# 我的需求

## 说明已有的项目结构

your-project/

│

├── shared/ # 共享模块

│ ├── models/ # 数据库模型

│ └── services/ # 业务逻辑

│

├── next-app/ # Next.js 应用

│ ├── pages/ # 页面组件

│ ├── public/ # 静态资源

│ └── ... # 其他 Next.js 相关文件

│

├── api/ # Vercel无服务器函数(提供API接口)

│ └── ... # API相关的函数

│

└── koa-app/ # Koa 服务端应用(非Vercel部署)

├── controllers/ # 控制器

├── routes/ # 路由

└── app.js # Koa 应用的入口文件

## 说明数据模型

SQL脚本

## 说明功能现状

我的第一个版本只实现这四个功能。我的后台数据库和Koa的部分（controllers, routers, app）已经实现完成了，Shared里面的models也完成了。

## 说明接下来的功能需求

我现在需要实施的搭建一套基于Koa的Admin UI系统，使用Tabler，我们的后台包含如下几个功能：

### 1. 登录

用户需要在系统里面存着的用户

### 2. 导航

我需要使用Tabler的布局，顶部菜单的布局方式

### 3. 登录后第一个页面是Forms 页面

一个租户可以创建一个或多个Form，默认的这个页面是当前租户的Form的列表

### 4. Account & Billing 账户和付费

按照常规方式，在Tabler的头部，右侧，登录用户的下拉菜单里面有一个这个功能

# 账户和付费工作步骤提纲

### 准备Stripe账户

注册并设置Stripe账户。

创建三个定价计划：免费、9.9美元/月、99美元/年。

### 设计数据库模型

设计用户模型以存储用户的订阅状态。

* 为了保存 Stripe 的客户 ID，在 Tenants 表中添加一个新的字段（StripeCustomerId）来存储这个信息。这样，每个租户都可以有一个与之关联的 Stripe 客户 ID，这样可以跟踪和管理系统的租户在Stripe中的订阅关系。

### 后端开发

集成Stripe SDK。

* 开发Webhook
  + 接收支付成功事件
* 检查用户订阅状态

### 前端开发

* 账户页面，使用Stripe的Price Table功能展示订阅选项（免费、月付、年付）。
* 对于已付费用户，展示他们的当前订阅计划和账单历史。

### 测试和验证

本地测试Webhook功能

转发消息，stripe listen --forward-to localhost:3000/api/stripe/webhook

触发测试事件，stripe trigger checkout.session.completed

* 测试前端页面在不同订阅状态下的用户体验。

### 部署和监控

将应用部署到生产环境。

监控应用的性能和用户的支付行为，确保系统稳定运行。

# 实现步骤

建立一个基于Koa和Tabler的Admin UI系统，你需要遵循以下步骤，结合你已有的后端逻辑和数据库结构，来开发前端界面和相应的后端接口。下面是一个大致的步骤指导：

## 1. [Done] 设置Koa项目结构

首先，确保你的Koa项目结构已经准备就绪，包括但不限于安装所需的npm包（如koa-router、koa-bodyparser等），并确保你的app.js文件已经设置好了基础的中间件和路由。

## 2. [Done] 集成Tabler UI

下载并集成Tabler: 下载Tabler UI的CSS和JS文件，将它们放置在你的Koa应用的静态资源目录下，例如public文件夹。

创建布局: 根据Tabler文档，创建一个基本的HTML布局文件，这个文件将作为你的Admin UI的模板。确保这个布局包含了Tabler的顶部导航菜单。

## 3. [Done] 实现登录功能

后端登录接口: 在api或koa-app/controllers目录下，创建一个用于处理登录逻辑的控制器。这个接口需要验证用户的用户名和密码，成功后返回一个session或token给前端。

前端登录页面: 在next-app/pages或koa-app的静态文件中，创建一个登录表单页面，使用Ajax或表单提交调用你的登录API。

## 4. [Done] 实现导航

使用Tabler布局: 利用你已创建的Tabler基础布局，添加顶部导航条。根据你的需求，这可能包括链接到Forms页面、Account & Billing页面等。

