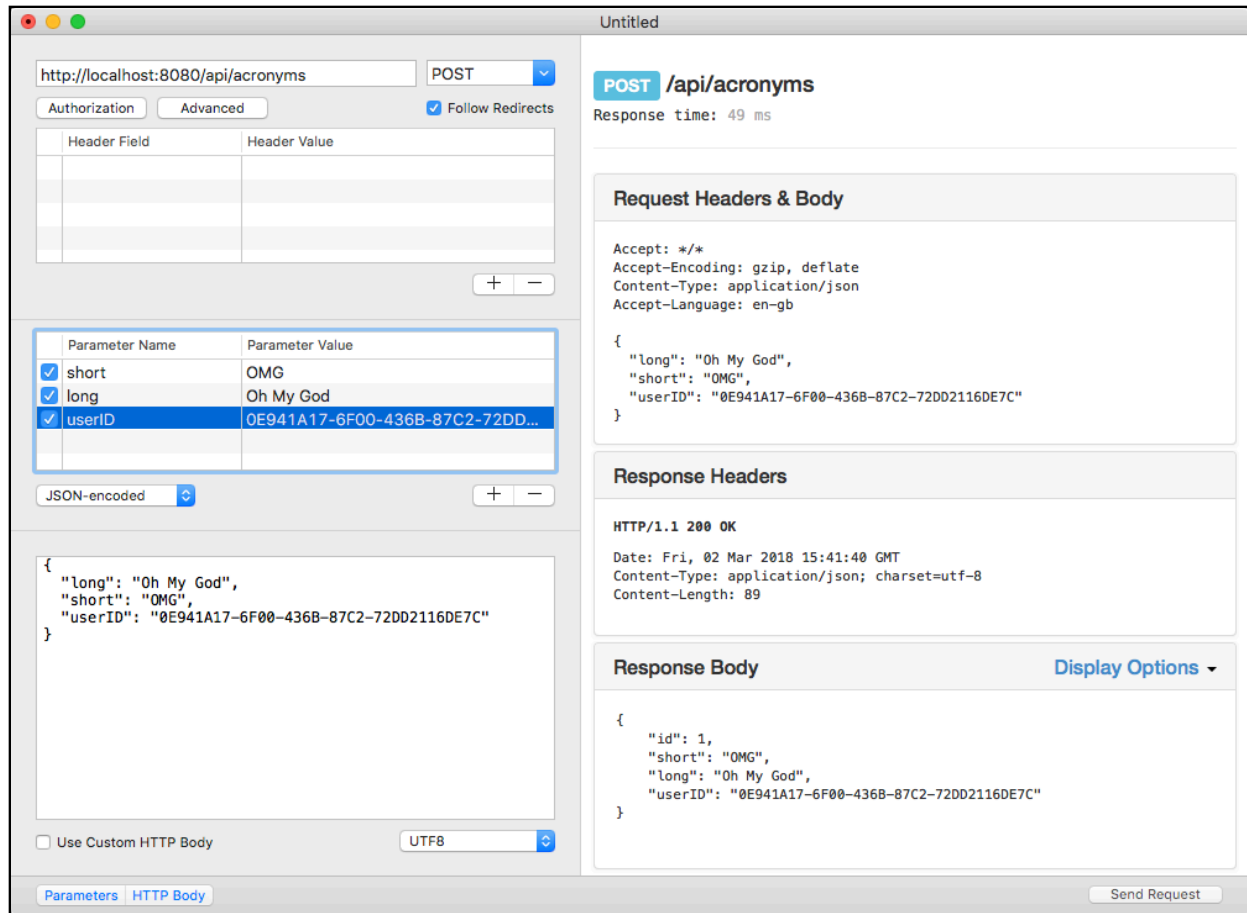
在RESTed中创建一个新请求并按如下方式配置它：

- **URL:** `http://localhost:8080/api/acronyms`
- **method:** POST
- **Parameter encoding:** JSON-encoded

