



Serenity

Developer Guide

# Table of Contents

前言	1.1
准备开始	1.2
从 Visual Studio 库中获取安装 SERENE	1.2.1
直接在 Visual Studio 中安装 SEREN	1.2.2
开始使用 Serene	1.2.3
Serene 功能概览	1.3
主题	1.3.1
本地化	1.3.2
用户和角色管理	1.3.3
列表页	1.3.4
编辑对话框	1.3.5
教程	1.4
Movie 网站 (类似于IMDB)	1.4.1
创建电影 (Movie) 表	1.4.1.1
为影片 (Movie) 表生成代码	1.4.1.2
自定义影片界面	1.4.1.3
处理 Movie 的导航	1.4.1.4
自定义快速检索	1.4.1.5
添加一个影片类型字段	1.4.1.6
添加影片流派 (Movie Genres)	1.4.1.7
更新 Serenity 程序包	1.4.1.8
允许选择多个流派	1.4.1.9
筛选具有多个流派的列表	1.4.1.10
演员和角色	1.4.1.11
在人员对话框列出参演影片	1.4.1.12
添加海报 (Primary) 和简介 (Gallery) 图片	1.4.1.13
多租户系统	1.4.2

添加租户（Tenants）表和 TenantId 字段	1.4.2.1
为租户（Tenants）生成代码	1.4.2.2
在用户对话框中选择租户	1.4.2.3
使用 TenantId 筛选用户	1.4.2.4
从用户窗体移除租户下拉列表	1.4.2.5
在服务端对租户选择进行安全检测	1.4.2.6
为新用户设置 TenantId	1.4.2.7
防止编辑其他租户的用户	1.4.2.8
隐藏租户管理权限	1.4.2.9
多租户角色	1.4.2.10
使用 Serenity 服务行为	1.4.2.11
扩展多租户行为到 Northwind	1.4.2.12
处理检索脚本（Lookup Scripts）	1.4.2.13
会议管理	1.4.3
创建检索表	1.4.3.1
疑难解惑指南	1.5
如何删除 Serene 的 Northwind 及其他示例？	1.5.1
如何更新 Serenity 的 NuGet 程序包？	1.5.2
如何升级到 Serenity 2.0 并启用 TypeScript？	1.5.3
如何使用其他类型的数据库？	1.5.4
如何使用 Active Directory 或 LDAP 进行身份验证？	1.5.5
如何设置连接的数据库方言（Database Dialect）？	1.5.6
如何删除网格中的新增（add）按钮？	1.5.7
如何使用 SlickGrid 格式化器（Formatter）？	1.5.8
如何将内联动作按钮（Inline Action Buttons）添加到网格？	1.5.9
如何添加一个行选择列？	1.5.10
如何设置级联编辑器（Cascaded Editors）？	1.5.11
如何使用验证码？	1.5.12
如何在 Serene 中注册权限？	1.5.13
如何在 Serenity 中使用第三方插件？	1.5.14

如何启用脚本合并 (Script Bundling) ?	1.5.15
常见问题	1.6
故障排除	1.7
服务定位器(Service Locator) & 初始化	1.8
静态依赖类	1.8.1
IDependencyResolver 接口	1.8.2
IDependencyRegistrar 接口	1.8.3
MunqContainer 类	1.8.4
CommonInitialization 静态类	1.8.5
认证 & 授权	1.9
IAuthenticationService 接口	1.9.1
IAuthorizationService 接口	1.9.2
IPermissionService 接口	1.9.3
IUserDefinition 接口	1.9.4
IUserRetrieveService 接口	1.9.5
静态 Authorization 类	1.9.6
配置系统	1.10
定义配置设置	1.10.1
IConfigurationRepository 接口	1.10.2
AppSettingsJsonConfigRepository	1.10.3
静态 Config 类	1.10.4
本地化	1.11
LocalText 类	1.11.1
语言标识符	1.11.2
语言回退 (Language Fallbacks)	1.11.3
ILocalTextRegistry 接口	1.11.4
LocalTextRegistry 类	1.11.5
待审状态	1.11.5.1
注册翻译	1.11.6
手工注册翻译	1.11.6.1

嵌入本地化文本	1.11.6.2
枚举文本	1.11.6.3
JSON 本地化文本	1.11.6.4
缓存	1.12
本地缓存	1.12.1
ILocalCache 接口	1.12.1.1
静态 LocalCache 类	1.12.1.2
用户简介 (User Profile) 缓存示例	1.12.1.3
分布式缓存	1.12.2
网站群和缓存	1.12.2.1
IDistributedCache 接口	1.12.2.2
静态 DistributedCache 类	1.12.2.3
DistributedCacheEmulator 类	1.12.2.4
CouchbaseDistributedCache 类	1.12.2.5
RedisDistributedCache 类	1.12.2.6
二级缓存	1.12.3
同步本地和分布式缓存	1.12.3.1
TwoLevelCache 类	1.12.3.2
实体(Row)	1.13
Mapping 特性	1.13.1
FieldFlags 枚举	1.13.2
流式 SQL (Fluent SQL)	1.14
SqlQuery 对象	1.14.1
Criteria 对象	1.14.2
连接和事务	1.15
服务	1.16
服务终结点	1.16.1
列表请求处理器 (ListRequestHandler)	1.16.2
部件(Widgets)	1.17
ScriptContext 类	1.17.1

---

Widget 类	1.17.2
Widget < TOptions > 泛型类	1.17.3
TemplatedWidget 类	1.17.4
TemplatedDialog 类	1.17.5
网格列表	1.18
格式化器类型	1.18.1
持久化设置	1.18.2
代码生成器 (Sergen)	1.19
使用的工具和库	1.20

---

## 前言

### 什么是Serenity平台

Serenity 是建立在开源技术上的 ASP.NET MVC/Javascript 应用程序平台。

它旨在使开发变得更简单，同时避免重复代码，减少花在重复任务的时间并提供最佳的软件设计实践，从而降低维护成本。

### 谁适合使用该平台

Serenity 最适合应用于有大量数据输入的表单业务应用程序或者面向公众的后台管理网站，它的功能同样也适用于其他类型的Web应用程序。

### 在哪里可找到相关信息

在阅读本指南和其教程以后，跟随以下资源可获取有关 Serenity 的详细信息。

#### ***Github Repository:***

<https://github.com/volkanceylan/Serenity>

#### ***Issues / Questions***

<https://github.com/volkanceylan/Serenity/issues>

#### ***Change Log:***

<https://github.com/volkanceylan/Serenity/blob/master/CHANGELOG.md>

#### ***Serene Application Template:***

<https://visualstudiogallery.msdn.microsoft.com/559ec6fc-feef-4077-b6d5-5a99408a6681>

#### ***Tutorial / Sample Source Code:***

<https://github.com/volkanceylan/Serenity-Tutorials>

### 名字有什么含义

Serenity 在字典中解释为 平和，舒适，沉着。

这正是我们使用 Serenity 努力实现的目标，我们希望你在安装并使用 Serenity 之后，你会觉得这太.....

## 它提供了哪些功能

- 模块化、基于服务的 web 应用程序模型
- 为 SQL 表生成初始服务/用户界面的代码生成器
- 服务器上的基于 T4 模板生成智能感知/编译时验证的引用脚本部件
- 基于 T4 模板生成从脚本端调用 AJAX 服务的同时提供安全的编译时类型和智能感知
- 基于属性 (**Attribute**) 的表单定义系统 (在服务器端用一个简单的 C# 类编写用户界面)
- 通过表单定义实现全自动无缝数据绑定 (表单 <-> 实体 <-> 服务)
- 缓存助手 (本地/分布式)
- 自动缓存验证
- 配置系统 (独立的存储介质。可以数据库、文件等形式保存设置)
- 简单的日志记录
- 报表 (报表只提供数据，不依赖于渲染形式，类似于 MVC)
- 脚本引用，压缩 (使用 Node / UglifyJS / CleanCSS) 和内容版本化 (不再需要额外按 F5/清除浏览器缓存)
- 流式 SQL 生成器 (SELECT/INSERT/UPDATE/DELETE)
- 微型 ORM (集成 Dapper)
- 可定制的类 REST 服务，用于实体类的重用信息并做自动验证
- 基于属性 (**Attribute**) 的导航菜单
- 用户界面本地化 (本地化文本可存储在 json 文件、嵌入的资源文件、数据库、内存等)
- 数据本地化 (使用扩展表机制帮助本地化，甚至支持用户输入，如查找表)
- 脚本插件系统 (灵感来自 jQueryUI，但更适合 C# 代码)
- 客户端和服务端验证 (基于 jQuery 验证插件，但可分离依赖)
- 操作日志 (在 CDC 中不可用)
- 基于数据系统集成测试
- 动态脚本
- 脚本端模板

## 背景

这部分原本是写在 CodeProject 上的一编介绍 Serenity 的文章，由于这篇文章是一篇宣传代码的广告而没有包含代码就被拒绝了。他们没有错，因为我在本指南中做了一个视频教程链接，而不是复制粘贴代码。

如果你不喜欢了解历史，可以毫无影响地跳到下一章 :)

我们开发人员每天都在解决一系列相同的问题，就像大学生做习题一样工作。

即使我们知道他们已经有合适的解决方案，但仍避免不了面对相同的工作。实际上，这样可以帮助提高我们的技术，你总不能在不犯错误中学习吧？但我们应该学会如何界定培训和浪费时间。

当你开始一个新的项目时，你需要作出几个决定：使用哪个平台？如何组织架构及使用哪些类库？今天，你可以在每一次的话题中有如此多的选择，是的，具有一些选项是好事，只要这些选项是在有限的范围内，因为我们的时间不是无限的。

下面是关于 **Serenity** 的简短历史：**Serenity** 旨在处理你业务应用程序中的常见任务，并让你腾出宝贵的时间专注于特定应用程序域的功能。

我的第一份正式的关于 Web 技术的工作是在设计国家特定 Web 站点的一些大牌行业，如汽车公司（顺便感慨一下，我们在谈论的是十多年以前的往事了，时光流逝的真快）。

在与他们签约之前，我有一段桌面应用程序的软件架构师职业生涯，我应邀去为他们设计一个 ASP.NET WebForms 平台。他们解释说，他们有很多共享模块，如新闻、列表、导航。但由于需求不同，他们不得不先复制/粘贴，然后根据每个客户的具体需求自定义代码。当他们想要添加一个共同的功能时，他们就得重复修改每个网站。

当时，在市场中没有那么多的 CMS 系统，我为他们设计了一个系统，甚至都不知道它被称为 CMS。对我来说，它并不完美，甚至还有瑕疵，因为我只花了几周的时间去设计它，但是他们却非常满意这个结果，因为它使开发一个新网站的时间由数月提高到数日/周，同时生成的代码也比以前任何时候都更易于管理。

从错误和经验中学习，那个简陋的 CMS 逐渐完善成为更好的系统平台，后来，该平台不断地演变并被应用在不同领域，例如帮助台系统、CRM、ERP、人事管理系统、电子文档管理系统、大学学生信息系统等。

为了与不同类型的的应用程序、系统及老旧的数据库兼容，它必须具有足够的灵活性，并经历了多次的架构调整。

现在我们看到的 **Serenity**，即管它是一个开源了 2 年左右的项目，它还有更久远的历史，但它依然年轻，精力充沛，并且不害怕改变。它与时俱进并保持稳定，这可能意味着随着时间的推移会有不继的重大更改，但我们努力将更改保持在最低限度且不偏执地向后保持兼容。

## 准备开始

让你快速上手的最好、最快的方法是使用示例应用程序模板 **SERENE**。

你有两种方式安装 **SERENE** 模板到你的 Visual Studio 中：

- 从 Visual Studio 库中获取安装 SERENE
- 直接在 Visual Studio 中安装 SEREN

即使你不使用该模板，也能够使用 Serenity 进行工作。

你可以在 NuGet 中把 Serenity 库添加到任何 .NET 项目。

也可以把 Serenity 作为 GIT 子模块添加到项目中，并让其总保持最新的源代码。

我们将在其他章节进行更详细的介绍

# 从 Visual Studio 库中获取安装 SERENE

在浏览器打开链接 <http://visualstudiogallery.msdn.microsoft.com/>

在搜索 Visual Studio 库 搜索框中输入 **Serene** 并按 ENTER 键。

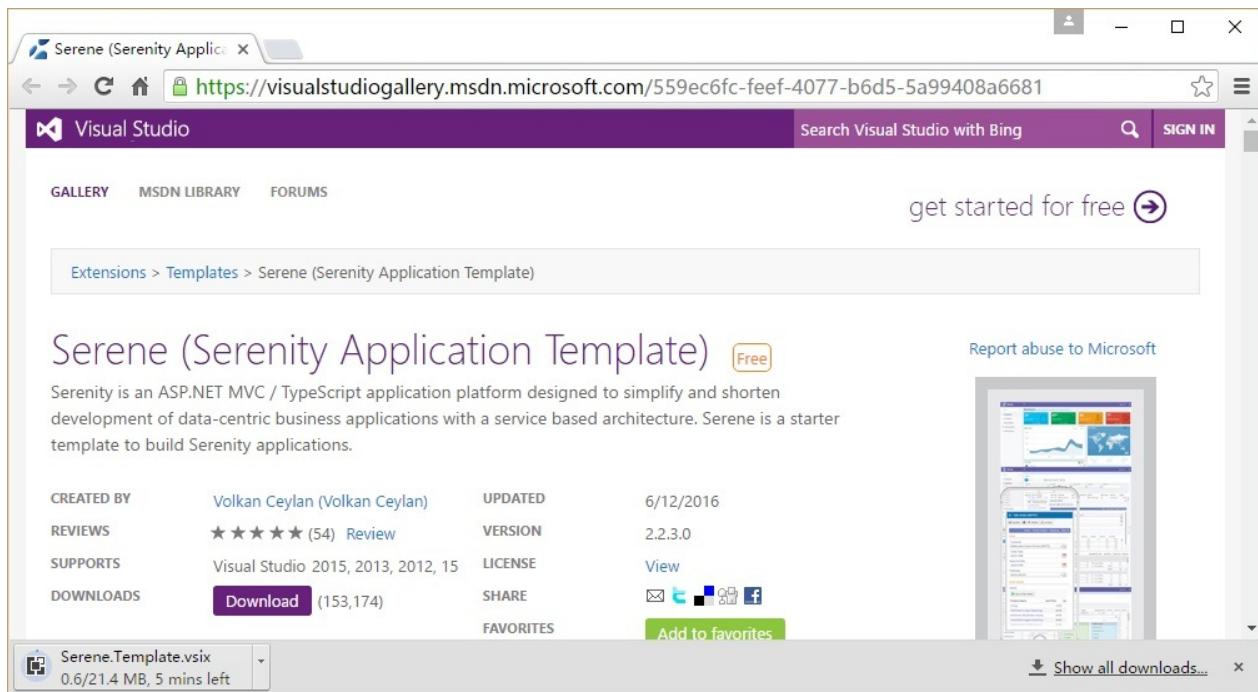
The screenshot shows the Visual Studio Gallery homepage. At the top right, there is a search bar with the placeholder "Search Visual Studio with Bing". Below it, a sign-in button is visible. On the left, there's a sidebar with options like "Products and Extensions", "Visual Studio", "GALLERY", "MSDN LIBRARY", and "FORUMS". The main content area has a heading "Products and Extensions for Visual Studio". It features three main buttons: "Browse" (6,563 items), "Upload" (Add Products and Extensions), and "My Extensions" (Sign in to access your Gallery). To the right, there's a "get started for free" button with a link. A search bar on the right contains the word "serene". Below these, there are sections for "Popular Searches" (Visual Studio 2010, Microsoft, power commands, T4, power tools, theme, Coding) and a grid of recently added extensions. One extension, "HBase ADO.NET Provider" by CDATA SOFTWARE, is highlighted with a "Trial" badge.

