Chapter 26: Database/API Versioning & Migration

By Jonas Schwartz

在本书的前三节中，无论何时对模型进行更改，都必须删除数据库并重新开始。没有任何数据时没问题。但是有数据时或将项目移至生产阶段后，您将无法再删除数据库。您应该要做的是修改您的数据库，在Vapor中，使用**migrations**完成。

> 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。

在本章中，您将使用migrations对TILApp进行两次修改。首先，您将向User添加一个新字段以包含Twitter。其次，您将确保类别是唯一的。最后，您将修改应用程序，以便您的应用程序仅在开发或测试模式下运行时才创建管理员用户。

> 注意：本章提供的TILApp版本的示例文件不是第3节末尾的完整版本。相反，它是一个简化的早期迭代版。如果您愿意，可以将这些更改集成到项目的工作副本中。

修改表

修改现有数据库始终是一项有风险的业务。您已经拥有了不想丢失的数据，因此删除整个数据库并不是一个可行的解决方案。同时，您不能简单地在现有表中添加或删除属性，因为所有数据都纠缠在一个大的连接和关系网络中。

相反，您使用Vapor的Migration协议尝试您的修改。这使您可以谨慎地尝试您的修改，同时如果它们不能按预期工作，仍然具有恢复选项。

> 修改生产数据库始终是一个复杂的过程。在生产中进行任何修改之前，您必须确保已对其进行了正确地测试。如果您有很多重要数据，最好在修改数据库之前进行备份。

为了使代码保持整洁且便于按时间顺序查看更改，您应该创建一个包含所有migrations的目录。每次migration都应该有自己的文件。对于文件名，请使用一致且有用的命名方案，例如：**YY-MM-DD-FriendlyName.swift**。这使您可以一目了然地查看数据库的版本。

编写migrations

当Migration用于更新现有模型时，通常将其编写为struct。当然，这个struct必须遵循Migration协议。Migration需要您提供三件事：

```
typealias Database: Fluent.Database

static func prepare(
  on connection: Database.Connection) -> Future<Void>

static func revert(
  on connection: Database.Connection) -> Future<Void>
```

Typealias Database

首先，您必须指定可以运行migration的数据库类型。Migrations需要数据库连接才能正常工作，因为它们必须能够查询MigrationLog模型。如果无法访问MigrationLog，migration将失败，在最坏的情况下，会中断您的应用程序。

Prepare方法

prepare(on:)包含migration对数据库的更改。它通常是两种选择之一：

- 创建一个新表
- 通过添加新属性修改现有表。

这是一个向数据库添加新模型的示例：

```
static func prepare(
  on connection: PostgreSQLConnection) -> Future<Void> {
  // 1
  return Database.create(
    NewTestUser.self,
    on: connection) { builder in
      // 2
      builder.field(for: \.id, isIdentifier: true)
  }
}
```

1. 您可以指定要执行的操作和要使用的模型。如果要向数据库添加新的Model类型，则使用create(_:on:closure:)。如果要向现有Model类型添加字段，请使用update(_:on:closure:)。此示例使用create(_:on:closure:)创建具有字段id的新模型。

2. 接下来，指定一个闭包，该闭包接受模型的SchemaBuilder并执行实际修改。您可以在构建器上调用field(for:isIdentifier:)来描述要添加到模型中的每个字段。通常，您不需要包含字段的类型，因为Fluent可以推断出最适合使用的类型。

Revert方法

revert(on:)与prepare(on:)相反。它的工作是撤消prepare(on:)做的任何事。如果在prepare(on:)中使用create(_:on:closure:)，则在revert(on:)里使用delete(_:on:)。如果你使用update(_:on:closure:)来添加一个字段，则在revert(on:)里还使用它来删除字段，在它的闭包里用deleteField(for:)删除具体字段。

这是一个与之前看到的prepare(on:)配对的示例：

```
static func revert(
  on connection: PostgreSQLConnection) -> Future<Void> {
    return Database.delete(NewTestUser.self,
                           on: connection)
}
```

同样，您指定要执行的操作和要还原的模型。由于您使用create(_:on:closure:)来添加模型，因此在此处使用delete(_:on:)。

使用--revert选项启动应用程序时，将执行此方法。

添加用户的Twitter

为了演示现有数据库的migration过程，您将添加对收集和存储用户的Twitter的支持。首先，您需要创建一个新文件夹来保存所有migrations，并创建一个新文件来保存AddTwitterToUser migration。在终端中，导航到保存TILApp项目的目录并输入：

```
# 1
mkdir Sources/App/Migrations
# 2
touch Sources/App/Migrations/18-06-05-AddTwitterToUser.swift
# 3
vapor xcode -y
```