添加三个带名称和值的参数：

- **short:** OMG
- **long:** Oh My God
- **userID:** the ID you copied earlier

单击发送请求。您的应用程序使用指定的用户创建缩略词：



最后，打开 `AcronymsController.swift` 并用以下内容替换 `updateHandler(_)`，以支持 Acronym 上的新属性：

```
func updateHandler(_ req: Request) throws -> Future<Acronym> {
    return try flatMap(
        to: Acronym.self,
        req.parameters.next(Acronym.self),
        req.content.decode(Acronym.self)
    ) { acronym, updatedAcronym in
        acronym.short = updatedAcronym.short
        acronym.long = updatedAcronym.long
        acronym.userID = updatedAcronym.userID
        return acronym.save(on: req)
    }
}
```

这将使用请求中提供的新值更新缩略词的属性。

查询关系

用户和缩略词现在是父子关系相关联。但是，在能查询这些关系之前，这并不是很有用。再次，Fluent让这一切变得简单。

获取父级

打开**Acronym.swift**并在文件底部添加一个扩展以获取缩略词的父级：

```
extension Acronym {  
    // 1  
    var user: Parent<Acronym, User> {  
        // 2  
        return parent(\.userID)  
    }  
}
```

这是它的作用：

1. 向Acronym添加一个计算属性，以获取缩略词所有者的User对象。这将返回Fluent的泛型Parent类型。
2. 使用Fluent的parent(⌵) 函数来检索父级。这是引用缩略词里的用户key path。

打开**AcronymsController.swift**并在sortedHandler(⌵)之后添加一个新的路由处理程序：

```
// 1  
func getUserHandler(_ req: Request) throws -> Future<User> {  
    // 2  
    return try req  
        .parameters.next(Acronym.self)  
        .flatMap(to: User.self) { acronym in  
            // 3  
            acronym.user.get(on: req)  
        }  
}
```

以下是此路由处理程序的作用：

1. 定义一个返回Future<User>的新路由处理程序getUserHandler(⌵)。
2. 获取请求参数中指定的缩略词并解包返回的future。
3. 使用上面创建的新计算属性来获取缩略词的所有者。

在 `boot(router:)` 末尾注册路由处理程序：

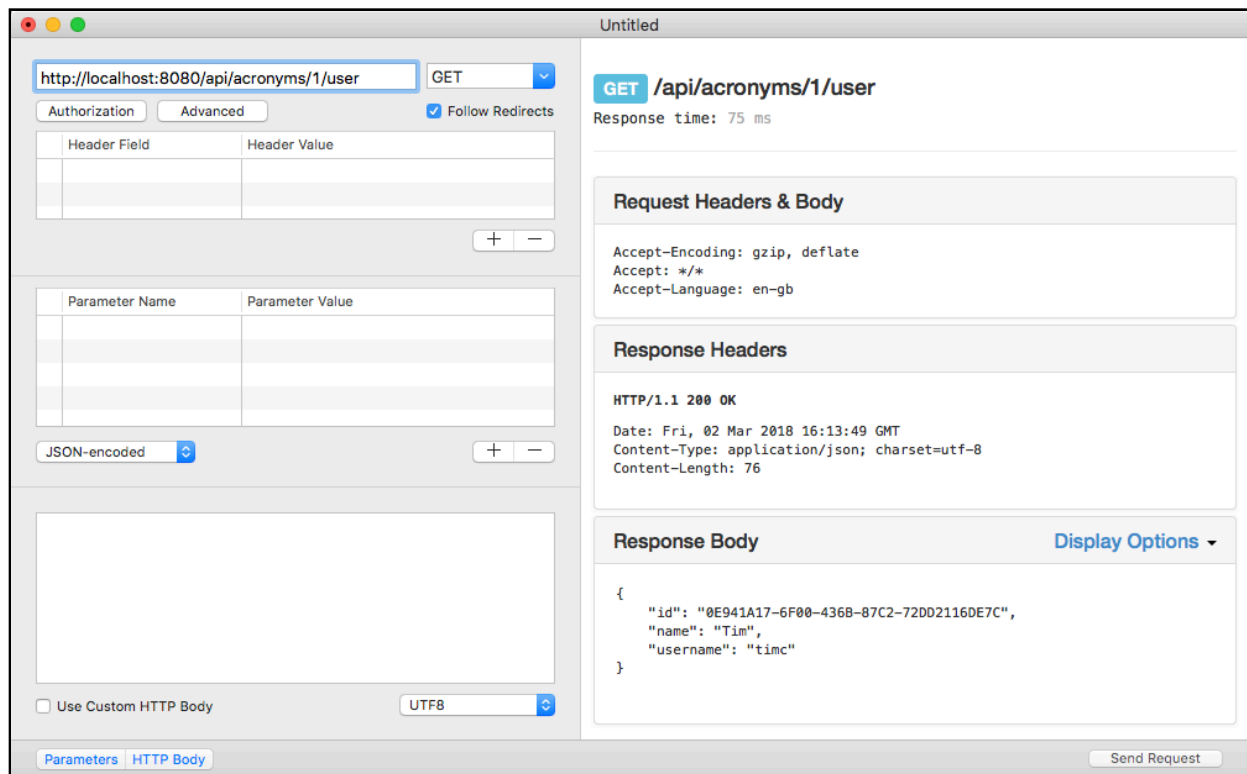
```
acronymsRoutes.get(  
    Acronym.parameter, "user",  
    use: getUserHandler)
```