你将看到 **Serene (Serenity Application Template)**，单击标题进入详细页面。

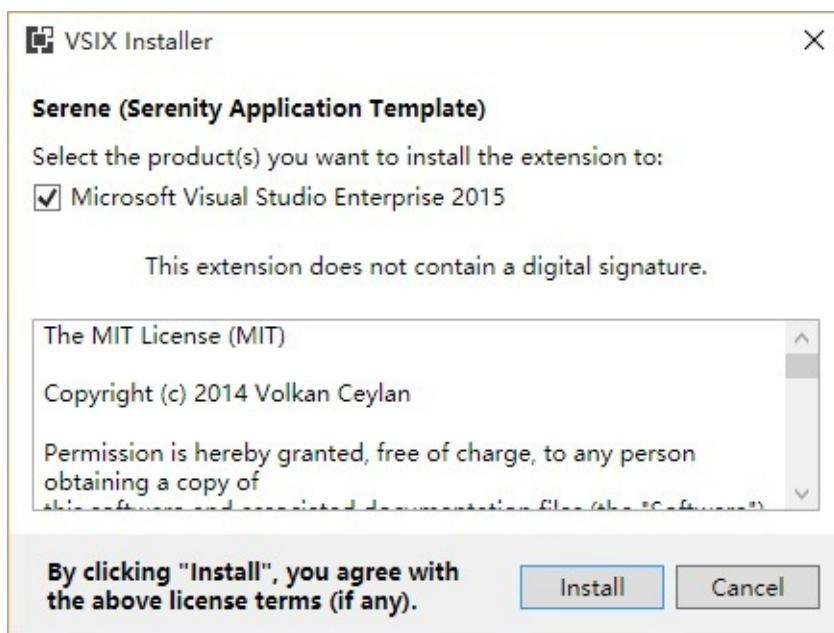
The screenshot shows the search results for "serene" on the Visual Studio Gallery. The URL in the address bar is [https://visualstudiogallery.msdn.microsoft.com/site/search?query=serene&f\[0\].value=serene&f\[0\].type=s](https://visualstudiogallery.msdn.microsoft.com/site/search?query=serene&f[0].value=serene&f[0].type=s). The search bar at the top contains "serene". The results show one item: "Serene (Serenity Application Template)" by Volkan Ceylan. The item details include: "1 result in serene [Clear]", "Sort by: Relevance", "Relevance", "Free", "Updated 6/12/2016", "Released 11/4/2014", "153,148 Downloads", and "Template". The extension icon is a blue swirl logo.

点击下载，VSIX 文件将下载到你的电脑。

## 从 Visual Studio 库中获取安装 SERENE



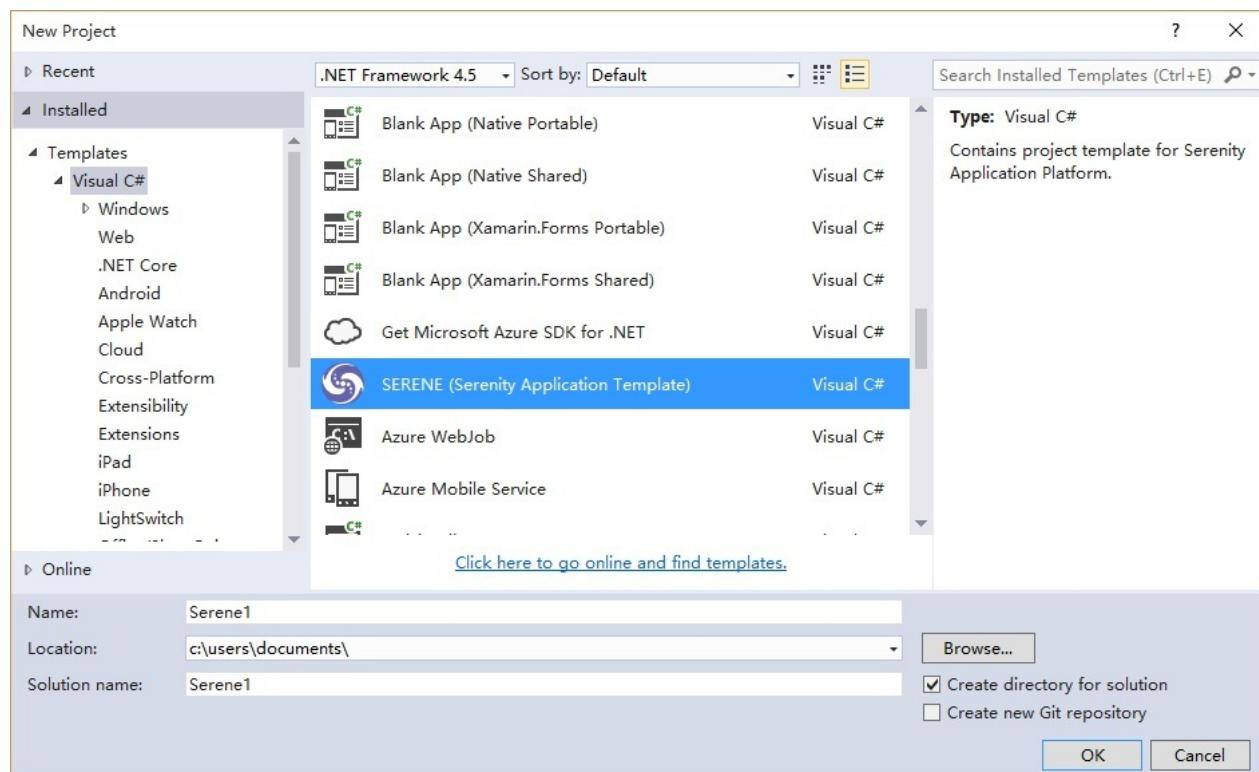
下载完成后，单击下载的 VSIX 文件启动 Visual Studio 扩展安装对话框，并单击安装。



请注意，这个应用程序模板需要 Visual Studio 2012 或更高的版本。请确保你已经安装了最新的 Visual Studio 更新。

启动 Visual Studio（如果已经打开，请重新启动）。单击文件 => 新建项目。你应该能在 已安装模板 => Visual C# 中看到 Serenity 模板。

## 从 Visual Studio 库中获取安装 SERENE



在名称中输入新建应用程序名称，如 *MyCompany*、*MyProduct*、*HelloWorld* 或使用默认名称 *Serene1*。

请不要将其命名为 *Serenity*，它可能会与其他 Serenity 程序集冲突。

点击 OK 并稍等片刻，Visual Studio 将创建模板解决方案。

# 直接在 Visual Studio 中安装 SEREN

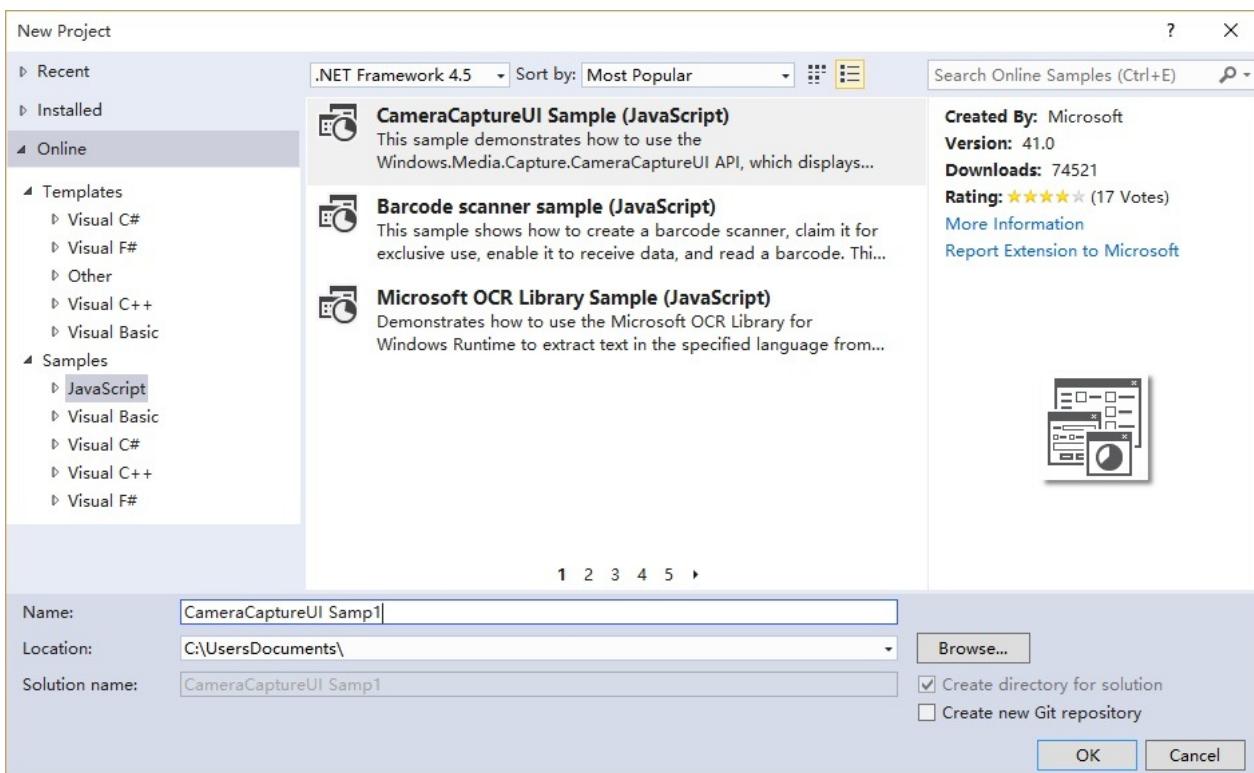
启动 Visual Studio 并点击 新建 => 项目。

请注意，这个应用程序模板需要 Visual Studio 2012 或更高的版本。请确保你已经安装了最新的 Visual Studio 更新。

在新建项目对话框左侧显示着 最近的模板、已安装的模板 和 联机模板，其中 已安装的模板 是我们最常用的部分。

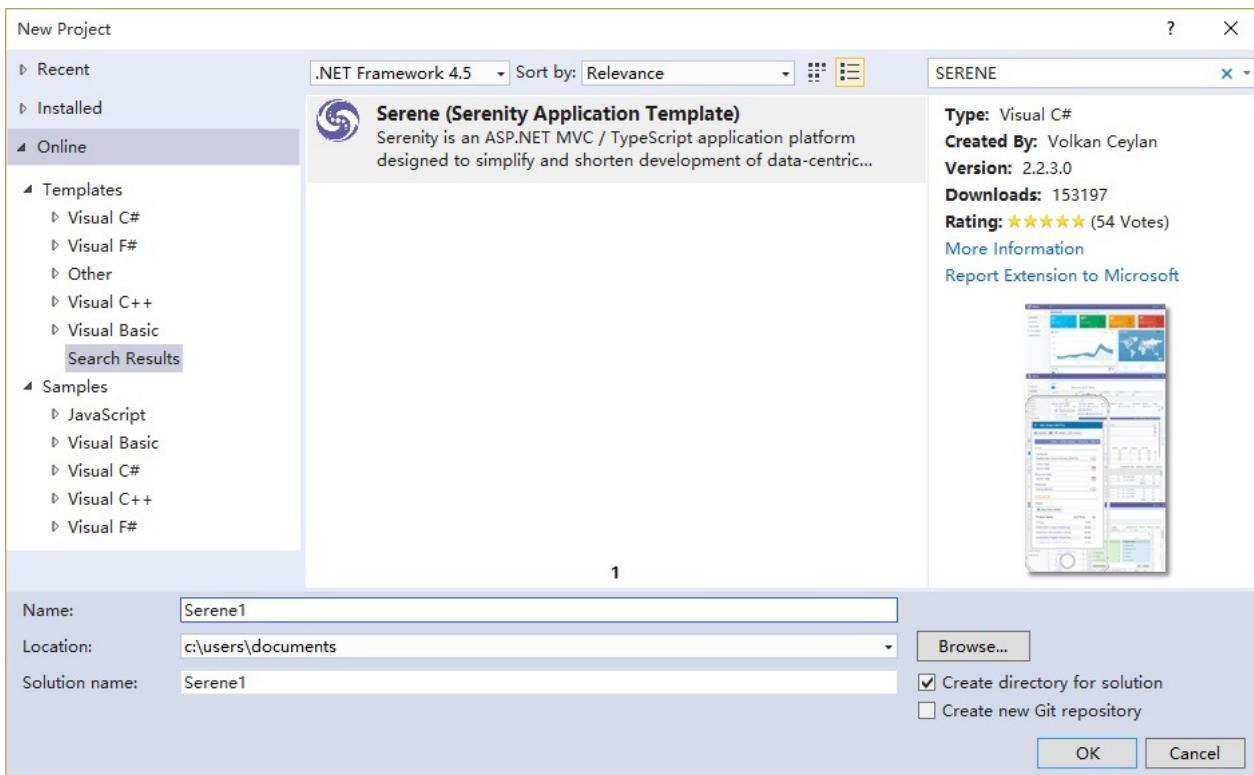
点击 联机模板

在加载结果时请耐心等候。



在搜索框中输入 SERENE 并按 ENTER 键确认。

你将看到 *Serene (Serenity Application Template)* 模板：



在名称中输入新建应用程序名称，如 *MyCompany*、*MyProduct*、*HelloWorld* 或使用默认名称 **Serene1**。

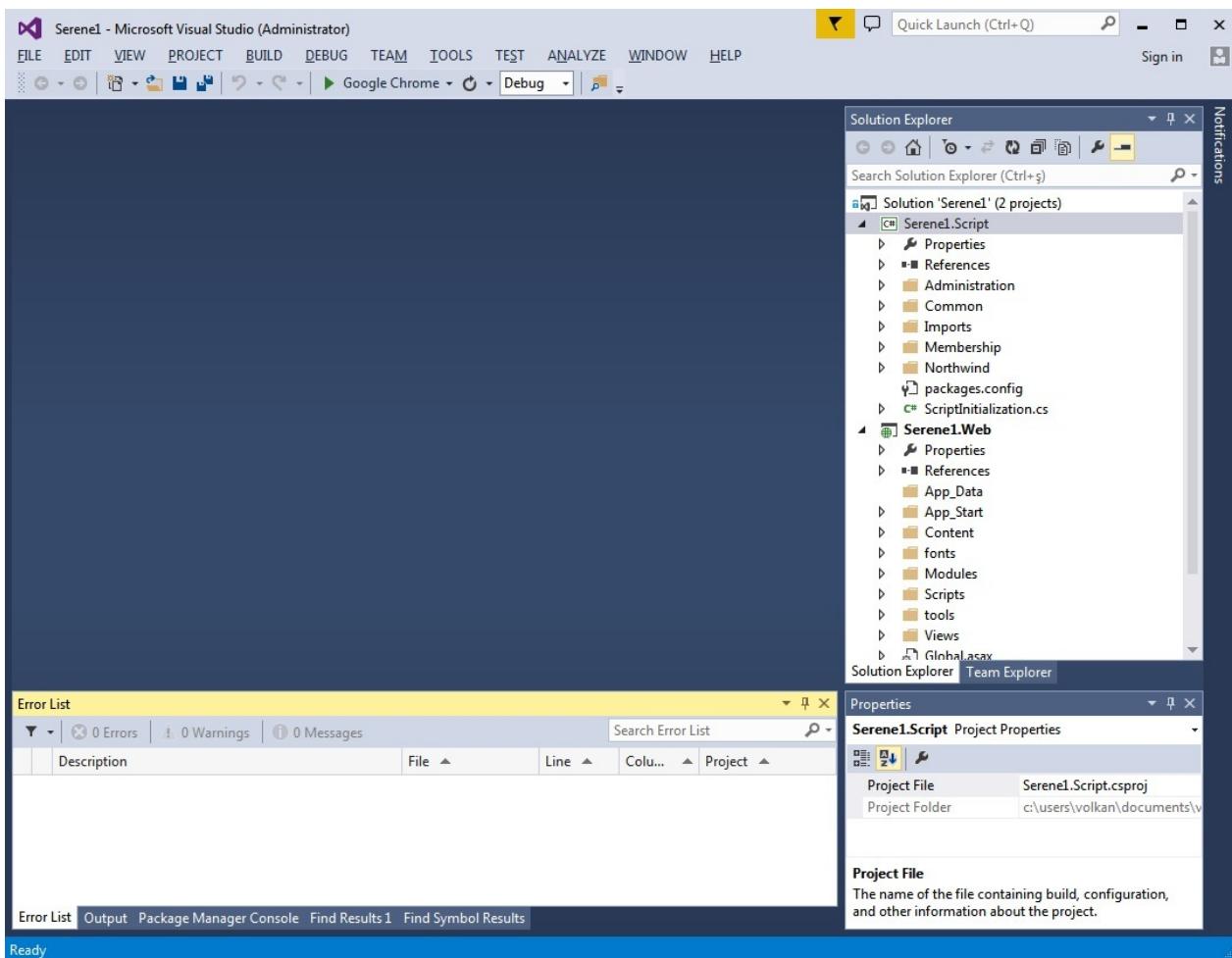
请不要将其命名为 *Serenity*，它可能会与其他 Serenity 程序集冲突。

