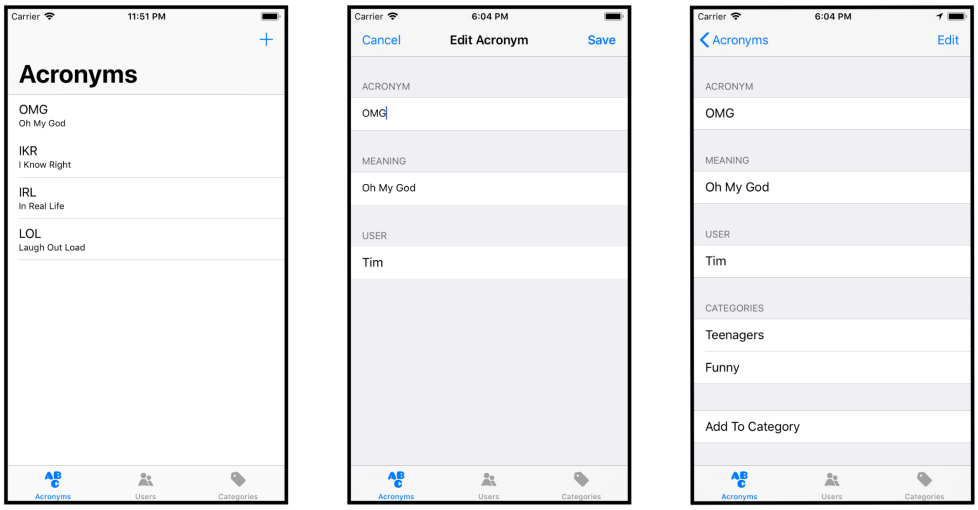
Chapter 12: Creating a Simple iPhone App, Part 1

By Tim Condon

在前面的章节中，您创建了一个API并使用RESTed与它进行了交互。但是，用户希望使用TIL更好一点！接下来的两章将向您展示如何构建一个与API交互的简单iOS应用程序。在本章中，您将学习如何创建不同的模型并从数据库中获取模型。

在这两章的最后，您将拥有一个iOS应用程序，可以完成您到目前为止所学到的一切。它看起来类似于以下内容：



Getting started

要开始工作，请下载本章的材料。在终端中，转到下载资料的目录并输入：

```
cd TILApp
vapor build
vapor run
```

这将构建并运行iOS应用程序将要访问的TIL应用程序。如果您愿意，可以使用现有的TIL应用程序。

> 注意：这要求数据库的Docker容器正在运行。有关说明，请参见第6章“Configuring a Database”。

接下来，打开**TILiOS**项目。 **TILiOS**包含一个与TIL API交互的框架应用程序。它是一个带有三个选项卡的标签栏应用程序：

- **Acronyms**: 查看所有缩略词、查看缩略词的详细信息和添加缩略词。
- **Users**: 查看所有用户和创建用户。
- **Categories**: 查看所有类别和创建类别。

该项目包含几个空表视图控制器，可以配置为显示来自TIL API的数据。

看看项目中的**Models**组；它提供了三个模型类：

- `Acronym`
- `User`
- `Category`

您可能看到代码很熟悉 - 这些是API应用程序中的相同模型！这显示了对客户端和服务器使用相同语言的强大功能。甚至可以创建一个单独的模块，两个项目都使用，因此您不必重复代码。

查看缩略词

第一个选项卡的表显示所有缩略词。在**Utilities**组中创建一个名为**ResourceRequest.swift**的新**Swift**文件。打开文件并创建一个枚举来表示调用TILApp API的结果：

```
enum GetResourcesRequest<ResourceType> {
  // 1
  case success([ResourceType])
  // 2
  case failure
}
```