这将 `/api/acronyms/<ACRONYM ID>/user` 的 HTTP GET 请求连接到 `getUserHandler(_:)`。

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/api/acronyms/1/user`
- **method:** GET

发送请求，您将看到响应返回缩略词的用户：



获取子级

获取模型的子级遵循类似的模式。打开 `User.swift` 并在文件底部添加一个扩展以获取用户的缩略词：

```
extension User {  
    // 1  
    var acronyms: Children<User, Acronym> {
```

```
        // 2
        return children(\.userID)
    }
}
```

这是它的作用：

1. 向User添加一个计算属性，以获取用户的缩略词。这将返回Fluent的泛型Children类型。
2. 使用Fluent的children(⋮)函数来检索子级。这是引用缩略词里的用户key path。

打开**UsersController.swift**并在getHandler(⋮)之后添加一个新的路由处理程序：

```
// 1
func getAcronymsHandler(_ req: Request)
    throws -> Future<[Acronym]> {
    // 2
    return try req
        .parameters.next(User.self)
        .flatMap(to: [Acronym].self) { user in
        // 3
        try user.acronyms.query(on: req).all()
    }
}
```

以下是此路由处理程序的作用：

1. 定义一个新的路由处理程序getAcronymsHandler(⋮)，它返回Future<[Acronym]>。
2. 获取请求参数中指定的用户并解包返回的future。
3. 使用上面创建的新计算属性，用fluent查询获取缩略词，以返回所有缩略词。