点击OK并稍等片刻，Visual Studio 将创建模板解决方案。

使用该方式创建第一个项目后，Serene 的模板已安装到 Visual Studio，所以你可以使用 新建项目 对话框中的 已安装的模板 中选择 Serenity 模板创建另一个新的 Serenity 应用程序。

# 开始使用 Serene

在 Visual Studio 中使用 Serene 模板创建第一个项目后，你将看到这样的解决方案：



你的解决方案包含一个名为 Serene1.Web 的 ASP.NET MVC 应用程序。

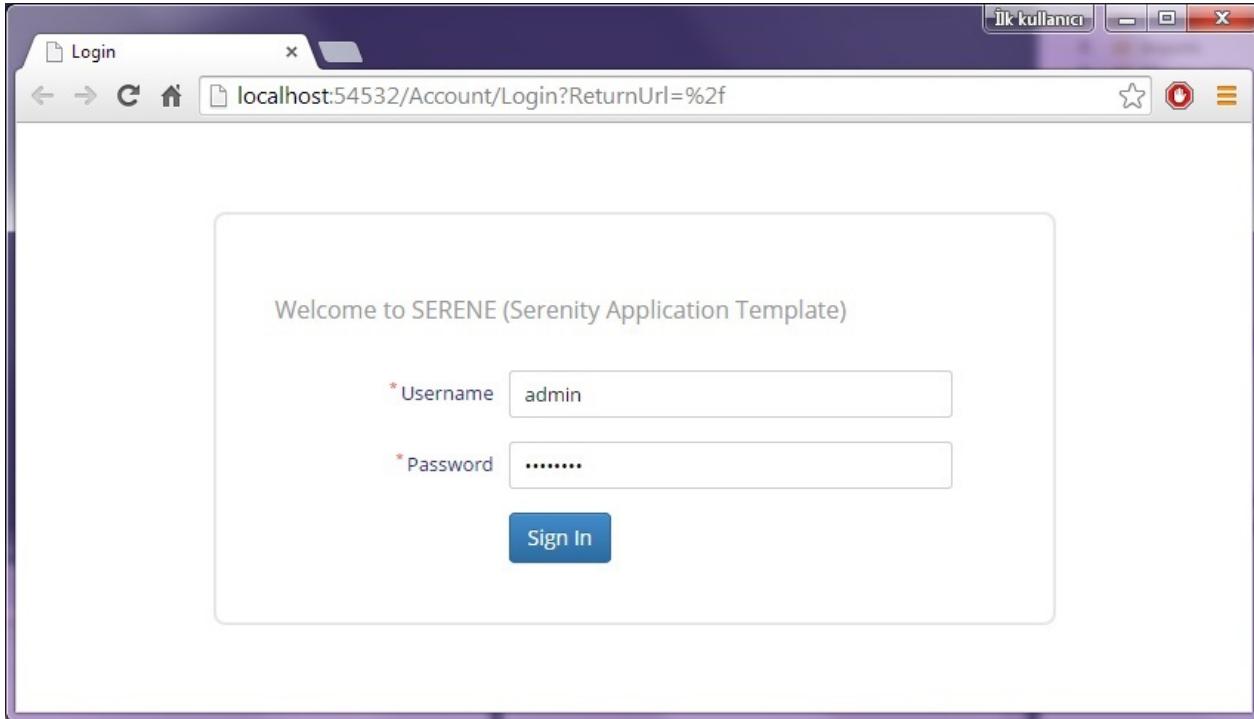
Serene1.Web 项目包含 C# (.cs) 编写的服务端代码和 TypeScript (.ts) 编写的客户端代码。

Serene1.Web 拥有 Serenity 的 NuGet 程序包引用，这样你就可以根据需要使用包管理控制台随时更新它。

上面是 < 2.1 版的截图，其中有一个使用 Saltarelle 编译器生成客户端代码的 Serene1.Script 项目。从 2.1 之后我们更换为 TypeScript 编写客户端代码，并且客户端代码 (.ts 文件) 也全部集成到 Web 项目。

Serene 在首次运行时将自动在 SQL local db 中创建数据库，所以只须按 F5 就可运行使用示例。

当应用程序启动时使用 `admin` 用户名和 `serenity` 密码登录。然后你可以使用管理/用户管理 页面更改密码或创建更多的用户。



示例应用程序包含由 Serenity 代码生成器为 Northwind 数据库生成的服务和可编辑的用户界面。

## 数据库连接问题

如果你在第一次启动 Serene 时遇到类似于下面的数据库连接错误：

A network-related or instance-specific error occurred while establishing a connection to SQL Server. The server was not found or was not accessible. Verify that the instance name is correct and that SQL Server is configured to allow remote connections. (provider: SQL Network Interfaces, error: 50 - Local Database Runtime error occurred. The specified LocalDB instance does not exist. )

此错误意味着你可能没有安装 SQL Server Local DB 2012。SQL Server Local DB 2012 预装在 Visual Studio 2012 / Visual Studio 2013。

在 Serene.Web/web.config 文件中有 `Default` 和 `Northwind` 连接配置：

```
<connectionStrings>
    <add name="Default" connectionString="Data Source=(LocalDb)\v11.0;
        Initial Catalog=Serene_Default_v1; Integrated Security=True"
            providerName="System.Data.SqlClient" />
</connectionStrings>
```

(localdb)\v11.0 默认使用 SQL Server 2012 LocalDB 实例。

如果你没有 SQL LocalDB 2012，可以从下面的连接中获取安装：

<http://www.microsoft.com/en-us/download/details.aspx?id=29062>

Visual Studio 2015 内置 SQL Server 2014 LocalDB，它的默认实例名为 MsSqlLocalDB。因此，如果你使用的是 VS2015，请把连接字符串 (localdb)\v11.0 修改为 (localdb)\MsSqlLocalDB。

```
<connectionStrings>
    <add name="Default" connectionString="Data Source=(LocalDb)\MsSqlLocalDB;
        Initial Catalog=Serene_Default_v1; Integrated Security=True"
            providerName="System.Data.SqlClient" />
</connectionStrings>
```

如果你仍然有错误，请使用管理员身份打开 cmd 命令提示符，并输入：

```
> sqllocaldb info
```

这会列出类似下面的 localdb 实例：

```
MSSqlLocalDB
test
```

如果你的结果中没有列出 MsSqlLocalDB，可以创建它：

```
> sqllocaldb create MsSqlLocalDB
```

如果你有另一个 SQL server 实例，例如 SQL Express，请把数据源更改  
为 .\SqlExpress :

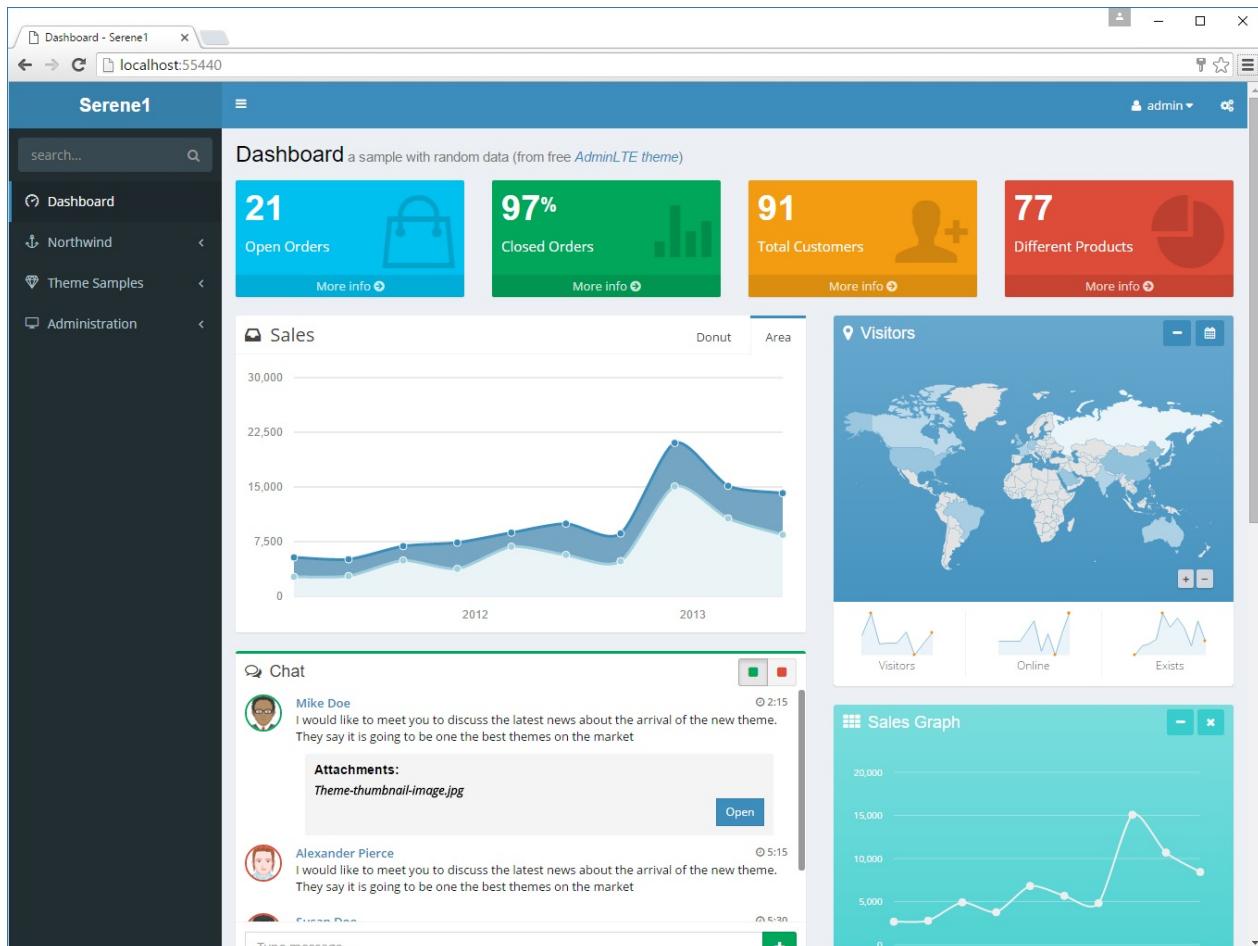
```
<connectionStrings>
    <add name="Default" connectionString="Data Source=.\SqlExpre
ss;
    Initial Catalog=Serene_Default_v1; Integrated Security=T
rue"
        providerName="System.Data.SqlClient" />
</connectionStrings>
```

你还可以使用另一个 SQL 服务器，只需更改连接字符串即可。

请分别为 Default 和 Northwind 数据库执行这些步骤，因为 Serene 1.6.4.3+ 默认  
创建这两个数据库。

# Serene 功能概览

登录 Serene 系统后，你将看到 控制面板 页面。



这是一个使用免费主题

AdminLTE(<https://almsaeedstudio.com/themes/AdminLTE/index.html>)的示例  
页面。

网页的数据内容，除了从 Northwind 数据库表计算而来，还包含一些随机数据。

网页的左侧是一个带有搜索功能的手风琴菜单，我们将在后面的章节中讨论如何自定义菜单内容。

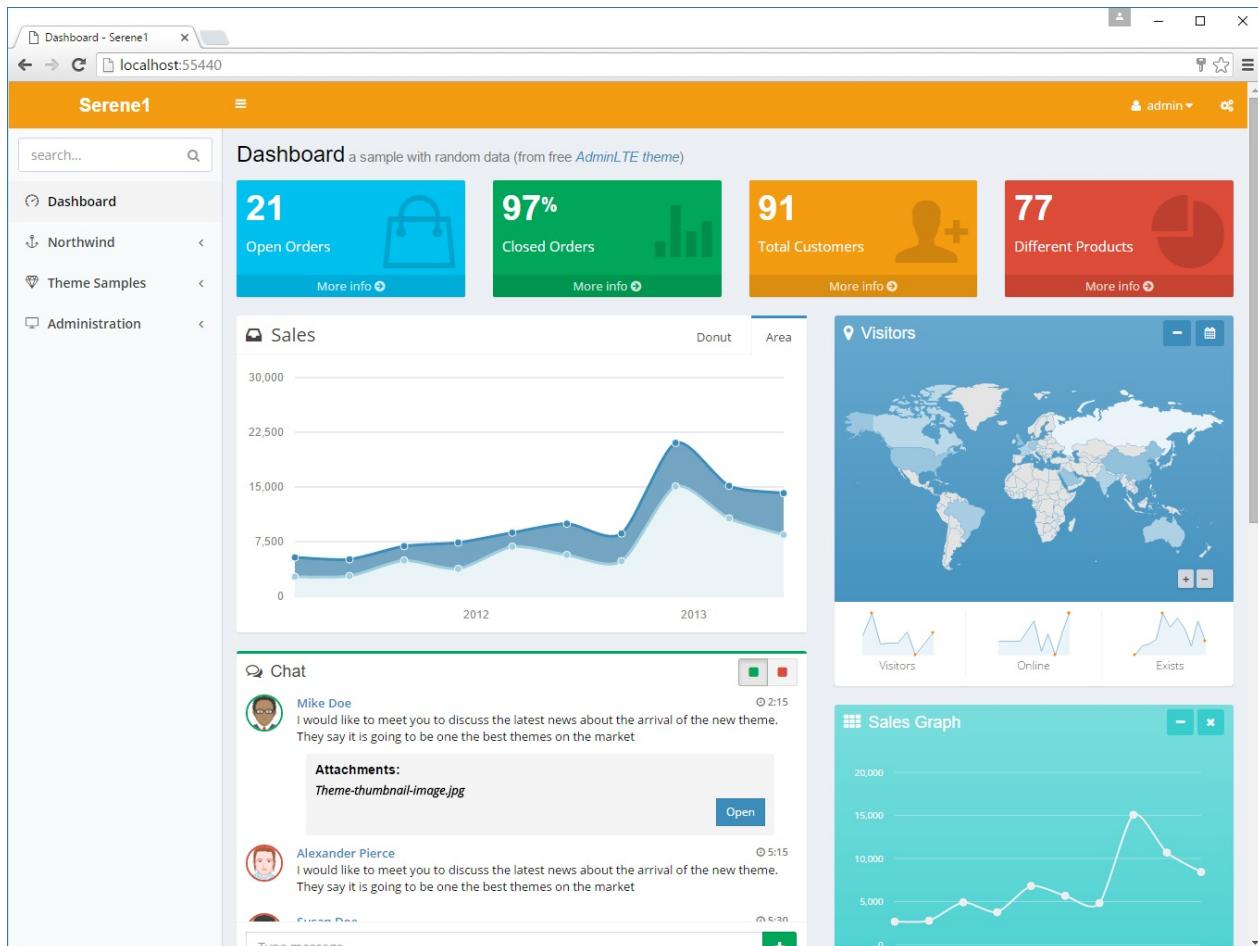
在顶部的导航中，左边显示网站的名称（或 logo），右边有一个包含当前用户名的下拉列表和一个设置按钮，我们可以通过设置按钮更改主题或显示的语言。