此枚举表示泛型资源类型，并提供两种情况：

1. 存储资源类型数组的成功情况。
2. 失败情况。

在GetResourcesRequest下面，创建一个类型来管理发出资源请求：

```
// 1
struct ResourceRequest<ResourceType>
  where ResourceType: Codable {

  // 2
  let baseURL = "http://localhost:8080/api/"
  let resourceURL: URL

  // 3
  init(resourcePath: String) {
    guard let resourceURL = URL(string: baseURL) else {
      fatalError()
    }
    self.resourceURL =
      resourceURL.appendingPathComponent(resourcePath)
  }

  // 4
  func getAll(
    completion: @escaping
      (GetResourcesRequest<ResourceType>) -> Void
  ) {
    // 5
    let dataTask = URLSession.shared
      .dataTask(with: resourceURL) { data, _, _ in
      // 6
        guard let jsonData = data else {
          completion(.failure)
          return
        }
        do {
          // 7
          let resources = try JSONDecoder()
```

```
            .decode([ResourceType].self,
                    from: jsonData)
          // 8
          completion(.success(resources))
        } catch {
          // 9
          completion(.failure)
        }
      }
      // 10
      dataTask.resume()
  }
}
```

这是它的作用：

1. 定义泛型ResourceRequest类型，其泛型参数必须遵循Codable协议。
2. 设置API的根URL。现在使用localhost。请注意，这需要禁用ATS。
3. 初始化特定资源的URL。
4. 定义一个函数以从API获取资源类型的所有值。将completion闭包作为参数。
5. 使用资源URL创建数据任务。
6. 确保响应返回一些数据。否则，使用.failure情况调用completion(_:)闭包。
7. 将响应数据解码为ResourceTypes数组。
8. 使用.success情况调用completion(_:)闭包并返回ResourceTypes数组。
9. 捕获任何错误并返回失败。
10. 启动dataTask。

打开**AcronymsTableViewController.swift**并在// MARK: - Properties下面添加以下内容：

```
// 1
var acronyms: [Acronym] = []
// 2
let acronymsRequest =
  ResourceRequest<Acronym>(resourcePath: "acronyms")
```

这是它的作用：

1. 声明一个缩略词数组。这些是列表显示的缩略词。

2. 为缩略词创建ResourceRequest。

获取缩略词

每当视图出现在屏幕上时，表视图控制器都会调用refresh(_:)。用以下内容替换refresh(_:)的实现：

```
// 1
acronymsRequest.getAll { [weak self] acronymResult in
  // 2
  DispatchQueue.main.async {
    sender?.endRefreshing()
  }

  switch acronymResult {
  // 3
  case .failure:
    ErrorPresenter.showError(
      message: "There was an error getting the acronyms",
      on: self)
  // 4
  case .success(let acronyms):
    DispatchQueue.main.async { [weak self] in
      guard let self = self else { return }
      self.acronyms = acronyms
      self.tableView.reloadData()
    }
  }
}
```

这是它的作用：

1. 调用getAll(completion:)以获取所有缩略词。这将在completion闭包中返回结果。

2. 请求完成后，在刷新控件上调用endRefreshing()。

3. 如果获取失败，使用ErrorPresenter实用程序显示带有相应错误消息的警报视图。

4. 如果获取成功，则用结果更新acronyms数组并重新加载列表。

显示缩略词

仍然在**AcronymsTableViewController.swift**中，更新tableView(_:numberOfRowsInSection:)以通过将return 1替换为以下内容来返回正确的缩略词数：

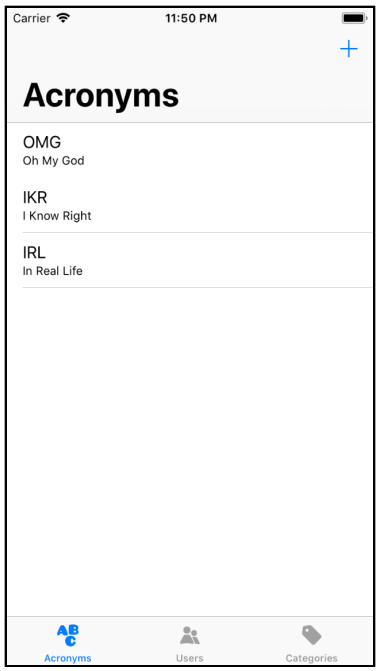
```
return acronyms.count
```

接下来，更新tableView(_:cellForRowAt:)以显示列表中的缩略词。在return cell之前添加以下内容：

```
let acronym = acronyms[indexPath.row]
cell.textLabel?.text = acronym.short
cell.detailTextLabel?.text = acronym.long
```

这将每个单元格的标题和副标题文本设置为缩写词的short和long属性。

构建并运行，您将看到您的列表填充了数据库中的缩略词：



查看用户

查看所有用户按照上述类似的模式进行。大多数视图控制器都已经建立好。打开**UsersTableViewController.swift**，并在var users: [User] = []下面添加以下内容：

```
let usersRequest = ResourceRequest<User>(resourcePath: "users")
```