在boot(router:⋮)末尾注册路由处理程序：

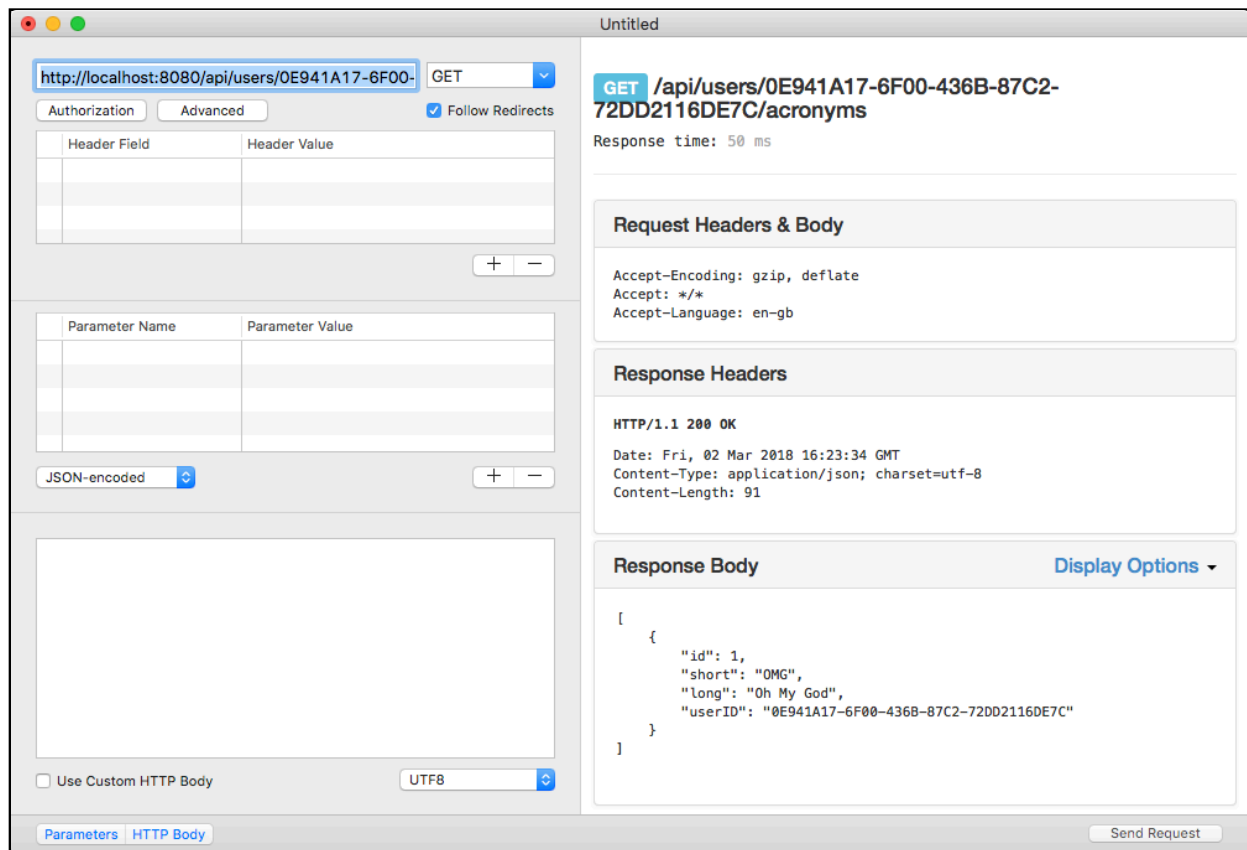
```
usersRoute.get(
    User.parameter, "acronyms",
    use: getAcronymsHandler)
```

这将/api/users/<USER ID>/acronyms的HTTP GET请求连接到getAcronymsHandler(⋮)。

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/api/users/<ID of your user>/acronyms`
- **method:** GET

发送请求，您将看到响应返回用户的缩略词：



外键约束

外键约束描述了两个表之间的链接。它们经常用于验证。目前，数据库中的用户表与缩略词表之间没有任何关联。Fluent是唯一了解链接的东西。

使用外键约束有许多好处：

- 它确保您无法为不存在的用户创建缩略词。
- 在删除所有缩略词之前，您无法删除用户。
- 在删除缩略词表之前，您无法删除用户表。

在migration中设置外键约束。打开**Acronym.swift**，并删除以下Migration扩展：

```
extension Acronym: Migration {}
```

接下来，在文件底部添加以下扩展：

```
// 1
extension Acronym: Migration {
    // 2
    static func prepare(
        on connection: PostgreSQLConnection
    ) -> Future<Void> {
        // 3
        return Database.create(self, on: connection) { builder in
            // 4
            try addProperties(to: builder)
            // 5
            builder.reference(from: \.userID, to: \User.id)
        }
    }
}
```

这是它的作用：

1. 使Acronym再次遵循Migration协议。
2. 根据Migration的要求实现prepare(on:)。这会覆盖默认实现。
3. 在数据库中为Acronym创建表。
4. 使用addProperties(to:)将所有字段添加到数据库。这意味着您无需手动添加每个列。
5. 在Acronym上的userID属性和User上的id属性之间添加引用。这将设置两个表之间的外键约束。

最后，因为您要将缩略词的userID属性链接到User表，所以必须首先创建User表。在**configure.swift**中，将User migration移至Acronym migration之前：

```
migrations.add(model: User.self, database: .psql)
migrations.add(model: Acronym.self, database: .psql)
```