这是它的作用：

1. 在**App**模块中创建一个新目录**Migrations**。
2. 在刚创建的**Migrations**目录中创建一个新文件**18-06-05-AddTwitterToUser.swift**。
3. 重新生成Xcode项目以将新文件添加到App target。

接下来，在Xcode中打开**User.swift**并将以下属性添加到User的var password: String下面：

```
var twitterURL: String?
```

这会添加String?类型的属性到模型。您将其声明为可选字符串，因为您的现有用户没有该属性，将来的用户也不一定拥有Twitter帐户。

接下来，使用以下内容替换初始化程序以支持新属性：

```
init(name: String,
     username: String,
     password: String,
     twitterURL: String? = nil) {
  self.name = name
  self.username = username
  self.password = password
  self.twitterURL = twitterURL
}
```

创建migration

使用migration将新属性添加到现有模型时，重要的是修改初始migration，以便仅添加原始字段。默认情况下，prepare(on:)会添加它在模型中找到的每个属性。如果由于某种原因（例如运行测试软件）还原整个数据库，允许它在初始migration中继续添加所有字段，将会导致新migration失败。

在User: Migration扩展中找到现有的`prepare(on:)`，并用以下内容替换`try addProperties(to: builder)`：

```
builder.field(for: \.id, isIdentifier: true)
builder.field(for: \.name)
builder.field(for: \.username)
builder.field(for: \.password)
```

这会手动将现有属性（不包括新的twitterURL）添加到数据库。

接下来，打开**18-06-05-AddTwitterToUser.swift**并添加以下内容以创建migration，为将新的twitterURL字段添加到模型中：

```
import FluentPostgreSQL
import Vapor

// 1
struct AddTwitterURLToUser: Migration {

  // 2
  typealias Database = PostgreSQLDatabase

  // 3
  static func prepare(
    on connection: PostgreSQLConnection
  ) -> Future<Void> {
    // 4
    return Database.update(
      User.self,
      on: connection
    ) { builder in
      // 5
      builder.field(for: \.twitterURL)
    }
  }

  // 6
  static func revert(
    on connection: PostgreSQLConnection
  ) -> Future<Void> {
    // 7
    return Database.update(
      User.self,
```

```
      on: connection
    ) { builder in
      // 8
      builder.deleteField(for: \.twitterURL)
    }
  }
}
```

这是它的作用：

1. 定义遵循Migration协议的新类型AddTwitterURLToUser。
2. 根据Migration的要求，使用typealias定义数据库类型。
3. 定义所需的prepare(on:)。
4. 由于用户已存在于您的数据库中，因此请使用update(_:on:closure:)来修改数据库。
5. 在闭包内，使用field(for:)添加与key path \.twitterURL对应的新字段。
6. 定义所需的`revert(on:)`。
7. 由于您正在修改现有Model，因此再次使用update(_:on:closure:)以删除新字段。
8. 在闭包内，使用deleteField(for:)删除与key path \.twitterURL对应的字段。

现在打开**configure.swift**并将AddTwitterURLToUser注册为其中一个migration。

由于migrations是按顺序执行的，因此它必须在列表中的现有migrations之后进行。在services.register(migrations)之前立即添加以下内容：

```
migrations.add(
  migration: AddTwitterURLToUser.self,
  database: .psql)
```

下次启动应用程序时，新属性将添加到User。与AdminUser一样，您应该使用add(migration:database:)来注册migration，因为它不是完整的模型。构建并运行您的应用程序；你应该能够在你的表中看到新属性。

在您的开发计算机上，您可以通过在终端中输入以下内容来查看表的属性：

```
docker exec -it postgres psql -U vapor
\d "User"
\q
```

API版本控制

您已将模型更改为包含用户的Twitter，但您尚未更改现有的API。虽然您可以简单地更新API以包含Twitter，但这可能会破坏API的现有使用者。相反，您可以创建一个新的API版本来返回用户及其Twitter。

为此，首先打开**User.swift**并在Public之后添加以下定义：

```
final class PublicV2: Codable {
  var id: UUID?
  var name: String
  var username: String
  var twitterURL: String?

  init(id: UUID?,
       name: String,
       username: String,
       twitterURL: String? = nil) {
    self.id = id
    self.name = name
    self.username = username
    self.twitterURL = twitterURL
  }
}
```

这将创建一个包含twitterURL的新PublicV2类。接下来，将以下内容添加到文件末尾，以使此新类遵循Content协议：

```
extension User.PublicV2: Content {}
```

接下来，为版本2 API再创建个转换函数。在User的扩展中将以下内容添加到convertToPublic()之后：

```
func convertToPublicV2() -> User.PublicV2 {
  return User.PublicV2(
    id: id,
    name: name,
    username: username,
    twitterURL: twitterURL)
}
```