- 主题
- 本地化



# 主题

Serene 默认是蓝黑色主题。可在屏幕右上角用户名旁边的设置按钮 中修改网站主题。



此功能是通过替换页面中 body 的 CSS 样式实现的。

如果你查看源代码，你可能会发现像下面 " 标签中的皮肤样式：

```
<body id="s-DashboardPage" class="fixed sidebar-mini hold-transition
skin-blue has-layout-event">
```

当你选择浅黄色的主题时，它实际上是将 body 标签的 "skin-blue" 样式更改为 "skin-yellow-light"：

```
<body id="s-DashboardPage" class="fixed sidebar-mini hold-transition
skin-yellow-light has-layout-event">
```

这些更改是在内存中完成，所以不需要重新加载页面。

另外，上面设置的“浅黄色”将作为“主题偏好”的内容保存在浏览器的 cookie 中，所以下次你启动 Serene 时，它会记住你的偏好，将显示浅黄色主题。

这些主题皮肤文件位于 Serene.Web 项目的 “Content/adminlte/skins/” 目录下。如果打开该目录文件，你将看到下面的文件：

```
_all-skins.less
skin.black-light.less
site.blue.less
site.yellow-light.less
site.yellow.less
```

我们使用 LESS 生成 CSS，所以如果要修改样式文件，你应该修改 LESS 文件而不是 CSS 文件，当编译项目的时候，LESS 文件会被编译生成 CSS 文件（使用 Node 以开发模式编译 Less）。

此操作是在 Serene.Web.csproj 文件中的生成步骤中配置：

```
...
<Target Name="CompileSiteLess" AfterTargets="AfterBuild">
  <Exec Command="$(ProjectDir)tools\node\lessc.cmd
    $(ProjectDir)Content\site\site.less
    $(ProjectDir)Content\site\site.css">
  </Exec>
</Target>
...
```

这里 `site.less` 与对应编译后的 `css` 文件存放在同一目录中。

关于 LESS 编译方式及其语法的更多信息，请查看 <http://lesscss.org/>。



# 本地化

Serene 允许你通过右上角的设置菜单 来修改网站当前使用的语言。

尝试把网站语言修改为西班牙语。

ID	Nombre de Empresa	Nombre de Contacto	Contacto Título	Ciudad	Región	Código Postal	País
ALFKI	Alfreds Futterkiste	Maria Anders	Sales Representative	Berlin		12209	Germany
ANATR	Ana Trujillo Emparedados y helados	Ana Trujillo	Owner	México D.F.		05021	Mexico
ANTON	Antonio Moreno Taquería	Antonio Moreno	Owner	México D.F.		05023	Mexico
AROUT	Around the Horn	Thomas Hardy	Sales Representative	London		WA1 1DP	UK
BERGS	Berglunds snabbköp	Christina Berglund	Order Administrator	Luleå		958 22	Sweden
BLAUS	Blauer See Delikatessen	Hanna Moos	Sales Representative	Mannheim		68306	Germany
BLONP	Blondesddsl père et fils	Frédérique Citeaux	Marketing Manager	Strasbourg		67000	France
BOLID	Bólido Comidas preparadas	Martín Sommer	Owner	Madrid		28023	Spain
BONAP	Bon app!	Laurence Lebihan	Owner	Marseille		13008	France
BOTTM	Bottom-Dollar Markets	Elizabeth Lincoln	Accounting Manager	Tsawassen	BC	T2F 8M4	Canada
BSBEV	B's Beverages	Victoria Ashworth	Sales Representative	London		EC2 5NT	UK
CACTU	Cactus Comidas para llevar	Patricia Simpson	Sales Agent	Buenos Aires		1010	Argentina
CENTC	Centro comercial Moctezuma	Francisco Chang	Marketing Manager	México D.F.		05022	Mexico
CHOPS	Chop-suey Chinese	Yang Wang	Owner	Bern		3012	Switzerland
COMM1	Comércio Mineiro	Pedro Afonso	Sales Associate	Sao Paulo	SP	05432-043	Brazil
CONSH	Consolidated Holdings	Elizabeth Brown	Sales Representative	London		WX1 6LT	UK

我并不熟悉西班牙语，这是使用机器翻译的，若有错误，请见谅。

当你更改语言时，页面将重新加载，与修改主题后没有刷新页面而直接应用修改的方式不同。

Serene 同样把 es 作为将作为“语言设置”的内容保存在浏览器的 cookie 中，所以下次你访问该站点时，它会记住你最新的设置，将显示为西班牙语。

当你第一次启动 Serene 时，你可能会看到网站内容显示的是英文，但也有可能显示为西班牙、土耳其语或俄语（这些都是目前可用的示例语言），这与你的操作系统或浏览器的语言设置有关。

可在 web.config 中配置：

```
<globalization culture="en-US" uiCulture="auto:en-US" />
```

在这里我们设置 UI 的本地化语言为自动选择，如果系统无法确定你浏览器的语言，将使用 EN-US。

你也可以设置其他语言作为备用语言：

```
<globalization culture="en-US" uiCulture="auto:tr-TR" />
```

或者设置一个默认语言，在这种情况下，网站将以设置的默认语言显示网站内容：

```
<globalization culture="en-US" uiCulture="es" />
```

如果不想要用户更改用户界面语言，你应该删除语言选择下拉列表。

你可以通过使用 系统管理 / 语言管理 菜单下的页面为语言下拉列表添加更多的语言选项。

## 本地化 UI 文本

Serene 有自定义资源文本译文的能力。

在导航菜单中选择 系统管理 / 翻译管理：

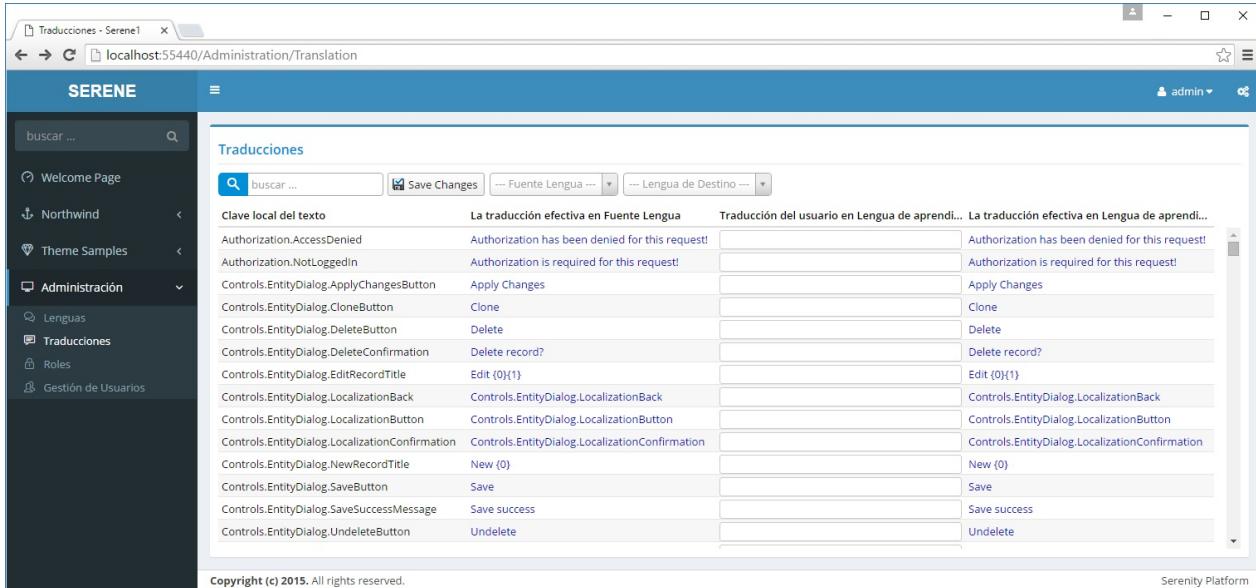
Local Text Key	Effective Translation in Source Language	User Translation in Target Language	Effective Translation in Target Language
Navigation.Administration	Administration		Administración
Navigation.Administration/Languages	Languages		Lenguas
Navigation.Administration/Roles	Roles		Roles
Navigation.Administration/Translations	Translations		Traducciones
Navigation.Administration/User Management	User Management		Gestión de Usuarios
Navigation.Dashboard	Dashboard		Salpicadero
Navigation.LogoutLink	Logout		Cerrar Sesión
Navigation.Northwind	Northwind		Northwind
Navigation.Northwind/Categories	Categories		Categorías
Navigation.Northwind/Customers	Customers		Clientes
Navigation.Northwind/Employees	Employees		Empleados
Navigation.Northwind/Orders	Orders		Órdenes
Navigation.Northwind/Products	Products		Productos
Navigation.Northwind/Regions	Regions		Regiones

在顶部左侧的搜索框中输入导航文本并查看文本与导航菜单的关系。

在页面的下拉列表中选择英语作为原文及西班牙语作为译文，并在本地文字标识 *Navigation.Dashboard* 的用户自定义中输入 *Welcome Page*。

点击 保存修改。

当你切换到西班牙语言时，控制面板菜单项将更改为 *Welcome Page* 而不是 *Salpicadero*。



Clave local del texto	La traducción efectiva en Fuente Lengua	Traducción del usuario en Lengua de aprendizaje
Authorization.AccessDenied	Authorization has been denied for this request!	Authorization has been denied for this request!
Authorization.NotLoggedIn	Authorization is required for this request!	Authorization is required for this request!
Controls.EntityDialog.ApplyChangesButton	Apply Changes	Aplicar cambios
Controls.EntityDialog.CloneButton	Clone	Clonar
Controls.EntityDialog.DeleteButton	Delete	Borrar
Controls.EntityDialog.DeleteConfirmation	Delete record?	¿Borrar el registro?
Controls.EntityDialog.EditRecordTitle	Edit {0}{1}	Editar {0}{1}
Controls.EntityDialog.LocalizationBack	Controls.EntityDialog.LocalizationBack	Controls.EntityDialog.LocalizationBack
Controls.EntityDialog.LocalizationButton	Controls.EntityDialog.LocalizationButton	Controls.EntityDialog.LocalizationButton
Controls.EntityDialog.LocalizationConfirmation	Controls.EntityDialog.LocalizationConfirmation	Controls.EntityDialog.LocalizationConfirmation
Controls.EntityDialog.NewRecordTitle	New {0}	Nuevo {0}
Controls.EntityDialog.SaveButton	Save	Guardar
Controls.EntityDialog.SaveSuccessMessage	Save success	Guardado exitosamente
Controls.EntityDialog.UndeleteButton	Undelete	Restaurar

当你保存更改时，Serene 在 *App\_Data/texts* 文件夹中创建 *user.texts.es.json* 文件，其内容如下：

```
{  
    "Navigation.Dashboard": "Welcome Page"  
}
```

在 *~/scripts/site/texts* 文件夹中，也有类似的 Serene 默认翻译的页面文本 JSON 文件：

- *site.texts.es.json*
- *site.texts.invariant.json*
- *site.texts.tr.json*

建议你在发布网站之前把 *user.texts.xx.json* 的内容拷贝到 *site.texts.xx.json* 文件中。如果 *App\_Data* 已被项目排除，你还可以使用这种方式对其进行版本控制。