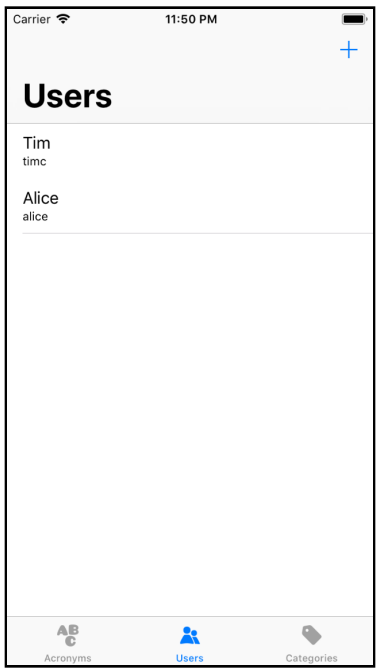
这将创建一个ResourceRequest以从API中获取用户。接下来，使用以下内容替换refresh(_:)的实现：

```
// 1
usersRequest.getAll { [weak self] result in
  // 2
  DispatchQueue.main.async {
    sender?.endRefreshing()
  }
  switch result {
  // 3
  case .failure:
    ErrorPresenter.showError(
      message: "There was an error getting the users",
      on: self)
  // 4
  case .success(let users):
    DispatchQueue.main.async { [weak self] in
      guard let self = self else { return }
      self.users = users
      self.tableView.reloadData()
    }
  }
}
```

这是它的作用：

1. 调用getAll(completion:)来获取所有用户。这将在completion闭包中返回结果。
2. 请求完成后，在刷新控件上调用endRefreshing()。
3. 如果获取失败，使用ErrorPresenter实用程序显示带有相应错误消息的警报视图。
4. 如果获取成功，则用结果更新users数组并重新加载列表。

构建并运行。转到“**Users**”选项卡，您将看到列表填充了数据库中的用户：



查看类别

按照类似的模式查看所有类别。打开**CategoriesTableViewController.swift**，并在var categories: [Category] = []下面添加以下内容：

```
let categoriesRequest =
  ResourceRequest<Category>(resourcePath: "categories")
```

这将设置ResourceRequest以从API获取类别。接下来，使用以下内容替换refresh(_:)的实现：

```
// 1
categoriesRequest.getAll { [weak self] result in
  // 2
  DispatchQueue.main.async {
    sender?.endRefreshing()
  }
  switch result {
  // 3
  case .failure:
    let message = "There was an error getting the categories"
    ErrorPresenter.showError(message: message, on: self)
  // 4
  case .success(let categories):
    DispatchQueue.main.async { [weak self] in
```

```
        guard let self = self else { return }
        self.categories = categories
        self.tableView.reloadData()
      }
    }
  }
```

这是它的作用：

1. 调用getAll(completion:)来获取所有类别。这将在completion闭包中返回结果。
2. 请求完成后，在刷新控件上调用endRefreshing()。
3. 如果获取失败，使用ErrorPresenter实用程序显示带有相应错误消息的警报视图。
4. 如果获取成功，则用结果更新categories数组并重新加载列表。

构建并运行。转到“**Categories**”选项卡，您将看到列表填充了TIL应用程序中的类别：

创建用户

在TIL API中，您必须有一个用户来创建缩略词，因此首先要建立创建用户流程。打开**ResourceRequest.swift**并在GetResourcesRequest下创建一个新的枚举来表示保存结果：

```
enum SaveResult<ResourceType> {
  case success(ResourceType)
  case failure
}
```