## 5. [Done] Forms页面和逻辑

Forms列表API: 在koa-app/controllers目录下创建一个API，用于获取当前租户的所有表单。这需要查询数据库中的Forms表。

前端Forms页面: 创建一个页面，用于显示表单列表。这个页面应该在用户登录后作为首页出现，调用Forms列表API获取数据，并展示给用户。

## 6. [Doing…] Account & Billing功能

实现账户信息和计费API: 在后端创建API，用于处理账户信息的获取和更新，以及展示计费信息。

创建前端页面: 在Tabler的头部导航中，添加账户和计费的下拉菜单项。对应地，创建页面用于展示和编辑账户信息，以及展示计费和付款选项。

### 准备Stripe账户

前往Stripe官网注册账户。

按照指南创建定价计划，确保有免费、9.9美元/月、99美元/年三种选项。

### 设计数据库模型

用户模型需要包含用户ID、邮箱、订阅状态等字段。

订阅模型可以包含用户ID、订阅计划ID、订阅开始时间、结束时间等。

### 后端开发

集成Stripe SDK到你的后端代码中。

编写处理订阅逻辑的代码，允许用户创建和修改他们的订阅。

实现一个接口来获取用户的订阅和账单信息。

### 前端开发

设计一个简洁的UI界面，让用户能轻松选择订阅计划。

集成Stripe的支付元素，以便用户可以输入他们的支付信息。

根据用户的订阅状态显示相应的界面和选项。

### 测试和验证

使用测试卡信息来验证支付流程。

确保所有的订阅逻辑在前端和后端都能正确工作。

### 部署和监控

部署你的应用到一个生产环境。

使用日志和监控工具来跟踪应用的表现和任何可能的问题。

## 7. 测试

测试后端API: 使用Postman或任何API测试工具，确保所有后端API都按预期工作，包括登录、获取表单列表、账户信息和计费信息等。

前端交互测试: 测试前端页面与后端API的交互是否顺畅，确保数据正确显示，且用户界面友好。

## 8. 部署和优化

部署: 将你的应用部署到服务器或云服务上。如果你使用Vercel部署Next.js应用，确保Koa应用也部署在适当的环境中。

优化和调整: 根据用户反馈和使用情况，进行必要的优化和调整。

这是一个简化的步骤指南。实际开发中，每一步都可能需要深入的开发和测试，特别是安全性（如登录认证、数据加密等）和性能优化。

# 涉及到的关键技术

## 基于租户的URL规范和导航

### 在URL对象后面增加tenantId

<http://localhost:3000/forms/1>

### 首页的控制以及路由设置

<http://localhost:3000/home/1>

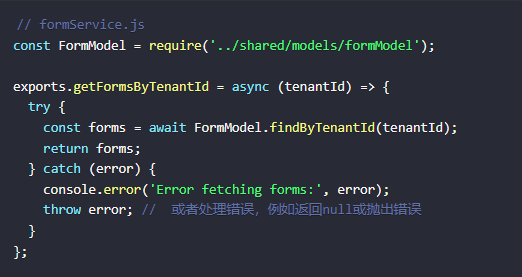
## 当API服务架构遇到业务逻辑架构后的重构

要同时支持API的标准结构（例如，使用路由、控制器来处理请求并返回标准的HTTP响应）和在程序逻辑中实现数据的调用，您可以采用以下策略：

1. 使用中间件：在Koa中，您可以使用中间件来处理数据逻辑，而不是直接在控制器中修改 ctx.body。这样，您可以在控制器中保持标准的HTTP响应处理，同时在中间件中处理数据逻辑。
2. 使用服务层：创建一个服务层，用于处理业务逻辑和数据访问。控制器可以调用服务层的方法来获取数据，然后将数据赋值给 ctx.body。这样，您可以在服务层中实现数据的调用和处理，同时保持控制器的简洁和专注于HTTP响应的处理。
3. 使用事件驱动的架构：在某些情况下，您可以使用事件驱动的架构，其中控制器触发事件，服务层监听这些事件并处理数据。这种方法可以使您的代码更加模块化和解耦。

### 服务层

首先，创建一个服务层文件，例如 formService.js：



然后，修改控制器以使用服务层：

文本

描述已自动生成

这样，您的控制器仍然负责处理HTTP请求和响应，而服务层负责处理业务逻辑和数据访问。这种方法使您的代码更加模块化，易于维护和扩展。

### 中间件

使用中间件是一种在Koa中处理数据逻辑的有效方法，它允许您在控制器之前或之后执行特定的逻辑，而不直接修改 ctx.body。这种方法可以帮助您保持控制器的简洁，同时在中间件中处理数据逻辑。

以下是一个使用中间件来处理数据逻辑的示例。在这个例子中，我们将创建一个中间件来获取租户ID，然后根据租户ID获取表单数据。

首先，创建一个中间件文件，例如 formMiddleware.js：

文本

描述已自动生成

然后，在您的路由文件中使用这个中间件：

文本

描述已自动生成

在这个例子中，getFormsByTenantId 中间件负责根据租户ID获取表单数据，并将数据存储在 ctx.state.forms 中。然后，控制器可以访问 ctx.state.forms 来获取表单数据，并将其用于渲染视图。这种方法允许您在不直接修改 ctx.body 的情况下处理数据逻辑。