现在，在扩展中的convertToPublic()之后添加以下内容，以返回Future：

```
func convertToPublicV2() -> Future<User.PublicV2> {
  return self.map(to: User.PublicV2.self) { user in
    return user.convertToPublicV2()
  }
}
```

最后，打开**UsersController.swift**并在getHandler(_:)之后添加以下内容：

```
 // 1
func getV2Handler(_ req: Request) throws
    -> Future<User.PublicV2> {
  // 2
  return try req.parameters.next(User.self).convertToPublicV2()
}
```

这个方法就像getHandler(_:)有两个变化：

1. 返回User.PublicV2

2. 调用convertToPublicV2()以生成正确的返回项。

现在，在boot(router:)末尾添加以下内容：

```
// API Version 2 Routes
 // 1
let usersV2Route = router.grouped("api", "v2", "users")
 // 2
usersV2Route.get(User.parameter, use: getV2Handler)
```

这是它的作用：

1. 添加将在**/api/v2/users**上解析的新API组。

2. 将GET请求连接到getV2Handler()。

现在，您有了一个新的网络端点来获取用户，并在API中使用**v2**返回twitterURL。

> 注意：对于更复杂的API修订，您应该创建新的控制器来处理新的API版本。这将简化您对代码的理解，并使其更易于维护。

更新网站

您的应用程序现在已经具备存储用户Twitter所需的全部功能，并且API已完成。您需要更新网站，以允许新用户在注册过程中提供Twitter地址。

打开**register.leaf**并在**name**的表单组之后添加以下内容：

```
<div class="form-group">
  <label for="twitterURL">Twitter handle</label>
  <input type="text" name="twitterURL" class="form-control"
   id="twitterURL"/>
</div>
```

这会在注册表单上为Twitter添加一个字段。接下来，打开**user.leaf**并将<h2>#(user.username)</h2>替换为以下内容：

```
<h2>#(user.username)
#if(user.twitterURL) {
 - #(user.twitterURL)
}
</h2>
```

这将在用户信息页面上显示Twitter（如果存在）。最后，打开**WebsiteController.swift**并将以下内容添加到RegisterData的末尾：

```
let twitterURL: String?
```

这允许您的表单处理程序访问从浏览器发送的Twitter信息。在registerPostHandler(_:data:)中，替换

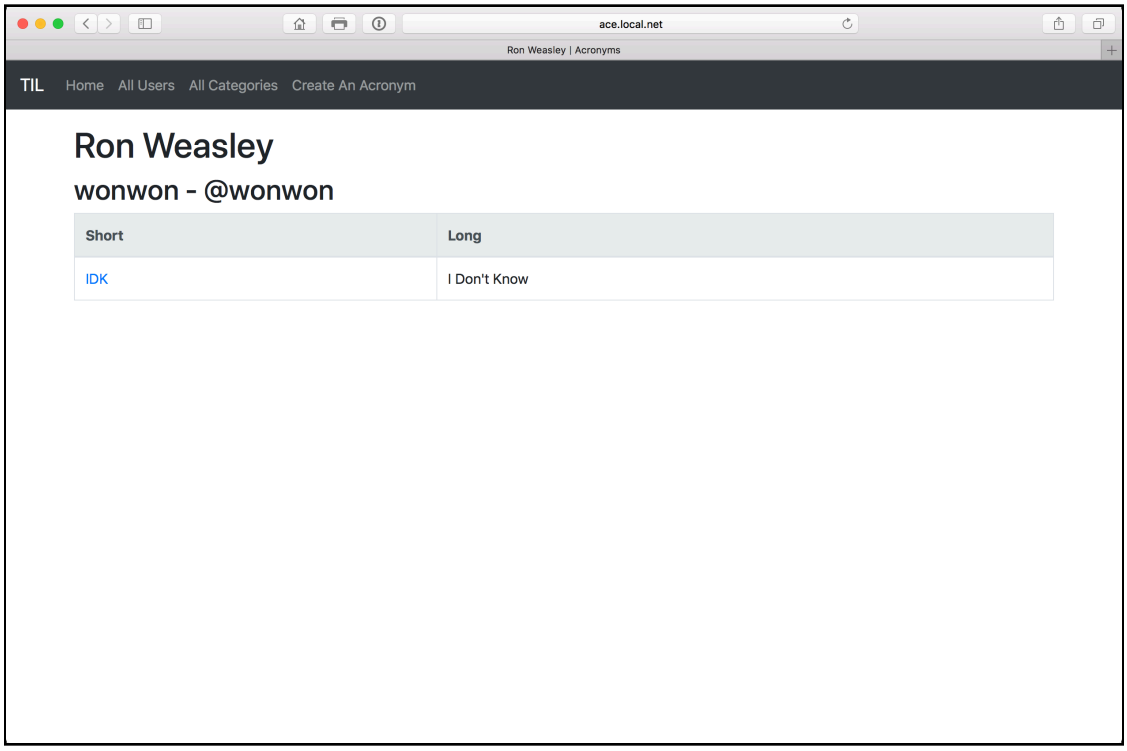
```
let user = User(
  name: data.name,
  username: data.username,
  password: password)
```