枚举有两种情况：成功，包含API的结果；以及失败。在ResourceRequest底部添加一个新方法来保存模型：

```
// 1
func save(
  _ resourceToSave: ResourceType,
  completion: @escaping (SaveResult<ResourceType>
) -> Void) {
  do {
    // 2
    var urlRequest = URLRequest(url: resourceURL)
    // 3
    urlRequest.httpMethod = "POST"
    // 4
    urlRequest.addValue("application/json",
                        forHTTPHeaderField: "Content-Type")
    // 5
    urlRequest.httpBody =
      try JSONEncoder().encode(resourceToSave)
    // 6
    let dataTask = URLSession.shared
      .dataTask(with: urlRequest) { data, response, _ in
        // 7
        guard
          let httpResponse = response as? HTTPURLResponse,
          httpResponse.statusCode == 200,
          let jsonData = data
          else {
            completion(.failure)
            return
        }

        do {
          // 8
          let resource = try JSONDecoder()
            .decode(ResourceType.self, from: jsonData)
          completion(.success(resource))
        } catch {
          // 9
          completion(.failure)
        }
    }
    // 10
```

```
      dataTask.resume()
    // 11
    } catch {
      completion(.failure)
    }
  }
```

这是新方法的作用：

1. 声明一个方法save(_:completion:)，它接受资源保存，以及一个获取保存结果的完成处理程序。

2. 为保存请求创建一个URLRequest。

3. 将请求的HTTP方法设置为**POST**。

4. 将请求的**Content-Type**标头设置为**application/json**，以便API知道有JSON数据要解码。

5. 将请求body设置为编码的资源类型。

6. 使用请求创建数据任务。

7. 确保有HTTP响应。检查响应状态是否是**200 OK**，成功保存后API返回的代码。确保响应body中有数据。

8. 将响应body解码为资源类型。使用成功结果调用完成处理程序。

9. 捕获解码错误并使用失败结果调用完成处理程序。

10. 启动数据任务。

11. 捕获任何错误并使用失败结果调用完成处理程序。

接下来，打开**CreateUserTableViewController.swift**；并用以下代码替换save(_:)的实现：

```
  // 1
  guard
    let name = nameTextField.text,
    !name.isEmpty
    else {
      ErrorPresenter
        .showError(message: "You must specify a name", on: self)
      return
  }

  // 2
  guard
```

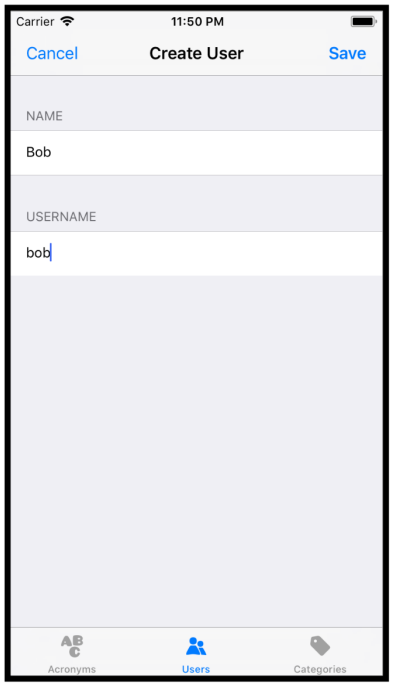
```
    let username = usernameTextField.text,
    !username.isEmpty
    else {
      ErrorPresenter.showError(
        message: "You must specify a username",
        on: self)
      return
  }

  // 3
  let user = User(name: name, username: username)
  // 4
  ResourceRequest<User>(resourcePath: "users")
    .save(user) { [weak self] result in
    switch result {
    // 5
    case .failure:
      let message = "There was a problem saving the user"
      ErrorPresenter.showError(message: message, on: self)
    // 6
    case .success:
      DispatchQueue.main.async { [weak self] in
        self?.navigationController?
          .popViewController(animated: true)
      }
    }
  }
```

这是它的作用：

1. 确保名称文本字段包含一个非空字符串。
2. 确保用户名文本字段包含一个非空字符串。
3. 从提供的数据创建一个新用户。
4. 为User创建ResourceRequest并调用save(_:completion:)。
5. 如果保存失败，则显示错误消息。
6. 如果保存成功，则返回到上一个视图：用户列表。

构建并运行。转到“**Users**”选项卡，然后点击按钮以打开“**Create User**”视图页面。填写两个字段，然后点按“**Save**”。

如果保存成功，本页面将关闭，新用户将显示在列表中：



创建缩略词

选择用户

使用API创建缩略词时，必须提供用户ID。要求用户记住并输入UUID并不是一个好的用户体验！ iOS应用应该允许用户按名称选择用户。

打开**CreateAcronymTableViewController.swift**并在viewDidLoad()下面创建一个新方法，以用默认用户填充创建缩略词视图中的User单元格：

```
func populateUsers() {
  // 1
  let usersRequest =
    ResourceRequest<User>(resourcePath: "users")

  usersRequest.getAll { [weak self] result in
    switch result {
    // 2
    case .failure:
      let message = "There was an error getting the users"
      ErrorPresenter
        .showError(message: message, on: self) { _ in
          self?.navigationController?
            .popViewController(animated: true)
```

```
        }
      // 3
      case .success(let users):
        DispatchQueue.main.async { [weak self] in
          self?.userLabel.text = users[0].name
        }
        self?.selectedUser = users[0]
      }
    }
  }
```