# 用户和角色管理

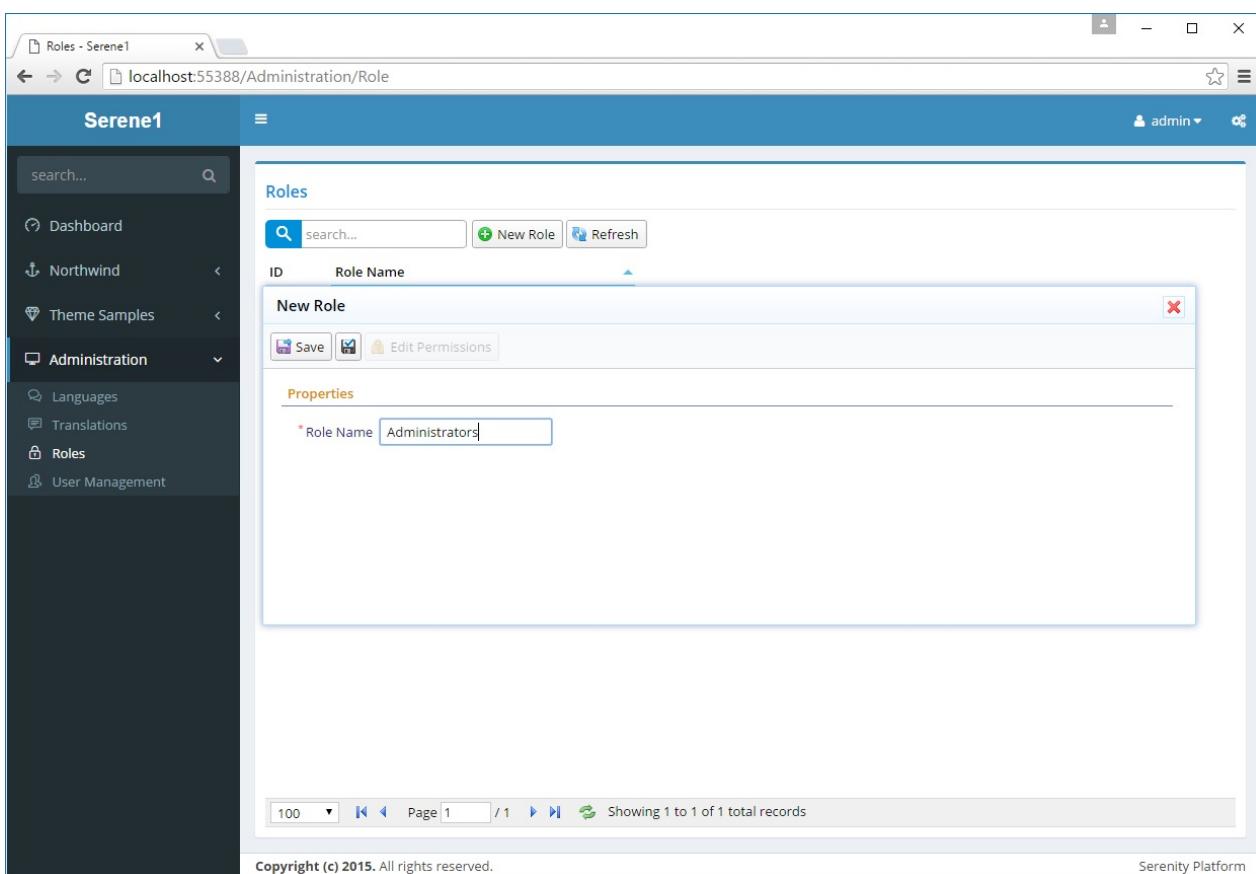
Serene 内置用户、角色和权限管理。

此功能并不是 Serenity 的内置功能，这里仅提供一个示例，所以你可以随时选择实现或使用你自己的用户管理方式，我们将在下面的章节介绍如何实现。

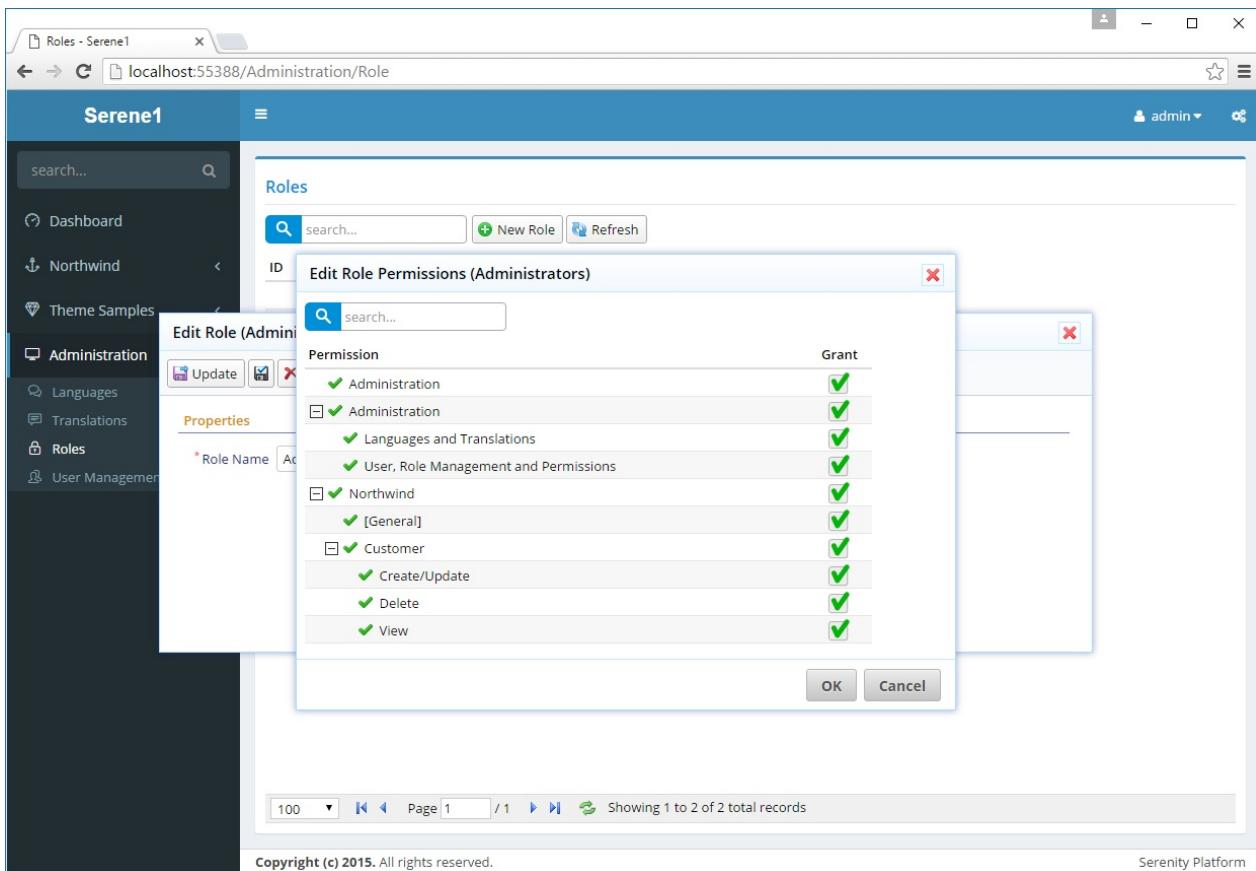
打开 系统管理 / 角色管理，并创建两个角色：*Administrators* 和 *Translators*。

点击 新增角色 并输入 *Administrators*，然后点击保存。

重复相同的步骤创建 *Translators* 角色。



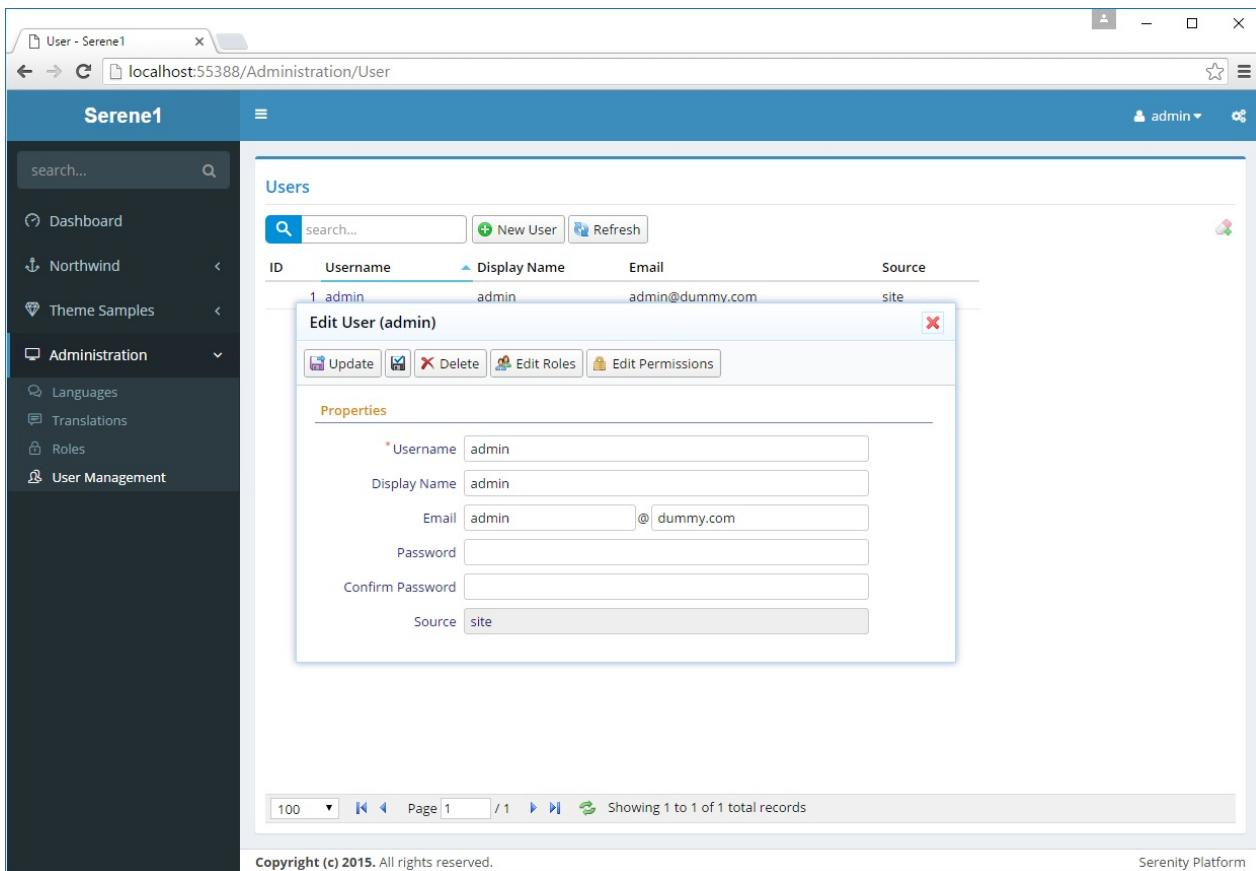
然后点击角色 *Administrators* 打开编辑对话框，再点击 修改权限 按钮修改角色权限，勾选所有复选框授予该角色所有操作的访问权限，然后点击 *OK* 按钮确认。



用相同的方式给 *Translations* 角色授权，但只授予 系统管理：语言和翻译管理 权限。

导航到 系统管理 / 用户管理 页面添加更多的用户。

点击 *admin* 用户编辑其详细信息。



在这里你可以修改 admin 的用户名、显示名称、电子邮件等详细信息。

你也可以在 密码 和 确认密码 输入框中输入密码，然后点击 更新 来修改密码（*serenity* 的默认密码）。

你也可以删除它，但这会使你无法登录而不能使用该系统。

**admin** 是 Serene 的特殊用户，因为它具有所有的操作权限，即使没有显式授予他操作权限。

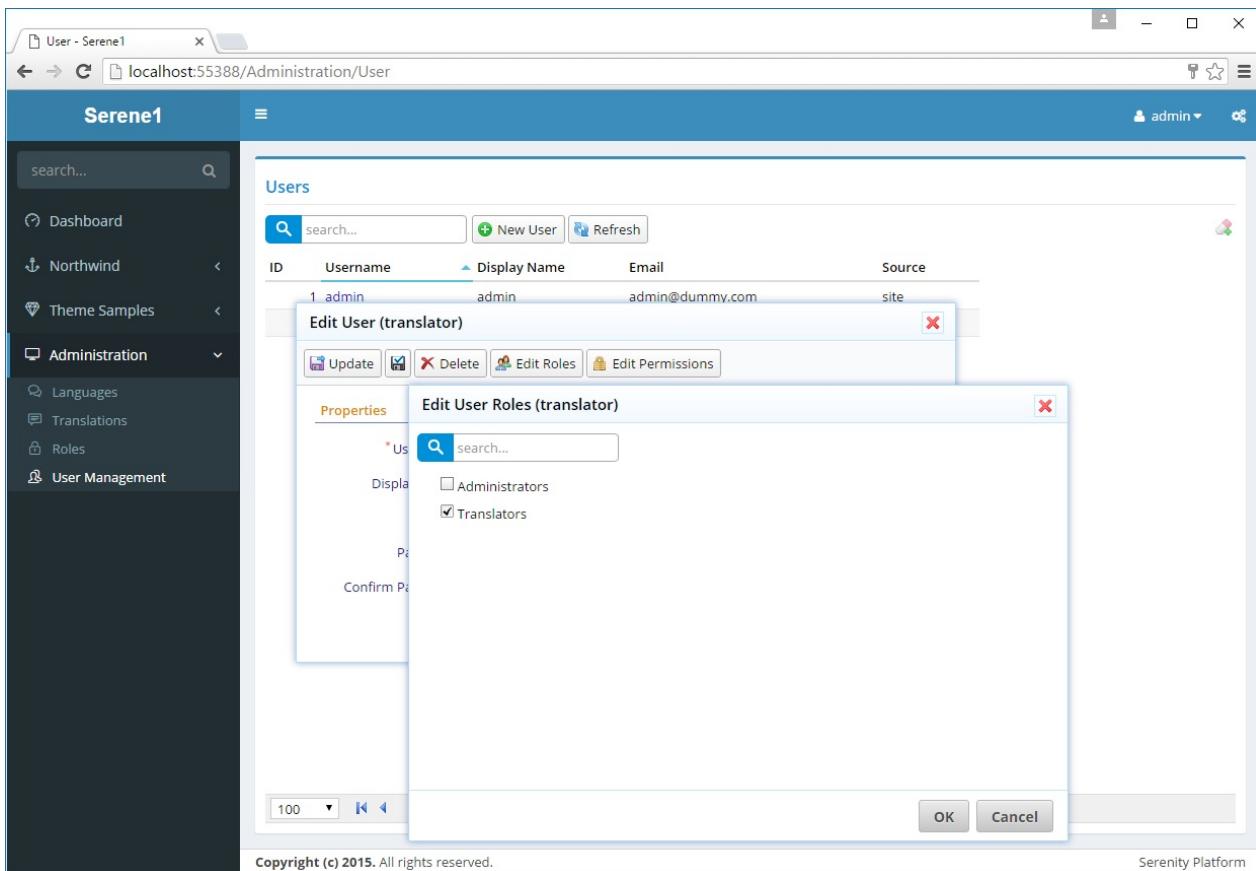
让我们创建另外一个用户并添加角色和授予操作权限。

关闭当前对话框，点击 新增用户 并输入 *Translator* 作为用户名，并填写完其他信息，然后点击 保存 按钮确认。

The screenshot shows the Serene1 application's User Management interface. On the left, there is a sidebar with navigation links: Dashboard, Northwind, Theme Samples, Administration (Languages, Translations, Roles, User Management), and a search bar. The main area displays a table of users with columns: ID, Username, Display Name, Email, and Source. A modal dialog is open over the table, titled 'New User'. It contains fields for Username ('translator'), Display Name ('Translator'), Email ('translator @ dummy.com'), Password ('\*\*\*\*\*'), Confirm Password ('\*\*\*\*\*'), and Source ('site'). Below the modal is a footer with pagination controls (Page 1 / 1) and a note: 'Showing 1 to 2 of 2 total records'. The bottom right corner of the page says 'Serenity Platform'.

你可能已经注意到 保存 按钮旁边有一个没有说明描述的黑色磁盘图标表示的应用更改按钮，不同于 保存 按钮，当你使用它时，对话框保持打开以便让你看到修改记录的保存结果，这样你就可以在关闭对话框之前修改角色和权限。

现在点击 **Translator** 的 修改角色 按钮打开编辑角色对话框，授予其 翻译管理 角色并点击 **OK** 按钮确认。



当你授予用户角色时，他自动获取授予角色的所有权限。通过单击 修改权限 按钮，你也可以显式授予额外的权限。同样地，你也可以显式撤销用户角色权限。

现在关闭弹出的所有对话框，并点击网站右上角用户名所在的下拉列表中的 退出 按钮注销登录。

使用 *Translator* 用户的账号密码登录系统。

*Translator* 用户将只能访问 控制面板、*Theme Samples*、翻译管理 页面。

## 用户和角色管理

The screenshot shows the Serene1 application interface for managing translations. The left sidebar has a dark theme with the following navigation items:

- Dashboard
- Theme Samples
- Administration
  - Languages
  - Translations

The main content area is titled "Translations". It features a search bar, a "Save Changes" button, and dropdown menus for "Source Language" and "Target Language". A table lists various local text keys and their corresponding effective translation in the source language and user translation in the target language. The columns are:

Local Text Key	Effective Translation in Source Language	User Translation in Target Language	Effective
Authorization.AccessDenied	Authorization has been denied for this request!		Autho
Authorization.NotLoggedIn	Authorization is required for this request!		Autho
Controls.EntityDialog.ApplyChangesButton	Apply Changes		Appl
Controls.EntityDialog.CloneButton	Clone		Clone
Controls.EntityDialog.DeleteButton	Delete		Delete
Controls.EntityDialog.DeleteConfirmation	Delete record?		Delete
Controls.EntityDialog.EditRecordTitle	Edit {0}{1}		Edit {C
Controls.EntityDialog.LocalizationBack	Controls.EntityDialog.LocalizationBack		Contro
Controls.EntityDialog.LocalizationButton	Controls.EntityDialog.LocalizationButton		Contro
Controls.EntityDialog.LocalizationConfirmation	Controls.EntityDialog.LocalizationConfirmation		Contro
Controls.EntityDialog.NewRecordTitle	New {0}		New {I
Controls.EntityDialog.SaveButton	Save		Save
Controls.EntityDialog.SaveSuccessMessage	Save success		Save s
Controls.EntityDialog.UndeleteButton	Undelete		Undel
Controls.EntityDialog.UndeleteConfirmation	Undelete record?		Undel
Controls.EntityDialog.UpdateButton	Update		Updat
Controls.EntityGrid.IncludeDeletedToggle	display inactive records		displa
Controls.EntityGrid.NewButton	New {0}		New {I
Controls.EntityGrid.RefreshButton	Refresh		Refres

At the bottom of the page, there is a copyright notice: "Copyright (c) 2015. All rights reserved." and the text "Serenity Platform".