用以下内容：

```
var twitterURL: String?
if
  let twitter = data.twitterURL,
  !twitter.isEmpty {
    twitterURL = twitter
}
let user = User(
  name: data.name,
  username: data.username,
  password: password,
  twitterURL: twitterURL)
```

如果用户未提供Twitter，则您希望在数据库中存储nil而不是空字符串。

构建并运行。在浏览器中访问**http://localhost:8080/**，并注册一个新用户，提供 Twitter。然后访问用户的信息页面，查看结果！



使类别唯一

正如您需要用户名是唯一的一样，您确实希望类别名称也是唯一的。到目前为止，您为实现类别所做的一切都使得无法创建重复项，但您也希望在数据库中强制执行重复项。是时候创建一个Migration，保证重复的类别名称不能插入数据库中。

首先，在**Migrations**目录中创建一个新文件。在终端中，输入：

```
touch Sources/App/Migrations/18-06-05-MakeCategoriesUnique.swift
vapor xcode -y
```

这将创建一个新文件以包含新的Migration，并重新生成您的Xcode项目。

在Xcode中，打开**18-06-05-MakeCategoriesUnique.swift**并输入以下内容：

```
import FluentPostgreSQL
import Vapor
// 1
struct MakeCategoriesUnique: Migration {
  // 2
  typealias Database = PostgreSQLDatabase
  // 3
  static func prepare(
    on connection: PostgreSQLConnection
  ) -> Future<Void> {
    // 4
    return Database.update(
      Category.self,
      on: connection
    ) { builder in
      // 5
      builder.unique(on: \.name)
    }
  }
  // 6
  static func revert(
    on connection: PostgreSQLConnection
  ) -> Future<Void> {
    // 7
    return Database.update(
      Category.self,
      on: connection
    ) { builder in
      // 8
      builder.deleteUnique(from: \.name)
    }
  }
}
```

1. 定义遵循Migration协议的新类型MakeCategoriesUnique。
2. 根据Migration的要求，使用typealias定义数据库类型。
3. 定义所需的`prepare(on:)`。
4. 由于Category已存在您的数据库中，因此请使用update(_:on:closure:)来修改数据库。
5. 在闭包内，使用unique(on:)添加与key path \.name对应的新唯一索引。
6. 定义所需的`revert(on:)`。

7. 由于您正在修改现有Model，因此再次使用update(_:on:closure:)删除新索引。

8. 在闭包内，使用deleteUnique(from:)删除对应于key path \.name的索引。

最后，打开configure.swift并将MakeCategoriesUnique注册为其中一个migration。在services.register(migrations)之前立即添加以下内容：

```
migrations.add(
  migration: MakeCategoriesUnique.self,
  database: .psql)
```

构建并运行；在控制台中观察新的migration

Seeding based on environment

在第18章“API Authentication, Part 1”中，您在数据库中输入了一个管理员用户。如前所述，您不应该使用“password”作为管理员密码。但是，当您仍在开发中并且只需要一个虚拟帐户进行本地测试时，它会更容易。确保不在生产中添加此用户的一种方法是在添加migration之前检测您的环境。在**configure.swift**中替换：

```
migrations.add(migration: AdminUSer.self, database: .psql)
```

用以下内容：

```
switch env {
case .development, .testing:
  migrations.add(migration: AdminUser.self, database: .psql)
default:
  break
}
```

现在，如果应用程序位于开发（默认）或测试环境中，则**AdminUser**仅添加到migrations中。如果环境是生产环境，则不会添加到migrations。当然，如果您仍希望在生产环境中拥有一个具有随机密码的管理员。在这种情况下，您可以在**AdminUser**中打开环境开关，也可以创建两个版本，一个用于开发，另一个用于生产。

然后去哪儿?

在本章中，您学习了在应用程序进入生产后如何使用migrations修改数据库。您了解了如何向User添加额外属性 - twitterUrl，如何还原此更新以及如何强制类别名称的唯一性。最后，您了解了如何在**configure.swift**中打开您的环境开关，从而允许您从生产环境中排除migrations。

您可以在 https://docs.vapor.codes/3.0/fluent/migrations/ 上的Vapor文档中了解有关migrations的更多信息。