这是它的作用：

1. 从API获取所有用户。

2. 如果请求失败，则显示错误。当用户关闭警报视图时，从创建缩略词视图页面返回。这在showError(message:on:dismissAction:)上使用了dismissAction。

3. 如果请求成功，将user字段设置为第一个用户的名称并更新selectedUser。

在viewDidLoad()的末尾添加以下内容：

```
  populateUsers()
```

您可以点击**USER**单元格以选择其他用户来创建缩略词。此手势将打开“**Select A User**”视图页面。

打开**SelectUserTableViewController.swift**。在var users: [User] = []下面添加以下内容：

```
  var selectedUser: User!
```

此属性包含所选的用户。接下来，将以下实现添加到loadData()，以便在加载视图时列表显示用户：

```
  // 1
  let usersRequest =
    ResourceRequest<User>(resourcePath: "users")

  usersRequest.getAll { [weak self] result in
    switch result {
    // 2
    case .failure:
      let message = "There was an error getting the users"
      ErrorPresenter
        .showError(message: message, on: self) { _ in
          self?.navigationController?
            .popViewController(animated: true)
      }
      // 3
```

```
    case .success(let users):
      self?.users = users
      DispatchQueue.main.async { [weak self] in
        self?.tableView.reloadData()
      }
    }
  }
```

知道它的作用：

1. 从API获取所有用户。

2. 如果请求失败，则显示错误消息。一旦用户点击警报视图关闭，则返回上一个视图页面。

3. 如果请求成功，保存用户并重新加载列表数据。

在tableView(_:cellForRowAt:)的return cell之前添加以下内容：

```
  if user.name == selectedUser.name {
    cell.accessoryType = .checkmark
  } else {
    cell.accessoryType = .none
  }
```

这会将当前单元格与当前所选的用户进行比较。如果它们相同，请在该单元格上设置复选标记。

当用户点击一个单元格时，SelectUserTableViewController使用unwind segue导航回CreateAcronymTableViewController。

在SelectUserTableViewController中添加以下prepare(for:)的实现，为segue设置所选用户：

```
  // 1
  if segue.identifier == "UnwindSelectUserSegue" {
    // 2
    guard
      let cell = sender as? UITableViewCell,
      let indexPath = tableView.indexPath(for: cell)
      else {
        return
    }
    // 3
    selectedUser = users[indexPath.row]
  }
```

这是它的作用：

1. 验证这是预期的segue。

2. 获取触发segue的单元格的index path。

3. 将selectedUser更新为点击单元格选择的用户。

unwind segue调用CreateAcronymTableViewController中的updateSelectedUser(_:)。打开**CreateAcronymTableViewController.swift**并将以下实现添加到updateSelectedUser(_:):

```
// 1
guard let controller = segue.source
  as? SelectUserTableViewController
  else {
    return
}
// 2
selectedUser = controller.selectedUser
userLabel.text = selectedUser?.name
```

这是它的作用:

1. 确保segue源自SelectUserTableViewController。
2. 使用新值更新selectedUser并更新用户label。

最后，添加以下实现到CreateAcronymTableViewController中的prepare(for:sender:)，以在SelectUserTableViewController上设置所选用户:

```
// 1
if segue.identifier == "SelectUserSegue" {
  // 2
  guard
    let destination = segue.destination
      as? SelectUserTableViewController,
    let user = selectedUser
    else {
      return
  }
  // 3
  destination.selectedUser = user
}
```

这是它的作用:

1. 验证这是预期的segue。
2. 从segue获取目标控制器并确保已选择用户。
3. 在SelectUserTableViewController上设置所选用户。