# 列表页

Serene 有 Northwind 数据库的列表和编辑页面，让我们来看看 Northwind 菜单下的 Products 页面。

ID	Product Name	Dis...	Supplier	Category	Quantity Per Unit	Unit Price	Units In Stock	Units On Order	Reorder L...
17	Alice Mutton	<input checked="" type="checkbox"/>	Pavlova, Ltd.	Meat/Poultry	20 - 1 kg tins	39	0	0	0
3	Aniseed Syrup	<input type="checkbox"/>	Exotic Liquids	Condiments	12 - 550 ml bottles	10	13	70	25
40	Boston Crab Meat	<input type="checkbox"/>	New England Seafood Cannery	Seafood	24 - 4 oz tins	18.4	123	0	30
60	Camembert Pierrot	<input type="checkbox"/>	Gai pâturage	Dairy Products	15 - 300 g rounds	34	19	0	0
18	Carnarvon Tigers	<input type="checkbox"/>	Pavlova, Ltd.	Seafood	16 kg pkg.	62.5	42	0	0
1	Chai	<input type="checkbox"/>	Exotic Liquids	Beverages	10 boxes x 20 bags	18	39	0	10
2	Chang	<input type="checkbox"/>	Exotic Liquids	Beverages	24 - 12 oz bottles	19	17	40	25
39	Chartreuse verte	<input type="checkbox"/>	Aux joyeux ecclésiastiques	Beverages	750 cc per bottle	18	69	0	5
4	Chef Anton's Cajun Seasoning	<input type="checkbox"/>	New Orleans Cajun Delights	Condiments	48 - 6 oz jars	22	53	0	0
5	Chef Anton's Gumbo Mix	<input checked="" type="checkbox"/>	New Orleans Cajun Delights	Condiments	36 boxes	21.35	0	0	0
48	Chocolate	<input type="checkbox"/>	Zaanse Snoepfabriek	Confections	10 pkgs.	12.75	15	70	25
38	Côte de Blaye	<input type="checkbox"/>	Aux joyeux ecclésiastiques	Beverages	12 - 75 cl bottles	263.5	17	0	15
58	Escargots de Bourgogne	<input type="checkbox"/>	Escargots Nouveaux	Seafood	24 pieces	13.25	62	0	20
52	Filo Mix	<input type="checkbox"/>	G'day, Mate	Grains/Cereals	16 - 2 kg boxes	7	38	0	25
71	Flotemysost	<input type="checkbox"/>	Norske Meierier	Dairy Products	10 - 500 g pkgs.	21.5	26	0	0

在这里我们看到按产品名称（初始排序顺序）排序的产品列表。

Grid 组件使用的是 SlickGrid，并自定义其主题。关于 SlickGrid 详见：

<https://github.com/mleibman/SlickGrid>

你可以通过单击列标题更改顺序。若要按降序排序，请再次单击相同的列标题。

要按多个列进行排序，可以使用 Shift + Click。

这是先按 Category 列然后用 Supplier 列进行排序的结果：

当你更改排序顺序时，列表由一个 **AJAX** 请求服务加载数据。

当你第一次打开页面，初始记录也是通过调用 **AJAX** 加载数据。

默认情况下列表加载记录的大小是每页 100 条，且仅从服务器加载当前页面的数据。可在 **列表左下角** 设置每页显示数据的大小。

在列表的左上角，你可以输入一些关键字进行搜索。

例如，输入 **coffee** 查找名称中含 **coffee** 的产品。

它使用产品名称字段搜索，也可以使用另一个或多个字段进行快速搜索，我们将在后面的章节进行详细介绍。

在列表右上角，有 *Supplier* 和 *Category* 筛选条件。

下拉列表组件使用的是 Select2，详见：

<https://github.com/select2/select2>

选择 *Seafood* 作为 *Category* 的过滤条件，结果将只显示该类别的产品。

The screenshot shows a web-based application interface for managing products. On the left is a sidebar with navigation links for Dashboard, Northwind (Customers, Orders, Products, Suppliers, Shippers, Categories, Regions, Territories), Theme Samples, and Administration. The main area is titled 'Products' and contains a table with columns: ID, Product Name, Dis..., Supplier, Category, Quantity Per Unit, Unit Price, Units In Stock, Units On Order, and Reorder Level. A search bar at the top has 'Seafood' selected in the Category dropdown. The table shows four rows of product data, all belonging to the 'Seafood' category. At the bottom of the grid, there is a pagination control showing 'Showing 1 to 4 of 4 total records'.

所有的排序、分页和过滤都是在服务器端进行，由 Serenity 生成动态 SQL 进行查询。

也可以在网格右下角单击 编辑筛选 添加列筛选条件。

This screenshot shows the same application interface as above, but with an 'Edit Filter' dialog box overlaid on the products grid. The dialog allows users to build complex filtering logic using AND and OR operators. The current filter conditions are:

- ( Category equal Seafood )
- and ( Units in Stock greater than or equal 1 )
- or ( Reorder Level greater than or equal 0 )
- and ( Supplier equal Bigfoot Breweries )

At the bottom of the dialog are 'OK' and 'Cancel' buttons, and a 'close' button in the top right corner. The background grid shows 20 products, with the first few being Côte de Blaye, Chai, Chang, and Lakkal.

在这里，你可以通过单击 增加筛选条件 添加列表的任意列并设置列名、过滤操作、条件值让其作为过滤条件。

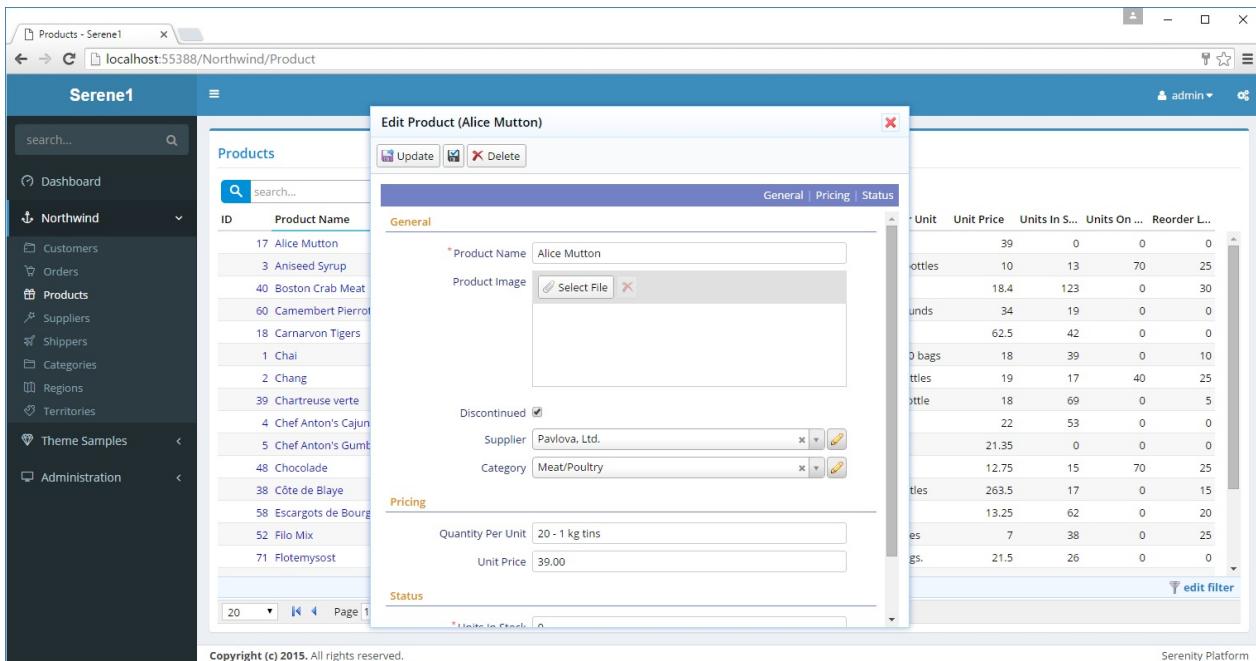
根据列类型，筛选条件有一些是简单的文本框，还有一些是下拉列表或其他自定义控件。

另外，也可以通过点击 *and* 把其修改为 *or*。

你还可以通过点击括号来调整筛选条件间的行距。

# 编辑对话框

当你在产品页面点击产品名称时，将显示该行的编辑对话框。



在客户端显示该对话框时，并没有 WebForm 那样的回发事件，点击产品名称后使用 AJAX 请求从服务器端加载该行的实体数据（仅返回数据，不包含页面标签）。对话框是自定义的 jQuery UI 对话框。

在此对话框窗体顶部有一蓝色背景导航栏，有三个分类导航项：*General*、*Pricing* 和 *Status*。通过点击此分类导航链接，可以导航到该分类的表单起始位置。

每个表单字段由 **label** 标签和输入控件（编辑器）水平排列成一行。如果需要的话，你也可以在一行中显示多个字段（需要调整 CSS）。

带有 “\*\*” 标记表示必填字段（不能为空）。

每个字段根据其数据类型有对应的输入控件，如字符串、图像上传、复选框、下拉列表等。

如果我们查看源代码，我们会看到这样的 HTML 代码（为了方便起见，做了简化）：

```
<div class="field ProductName">
    <label>Product Name</label>
    <input type="text" class="editor s-StringEditor" />
</div>

<div class="field ProductImage">
    <label class="caption"> Product Image</label>
    <div class="editor s-ImageUploadEditor">
        ...
    </div>
</div>

...
```

每个字段有一个独立的 "div" 标签，且该标签都含有 "field" 样式，该 "div" 标签内有一个 "label" 元素和另一个根据字段类型变化的元素（如：input, select, div）。

我们可以通过元素的类名称来标识输入控件的类型（例如，s-StringEditor, s-ImageUploadEditor）

在工具栏中，更新 按钮，保存并关闭当前实体对话框；应用更改 按钮，保存数据并保持对话框打开；删除 按钮，删除当前实体。

虽然你可以自定义按钮、字段、添加标签和其他元素，但很多 Serenity 编辑窗体都是类似这样的界面。

# 教程

- Movie 网站 (类似于IMDB)
- 多租户系统

# Movie 网站 (类似于IMDB)

让我们使用 Serenity 创建一些界面类似于 IMDB (IMDB: 互联网电影数据库，全称 The Internet Movie Database) 的网站。

你可以在这里找到本教程的源代码：

<https://github.com/volkanceylan/MovieTutorial>

## 创建一个名为 **MovieTutorial** 的新项目

在 Visual Studio 中选择 文件 -> 新建项目。选择 Serene 模板，在名称中输入 *MovieTutorial* 并单击确定按钮。

在解决方案资源管理器中，你应该看到一个名称为 *MovieTutorial.Web* 的项目。

*MovieTutorial.Web* 是一个 ASP.NET MVC 应用程序，包含服务器端代码以及静态资源，如 CSS 文件，图片等。

*MovieTutorial.Web* 项目根目录下有一个 *tsconfig.json* 文件，它表明该项目同时也是一个 TypeScript 项目。在 *Modules/* 和 *Scripts/* 目录下的所有 *.ts* 文件将在保存时被编译，并且它们输出的文件将合并到一个名为 *MovieTutorial.Web.js* 的文件，该文件位于 *scripts/site/* 文件夹内。

请确保你已安装 TypeScript 1.8.6+。

可从 <http://www.typescriptlang.org/#download-links> 下载适合 Visual Studio 的最新版本。

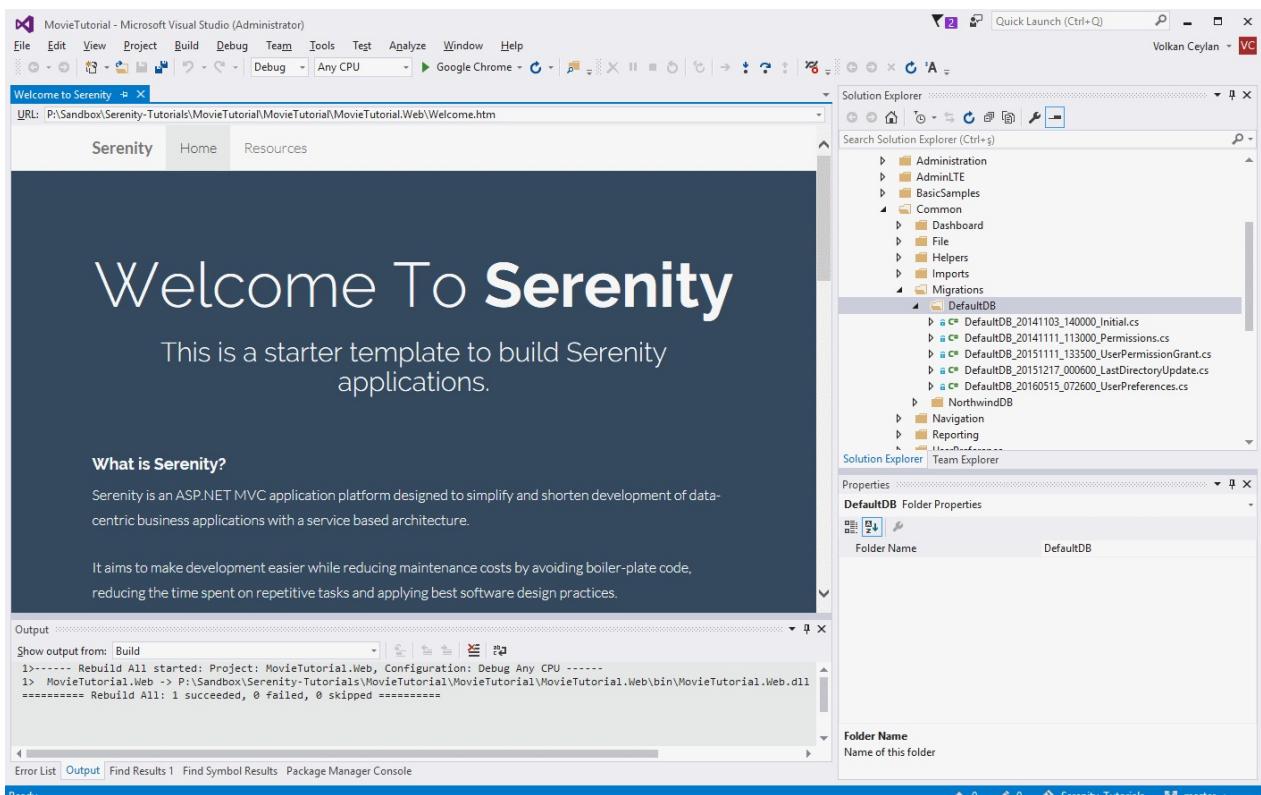
# 创建电影 (Movie) 表

要存储影片，我们需要一张电影 (Movie) 表。我们可以使用传统的方式创建该表，如使用 SQL Management Studio，但是我们更喜欢使用 *Fluent Migrator* 创建迁移类 (*migration*) 的方式创建表。

*Fluent Migrator* 是一个 .NET 迁移框架，它与 Ruby 的 Rails 迁移框架类似。迁移 (Migrations) 用结构化的方式改变数据库架构 (database schema)，替代创建大量必须通过开发人员手工运行的 sql 脚本。迁移解决由一个数据库架构演变为多个数据库的问题(例如，开发人员的本地数据库、测试数据库和生产数据库)。数据库架构的更改在一个类中使用 C# 描述，可以把该类签入到版本控制系统中。

关于 *FluentMigrator* 的更多信息，详见  
<https://github.com/schambers/fluentmigrator>。

在解决方案资源管理器 中导航到 *Modules / Common / Migrations / DefaultDB*。



在这里，我们已经有几个迁移类。一个迁移类就像一个操纵数据库结构的 DML 脚本。

以 `DefaultDB_20141103_140000_Initial.cs` 为例，它包含创建 `Northwind` 表和 `Users` 表的初始迁移内容。

在同一文件夹中，创建一个名为 `DefaultDB_20160519_115100_MovieTable.cs` 的文件。你可以拷贝一个现有的迁移类文件，然后重命名并修改内容。