这可确保Fluent以正确的顺序创建表。

在Xcode中停止应用程序并按照之前的步骤删除数据库。

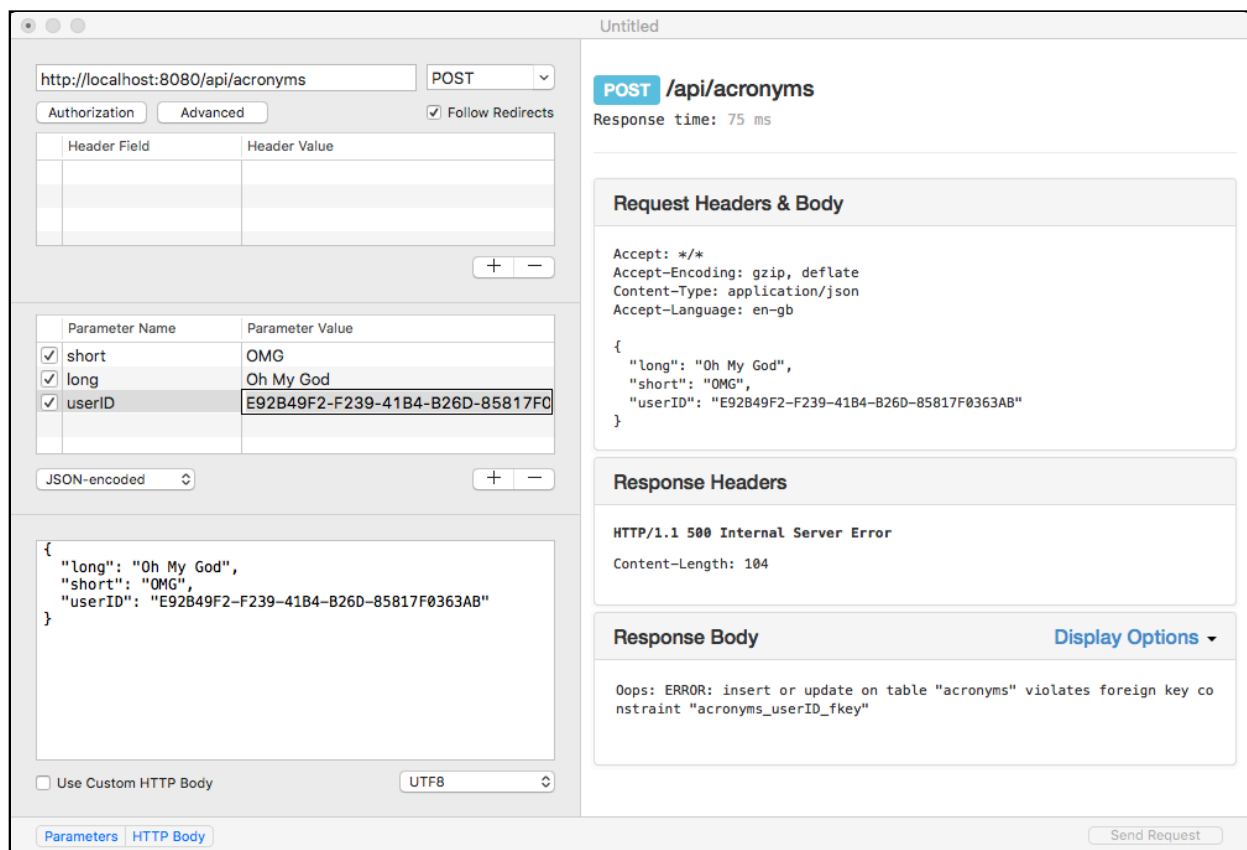
构建并运行应用程序，然后在RESTed中创建新请求。配置请求如下：

- **URL:** `http://localhost:8080/api/acronyms/`
- **method:** POST
- **Parameter encoding:** JSON-encoded

添加三个带名称和值的参数：

- **short:** OMG
- **long:** Oh My God
- **userID:** E92B49F2-F239-41B4-B26D-85817F0363AB

这是一个有效的UUID字符串，但由于数据库为空，因此不引用任何用户。发送请求；你会收到一个错误，说有一个外键约束违规：



像之前一样创建用户并复制ID。再次发送创建缩略词请求，这次使用有效ID。这次应用程序创建缩略词没有任何错误。

然后去哪儿？

在本章中，您学习了如何使用Fluent在Vapor中实现父子关系。这允许您在数据库的模型之间开始创建复杂的关系。下一章将介绍数据库中的另一种关系：兄弟关系。