构建并运行。在Acronyms标签中，点按“+”以显示“**Create An Acronym**”视图。点击用户行，应用程序将打开“**Select A User** ”视图，允许您选择用户。当您点按某个用户时，该用户就会在“**Create An Acronym**”页面上设置：

保存缩略词

最后，在**CreateAcronymTableViewController.swift**中替换save(_:)的实现，以保存缩略词到数据库中：

```
// 1
guard
  let shortText = acronymShortTextField.text,
  !shortText.isEmpty
  else {
    ErrorPresenter.showError(
      message: "You must specify an acronym!",
      on: self)
    return
}
guard
  let longText = acronymLongTextField.text,
  !longText.isEmpty
  else {
    ErrorPresenter.showError(
      message: "You must specify a meaning!",
      on: self)
    return
}
guard let userID = selectedUser?.id else {
  let message = "You must have a user to create an acronym!"
```

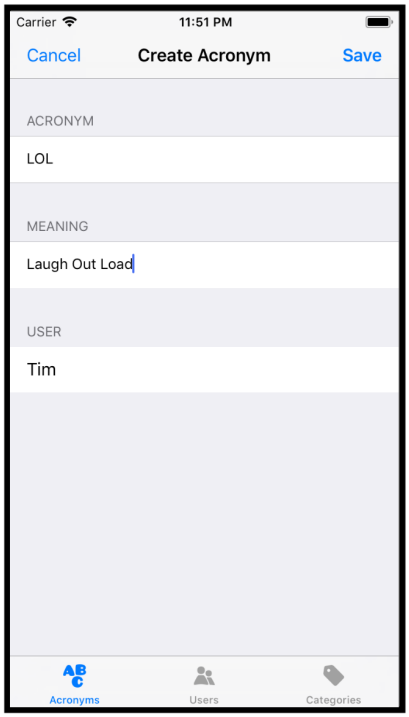
```
    ErrorPresenter.showError(message: message, on: self)
    return
  }

  // 2
  let acronym = Acronym(
    short: shortText,
    long: longText,
    userID: userID)
  // 3
  ResourceRequest<Acronym>(resourcePath: "acronyms")
    .save(acronym) { [weak self] result in
      switch result {
      // 4
      case .failure:
        let message = "There was a problem saving the acronym"
        ErrorPresenter.showError(message: message, on: self)
      // 5
      case .success:
        DispatchQueue.main.async { [weak self] in
          self?.navigationController?
            .popViewController(animated: true)
        }
    }
  }
```

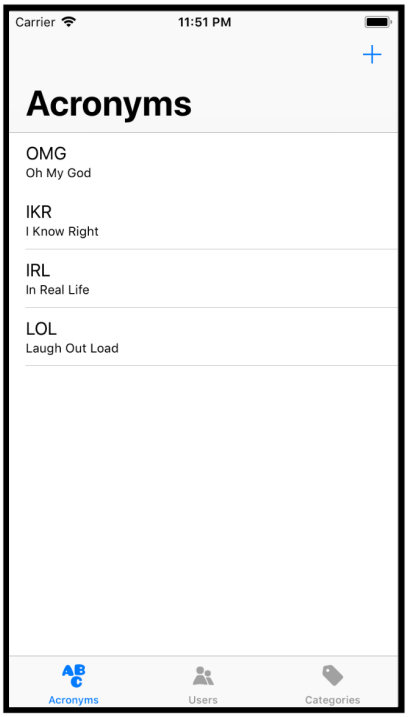
以下是保存缩略词的步骤：

1. 确保用户填写了缩略词和含义。检查所选用户是否为nil，并且用户具有有效ID。
2. 从提供的数据创建新的缩略词。
3. 为Acronym创建ResourceRequest并调用save(_:)。
4. 如果保存请求失败，则显示错误消息。
5. 如果保存请求成功，则返回到上一个视图：缩略词列表。

构建并运行。在**Acronyms**选项卡上，点按“+”。填写字段以创建缩略词，然后点 按“Save”。

已保存的缩略词将显示在列表中：





然后去哪儿?

在本章中，您学习了如何从iOS应用程序与API进行交互。您了解了如何创建不同的模型并从API中检索它们。您还学习了如何以用户友好的方式管理所需的关系。

下一章将在此基础上查看单个缩略词的详细信息。您还将学习如何实现其余的CRUD操作。 最后，您将了解如何在类别和缩略词之间建立关系。