迁移类文件名称/类名其实不重要，但建议你保持一致性并且正确地排序。

`20160519_115100` 对应于我们编写迁移类的 `yyyyMMdd_HHmmss` 格式的时间。它还将作为此迁移的唯一标识。

我们的迁移类文件看起来应该像下面这样：

## 创建电影 (Movie) 表

```
using FluentMigrator;
using System;

namespace MovieTutorial.Migrations.DefaultDB
{
    [Migration(20160519115100)]
    public class DefaultDB_20160519_115100_MovieTable : Migration
    {
        public override void Up()
        {
            Create.Schema("mov");

            Create.Table("Movie").InSchema("mov")
                .WithColumn("MovieId").AsInt32()
                    .Identity().PrimaryKey().NotNullable()
                .WithColumn("Title").AsString(200).NotNullable()
                .WithColumn("Description").AsString(1000).Nullab
le()
                .WithColumn("Storyline").AsString(Int32.MaxValue
).Nullable()
                    .WithColumn("Year").AsInt32().Nullable()
                .WithColumn("ReleaseDate").AsDateTime().Nullable
()
                .WithColumn("Runtime").AsInt32().Nullable();
        }

        public override void Down()
        {
        }
    }
}
```

确保你使用的命名空间是 MovieTutorial.Migrations.DefaultDB，因为 Serene 模板仅在此命名空间下对默认数据库应用迁移。

在 *Up()* 方法中，当应用指定的迁移时，数据库将创建名为 `mov` 的 **schema**。为避免与现有表发生冲突，我们为电影表使用一个独立的 **schema**。该方法将创建一个名为 `Movie` 的表，其字段有 "Movield, Title, Description..."。

我们可以实现 *Down()* 方法，以便能够撤消这个迁移(删除电影表和 **schema (mov)** 等)，但对于这个示例，我们让它空着。

不能撤消迁移可能并没有太大影响，但误删除表便会造成更大的损害。

在类的顶部，我们使用 **Migration** 特性。

```
[Migration("20160519115100")]
```

这里指定此迁移的唯一键。迁移应用到数据库后，在一个特定于 `FluentMigrator([dbo].[VersionInfo])` 的特殊表中记录该值，因此同一迁移不会被再次应用。

迁移的唯一键应该与类名同步(保持一致)，但没有强调迁移唯一键必须是 `Int64` 编号。

迁移是按唯一键顺序执行的，所以迁移的唯一键要使用一个可排序的日期时间模式，如把 `yyyyMMdd` 作为迁移唯一键就是个好主意。

请确保你始终使用相同数量的字符作为迁移的唯一键，如 14位 (`20160519115100`)。否则你的迁移命令会一团糟，由于使用混乱的顺序迁移，因此你会得到迁移错误。

## 执行迁移

默认情况下，`Serene` 模板将在应用程序启动时自动运行 `MovieTutorial.Migrations.DefaultDB` 命名空间下的所有迁移。运行迁移代码的是文件 `App_Start/SiteInitialization.cs` 和 `App_Start/SiteInitialization.Migrations.cs`：

### `SiteInitialization.Migrations.cs`:

```
namespace MovieTutorial
{
    // ...
}
```

## 创建电影 (Movie) 表

```
public static partial class SiteInitialization
{
    private static string[] databaseKeys = new[] { "Default",
        "Northwind" };

    //...
    private static void EnsureDatabase(string databaseKey)
    {
        //...
    }

    public static bool SkippedMigrations { get; private set;
    }

    private static void RunMigrations(string databaseKey)
    {
        // ...
        // safety check to ensure that we are not modifying
        an
        // arbitrary database. remove these two lines if you
        want
        // MovieTutorial migrations to run on your DB.
        if (cs.ConnectionString.IndexOf(typeof(SiteInitialization).Namespace +
            @"_" + databaseKey + "_v1",
            StringComparison.OrdinalIgnoreCase) < 0)
    {
        SkippedMigrations = true;
        return;
    }

    // ...

    using (var sw = new StringWriter())
    {
        // ...
        var runner = new RunnerContext(announcer)
        {
            Database = databaseType,
```

```

        Connection = cs.ConnectionString,
        Targets = new string[] {
            typeof(SiteInitialization).Assembly.Location },
        Task = "migrate:up",
        WorkingDirectory = Path.GetDirectoryName(
            typeof(SiteInitialization).Assembly.Location),
        Namespace = "MovieTutorial.Migrations." + databaseKey + "DB"
    };

    new TaskExecutor(runner).Execute();
}
}
}
}

```

这里有对数据库名称进行安全性检查，以避免在 Serene 默认数据库 (`MovieTutorial_Default_v1`) 之外的其他数据库上运行迁移。如果你了解其中的风险，可以删除此检查。例如，如果你把 `web.config` 中的默认连接更改为你自己的生产数据库，迁移将在生产数据库上运行，你将得到 `Northwind` 等表，即使之前你的生产数据库没有这些表。

现在按 F5 运行应用程序并在默认数据库中创建电影表。

## 确认迁移类已经运行

使用 Sql Server Management Studio 或 Visual Studio -> Connection To Database，打开数据库 `MovieTutorial_Default_v1 database in server (localdb)\v11.0` 的连接。

## 创建电影 (Movie) 表

(localdb)\v11.0 是 SQL Server 2012 LocalDB 创建的 LocalDB 实例。

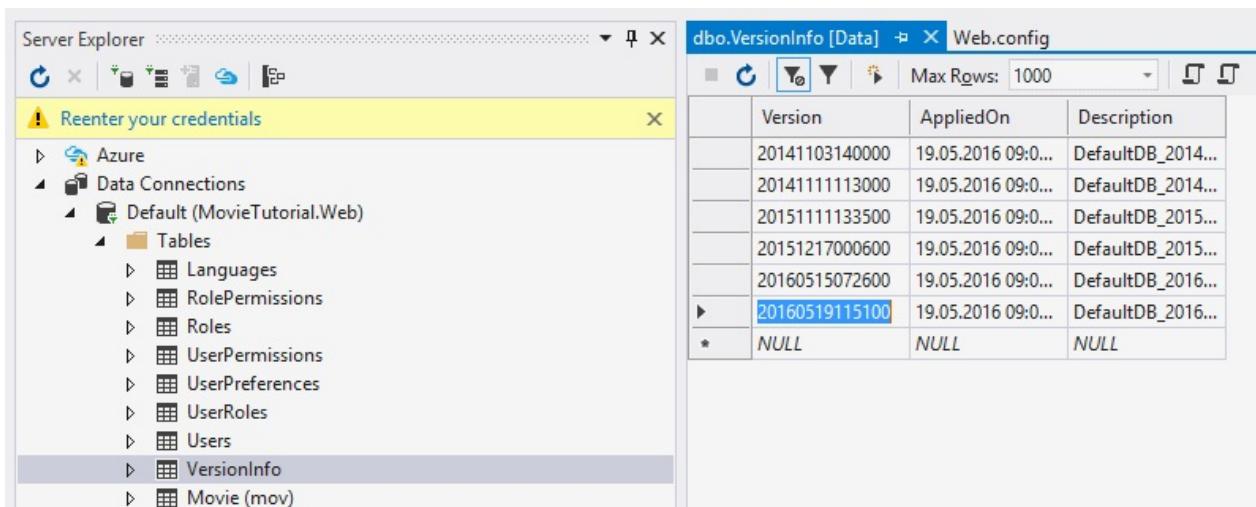
如果你还没有安装 LocalDB，可从 <https://www.microsoft.com/en-us/download/details.aspx?id=29062> 下载。

如果你有 SQL Server 2014 LocalDB，你的服务器名 (server name) 应该变为 (localdb)\MSSqlLocalDB 或者 (localdb)\v12.0，所以你应该修改 web.config 文件的连接字符串。

你也可以使用另外一个 SQL server 实例，只需把连接字符串配置为目标服务器，并删除迁移的安全检查。

你应该在 SQL 对象资源管理器中看到 [mov].[Movies] 表。

同样地，当你查看 [dbo].[VersionInfo] 表数据时，最后一行的 Version 列应为 20160519115100。它表明已经在此数据库执行该版本(迁移唯一键)的迁移。



The screenshot shows the SQL Server Object Explorer on the left and a data grid on the right. The Object Explorer displays a tree structure with nodes like Azure, Data Connections, Default (MovieTutorial.Web), Tables, Languages, RolePermissions, Roles, UserPermissions, UserPreferences, UserRoles, Users, VersionInfo, and Movie (mov). A yellow warning bar at the top of the Object Explorer says 'Reenter your credentials'. The data grid on the right is titled 'dbo.VersionInfo [Data]' and shows the following data:

Version	AppliedOn	Description
20141103140000	19.05.2016 09:0...	DefaultDB_2014...
20141111113000	19.05.2016 09:0...	DefaultDB_2014...
201511111133500	19.05.2016 09:0...	DefaultDB_2015...
20151217000600	19.05.2016 09:0...	DefaultDB_2015...
20160515072600	19.05.2016 09:0...	DefaultDB_2016...
20160519115100	19.05.2016 09:0...	DefaultDB_2016...
*	NULL	NULL

通常，你不必在每一个迁移后都做这些检查。在这里，我们只是演示如果你将来遇到迁移问题，你应该在到哪里去检查。

# 为影片 (Movie) 表生成代码

## Serenity 代码生成器

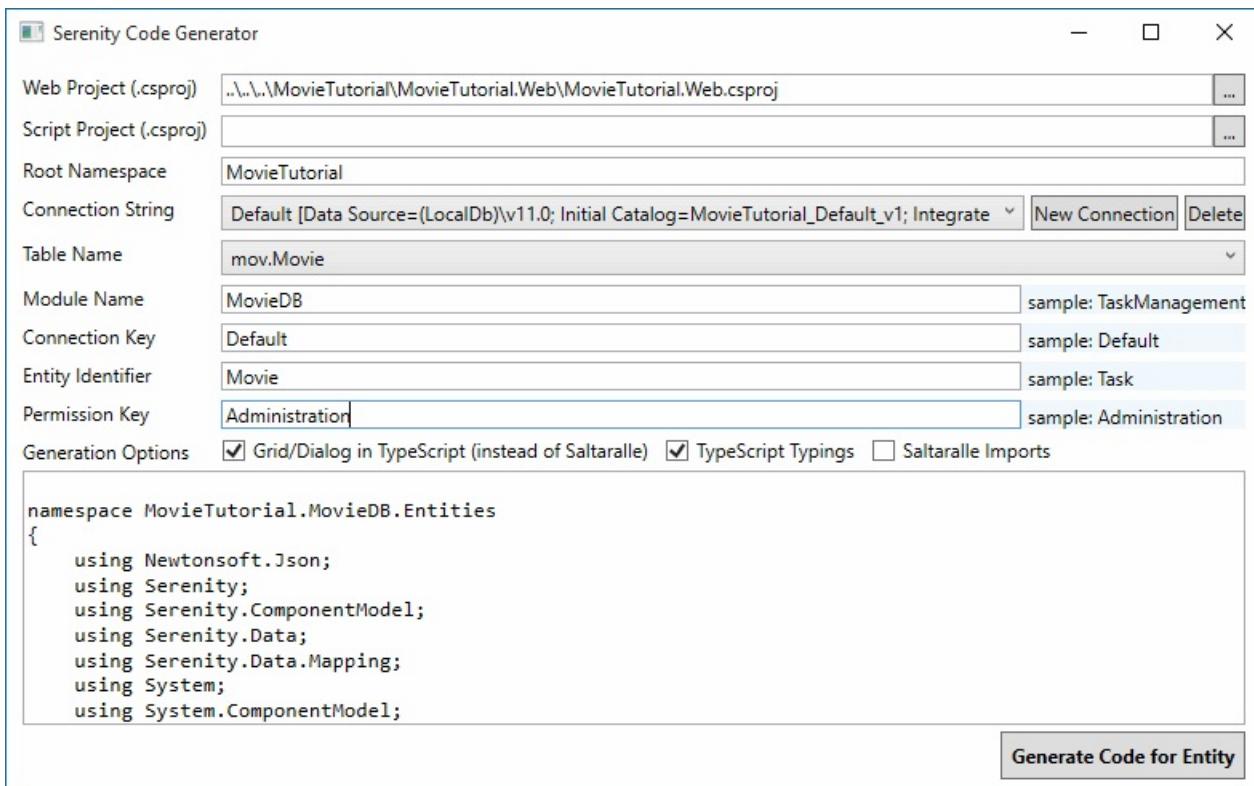
确保表存到数据库，我们将使用 Serenity 代码生成器 (sergen.exe) 生成初始的可编辑界面。

在 Visual Studio 中，通过点击 视图 => 其它窗体 => 包管理器控制台 打开 包管理器控制台。

输入 `sergen` 并按 Enter 键确认。

有时候包管理器控制台不能正确地设置路径，你可能会在执行 Sergen 的时候得到错误，重启 Visual Studio 可能解决该问题。

另一种解决方法是从资源管理器中打开 Sergen.exe。在解决方案资源管理器的 *MovieTutorial* 解决方案中右键，选择 文件资源管理器中打开，Sergen.exe 就在 *packages\Serenity.CodeGenerator.X.Y.Z\tools* 目录下面。



## 设置项目位置

当你第一次运行 Sergen，Web Project 字段默认为：

- ..\..\..\MovieTutorial\MovieTutorial.Web\MovieTutorial.Web.csproj

如果你把该值修改为其他路径，并且生成你的第一个页面，再次使用时，将不需要再次设置。所有的设置将保存在解决方案目录下的 `Serenity.CodeGenerator.config`。

该值是必填项，因为 Sergen 会借此把生成的文件自动包含到你的 WEB 项目。

对于仍包含 Saltaralle 编译器代码而不是 TypeScript 的 Serene 旧版本用户，在 v2.1+ 版本中，应该把 Script project 字段设置为空。

## Root Namespace 选项

你的 Root Namespace 选项设置为该解决方案使用的名称，如 `MovieTutorial`。如果你的项目名称为 `MyProject.Web`，你的 Root Namespace 则默认为 `MyProject`。

这是很关键的，所以确保你不能把它设置为其他值，因为默认情况下，Serene 模板期望所有生成的代码都在这个根命名空间下面。

该选项也被保存，因此下次使用时你将不用再次填写。

## 选择 Connection String

一旦设置 Web project 名称，Sergen 使用 `web.config` 的连接字符串中填充连接字符串下拉列表。我们这里有 `Default` 和 `Northwind`，选择 `Default`。

## 选择需要生成代码的表（Table Name）

Sergen 每次只为一张表生成代码。一旦我们选择了连接字符串，表下拉列表将会被所选数据库的表名填充。

选择 `Movie` 表

## 设置 Module Name

在 Serenity 术语中，一个模块（module）是一组逻辑页面（pages），共享同一个目标。

例如，在 Serene 模板中，所有与 *Northwind* 示例相关的页面都属于 *Northwind* 模块。

页面（Pages）是与网站的综合管理相关，如用户（users），角色（roles）等都属于 Administration 模块。

模块通常对应着数据库的 schema，或者单独的数据库，但没有什么可以阻止你在一个单独的数据库/schema 中使用多个模块，相反，多个数据库也可以使用一个模块。

本教程中，我们将为所有页面使用 MovieDB （类似于 IMDB）模块。

Module name 用于决定命名空间 和 生成页面的 url。

例如，我们的新页面会在 *MovieTutorial.MovieDB* 命名空间下，并使用 */MovieDB* 相对路径。

## ConnectionKey Parameter

Connection key 设置为选定 web.config 配置文件中连接字符串的连接键。你通常不需要更改它，让其保持默认值。

## Entity Identifier

这通常对应表名，但是有时候表名含有下划线或其他无效字符，所以需要你决定如何生成代码中的实体名称（一个有效的标识名称）。

我们的表名是 *Movie*，因此它是一个有效且不错的 C# 标识，所以我们把 *Movie* 作为实体标识。我们的实体类将被命名为 *MovieRow*。

该名称也被其他类使用。例如我们的页控制器（page controller）名称为 *MovieController*。

它还决定着页面的 url，在此示例中，我们的编辑页面的 URL 将为 */MovieDB/Movie*。

## Permission Key

Serenity 中，对资源（页、服务等）的访问控制都是由简单字符串组成的访问许可键（**permission keys**）控制。这些权限被授予用户（**users**）或角色（**roles**）。