使用中间件的好处是它可以帮助您保持代码的清晰和模块化，同时也使得数据处理逻辑更加容易管理和重用。

### 事件驱动

事件驱动的架构（Event-Driven Architecture, EDA）是一种设计模式，其中组件之间通过发布和订阅事件来进行通信。在Node.js和Koa应用中，可以使用事件驱动的架构来处理异步操作，如数据库查询、文件操作等。这种方法可以使您的代码更加模块化，易于维护和扩展。

以下是一个使用事件驱动的架构来处理数据逻辑的示例。在这个例子中，我们将创建一个事件发射器，用于触发事件，以及一个事件监听器，用于处理这些事件。

首先，创建一个事件发射器文件，例如 eventEmitter.js：

文本

描述已自动生成

然后，创建一个事件监听器文件，例如 formListener.js：

文本

描述已自动生成

在您的控制器中，您可以触发事件来获取表单数据，并监听相应的事件来处理结果：

文本

描述已自动生成

在这个例子中，控制器触发了一个事件来获取表单数据，并监听了两个事件：一个用于处理成功获取到的表单数据，另一个用于处理获取表单数据时出现的错误。这种方法使得控制器可以异步地处理数据逻辑，而不需要直接修改 ctx.body。

使用事件驱动的架构的好处是它可以帮助您处理异步操作，同时也使得您的代码更加模块化和解耦。这种方法特别适合于处理复杂的业务逻辑，或者需要在多个组件之间共享状态的情况。

### 对比说明

在决定哪种重构方式最适合您的SaaS应用时，需要考虑应用的具体需求、架构的复杂性、团队的技术栈和经验，以及未来的可扩展性和维护性。以下是对三种重构方式的对比说明和实践建议：

1. 使用中间件

对比说明：

优点：提供了一种清晰的方式来处理请求的生命周期，允许在控制器之前或之后执行特定的逻辑。这有助于保持控制器的简洁和专注于HTTP响应的处理。

缺点：如果应用的业务逻辑非常复杂，使用中间件可能会导致逻辑分散在多个地方，难以跟踪和维护。

实践建议：对于简单的业务逻辑，使用中间件是一个很好的选择。它可以帮助您保持控制器的简洁，同时在中间件中处理数据逻辑。

如果您的应用需要处理复杂的业务逻辑，可能需要考虑使用服务层或事件驱动的架构。

2. 使用服务层

对比说明：

优点：提供了一种模块化的方式来处理业务逻辑和数据访问，使得代码更加易于维护和扩展。服务层可以被多个控制器重用，减少了代码重复。

缺点：引入了额外的抽象层，可能会增加学习和理解代码的难度。

实践建议：

对于中等复杂度的应用，使用服务层是一个很好的选择。它可以帮助您保持代码的清晰和模块化，同时提供了一种方便的方式来重用业务逻辑。

如果您的应用需要处理复杂的业务逻辑，或者您希望更好地分离业务逻辑和控制器，使用服务层是一个很好的选择。

3. 使用事件驱动的架构

对比说明：

优点：提供了一种强大的方式来处理异步操作和复杂的业务逻辑。事件驱动的架构可以帮助您更好地管理状态和副作用，使得代码更加模块化和解耦。

缺点：对于初学者来说，理解和实现事件驱动的架构可能会比较困难。此外，如果不正确地使用，可能会导致代码难以跟踪和调试。

实践建议：

对于需要处理复杂异步操作或需要高度模块化和解耦的应用，使用事件驱动的架构是一个很好的选择。它可以帮助您更好地管理状态和副作用，同时提供了一种灵活的方式来处理业务逻辑。

如果您的应用需要处理复杂的业务逻辑，或者您希望更好地分离业务逻辑和控制器，使用事件驱动的架构是一个很好的选择。

总结

对于您的SaaS应用，最适合的重构方式取决于您的具体需求和应用的复杂性。如果您的应用主要是处理简单的HTTP请求和响应，使用中间件可能是最简单和直接的方法。如果您的应用需要处理复杂的业务逻辑或需要更好地分离业务逻辑和控制器，使用服务层或事件驱动的架构可能更合适。

在选择重构方式时，建议您考虑您的应用的具体需求，以及您的团队对这些方法的熟悉程度。同时，不要忽视代码的可读性和维护性，选择一种可以帮助您和您的团队更容易理解和维护代码的方法。