我们的影片页面只有管理员用户（或者以后的内容编辑）可使用，因此，现在让我们把它设为 *Administration*，默认情况下，在 Serene 模板中只有 **admin** 用户有这个权限。

## 生成第一个页面代码

在设置如图所示的参数（你只需要设置 **Module Name**，其他使用默认值）之后，单击 **Generate Code for Entity** 按钮。

Sergen 将生成几个文件并将它们包含进 MovieTutorial.Web 和 MovieTutorial.Script 项目。

现在你可以关闭 Sergen，并返回到 Visual Studio。

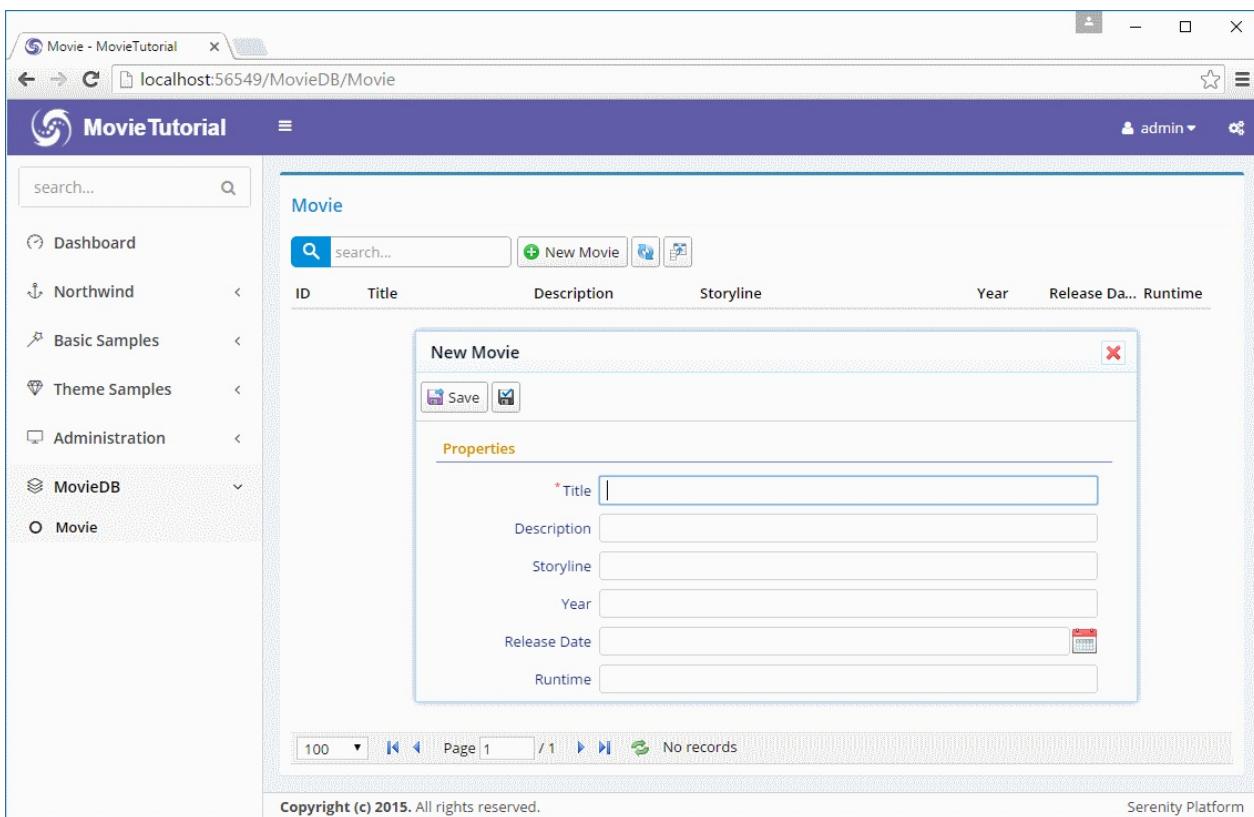
修改项目，Visual Studio 将询问你是否要重新加载更改，请单击加载所有。

重新生成解决方案，然后按 **F5** 启动应用程序。

使用 **admin** 作为用户名，并以 **serenity** 作为密码登录系统。

当你看到控制面板页面时，你会发现左侧导航的底部有一个新的菜单 **MovieDB**。

单击以展开它，然后单击 **Movie**，打开使用 Sergen 生成的第一个页面。



现在尝试添加一个新影片，然后更新并删除它。

我们不用写一行代码，使用 **Sergen** 为我们的表生成代码就可以工作。

这并不意味着我就不喜欢写代码，相反地我爱写代码。其实我并不是大部分设计师和代码生成器的粉丝。他们生产的代码通常都是混乱难以管理的。

**Sergen** 在这里仅是帮我们初始设置所需的分层结构和平台标准。我们将创建 10 个左右的文件，如实体（entity）、仓储（repository）、页面（page）、终结点（endpoint）、网格列表（grid）、表单（form）等，同时还需要在其他地方做一些设置。

即使我们从其他页面复制粘贴并替换代码，它也可能会出错，且至少需要 5-10 分钟。

**Sergen** 生成的代码文件中含最基本的少量代码，这得多亏 **Serenity** 的基类，它处理了大部分逻辑。一旦我们为一些表生成代码，我们可能就不会再对此表使用 **Sergen**，我们将在后面章节看到如何修改生成的代码，使之变为我们需要的。

# 自定义影片界面

## 自定义字段标题

在我们影片列表和表单中，有一个叫 **Runtime** 的字段。该字段需要一个整数，表示影片时长 (*minutes*)，但在其标题描述中并没有该提示信息。让我们把其标题改为 Runtime (mins)。

有几种方法可以修改标题内容：修改服务器端表单的定义、修改服务器端列的定义、修改网格列表的脚本代码等。但让我们在核心的位置做修改，即修改实体本身，这样实体的标题在所有使用的地方都将得到修改。

当 Sergen 为电影 (Movie) 表生成代码时，创建了一个叫 MovieRow 的实体类，你可以在 *Modules/MovieDB/Movie/MovieRow.cs* 找到它。

这是摘自源文件的 Runtime 属性：

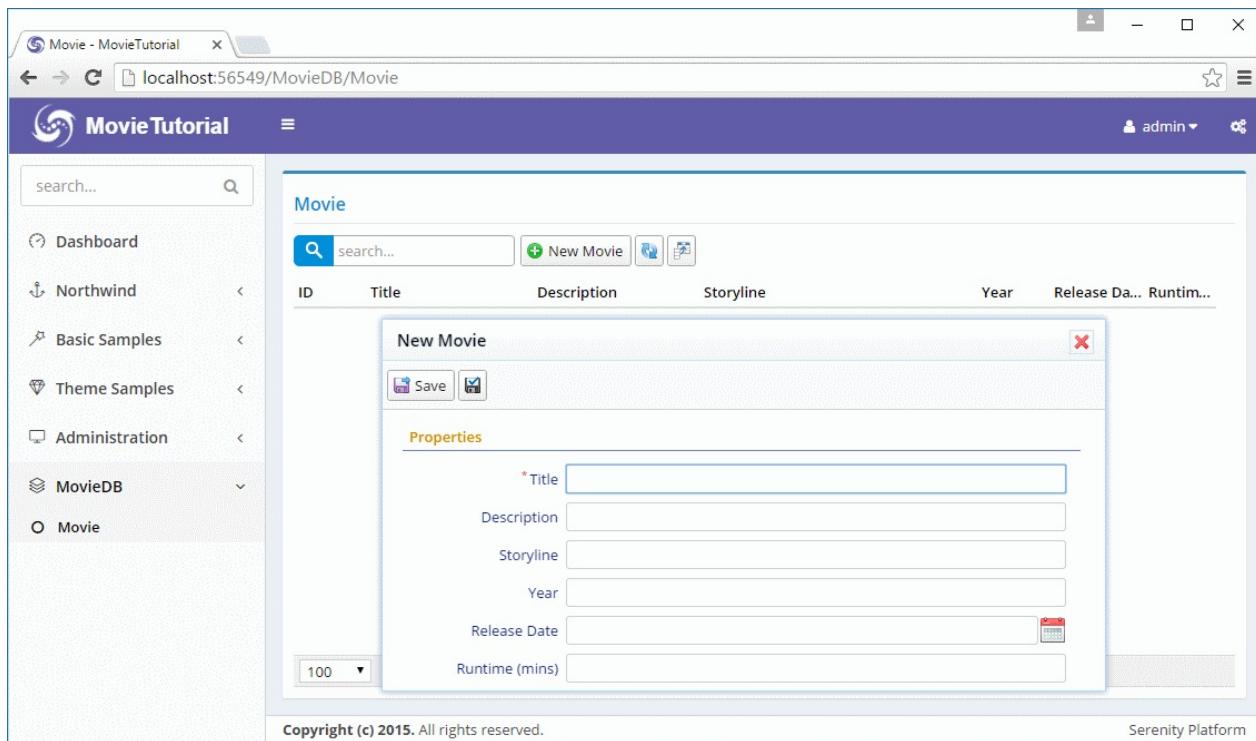
```
namespace MovieTutorial.MovieDB.Entities
{
    // ...
    [ConnectionKey("Default"), DisplayName("Movie"),
     InstanceName("Movie"), TwoLevelCached]
    public sealed class MovieRow : Row, IIdRow, INameRow
    {
        // ...
        [DisplayName("Runtime")]
        public Int32? Runtime
        {
            get { return Fields.Runtime[this]; }
            set { Fields.Runtime[this] = value; }
        }
        //...
    }
}
```

我们会在后面讨论 `entities`（或 `rows`），现在让我们把注意力集中在更改字段名称的目标上。我们把 `Runtime` 属性的 `DisplayName` 特性的值修改为 `*Runtime (mins)`：

```
namespace MovieTutorial.MovieDB.Entities
{
    // ...
    [ConnectionKey("Default"), DisplayName("Movie"), InstanceName("Movie"),
     TwoLevelCached]
    public sealed class MovieRow : Row, IIdRow, INameRow
    {
        // ...
        [DisplayName("Runtime (mins)")]
        public Int32? Runtime
        {
            get { return Fields.Runtime[this]; }
            set { Fields.Runtime[this] = value; }
        }
        //...
    }
}
```

现在生成并运行应用程序，你会在网格列表和对话框中看到字段标题的变化。

在列不够宽时，列标题有 "... "，但是它会在提示信息中显示完整的标题。我们很快就会看到这是如何处理的。



## 重写列的标题和宽度

到目前为止，一切看起来都还不错，如果我们想在网格列表（columns）或对话框（form）中显示另一个标题。我们可以重写它对应的定义文件。

我们先从 columns 开始，在 *MovieRow.cs* 旁边，你可以找到名为 *MovieColumns.cs* 的文件：

```
namespace MovieTutorial.MovieDB.Columns
{
    // ...

    [ColumnsScript("MovieDB.Movie")]
    [BasedOnRow(typeof(Entities.MovieRow))]
    public class MovieColumns
    {
        [EditLink, DisplayName("Db.Shared.RecordId"), AlignRight]
        public Int32 MovieId { get; set; }
        //...
        public Int32 Runtime { get; set; }
    }
}
```

你可能会注意到这个 `columns` 的定义是基于电影实体（`BasedOnRow` 特性）。

任何写在这里的特性都将重写定义在实体类中的特性。

让我们把 `DisplayName` 特性添加到 `Runtime` 属性：

```
namespace MovieTutorial.MovieDB.Columns
{
    // ...

    [ColumnsScript("MovieDB.Movie")]
    [BasedOnRow(typeof(Entities.MovieRow))]
    public class MovieColumns
    {
        [EditLink, DisplayName("Db.Shared.RecordId"), AlignRight]
        public Int32 MovieId { get; set; }
        //...
        [DisplayName("Runtime in Minutes"), Width(150), AlignRight]
        public Int32 Runtime { get; set; }
    }
}
```

现在，我们把列标题设置为 "Runtime in Minutes"。

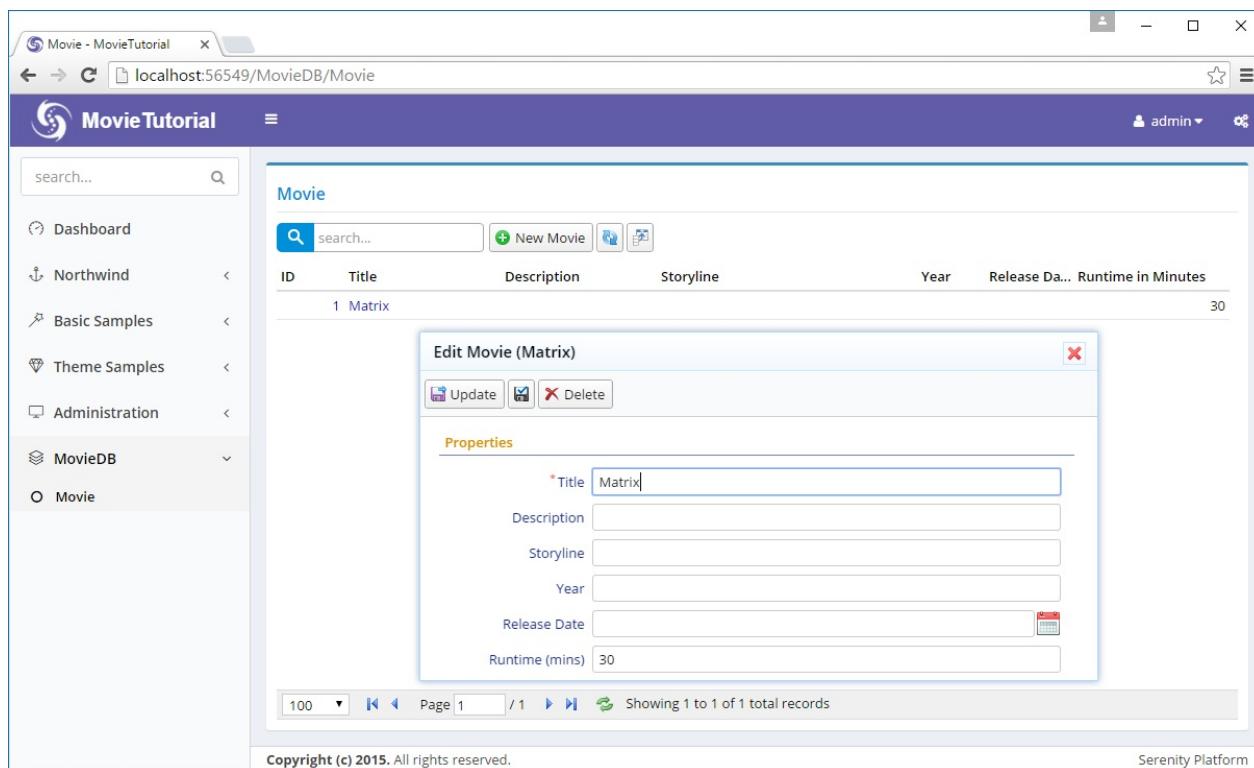
实际上，我们添加了更多的特性。

其中，重写列的宽度为 150px。

Serenity 将根据字段类型和字符串长度自动设置列的宽度，除非你显式设置了宽度。

并且把列设为右对齐（还可设为居中对齐、左对齐）

再次生成并运行项目，我们将看到：



表单的标题保持原样，而网格的列标题已经发生了变化。

相反，如果我们想重写表单字段，我们就在 `MovieForm.cs` 做类似的步骤。

## 修改描述（**Description**）和剧情（**Storyline**）的编辑器类型

描述（**Description**）和剧情（**Storyline**）比标题（**Title**）字段长一点，因此，让我们把他们的编辑器类型改为文本域（`textarea`）。

在 `MovieColumns.cs` 和 `MovieRow.cs` 的同一文件夹中，打开 `MovieForm.cs`。

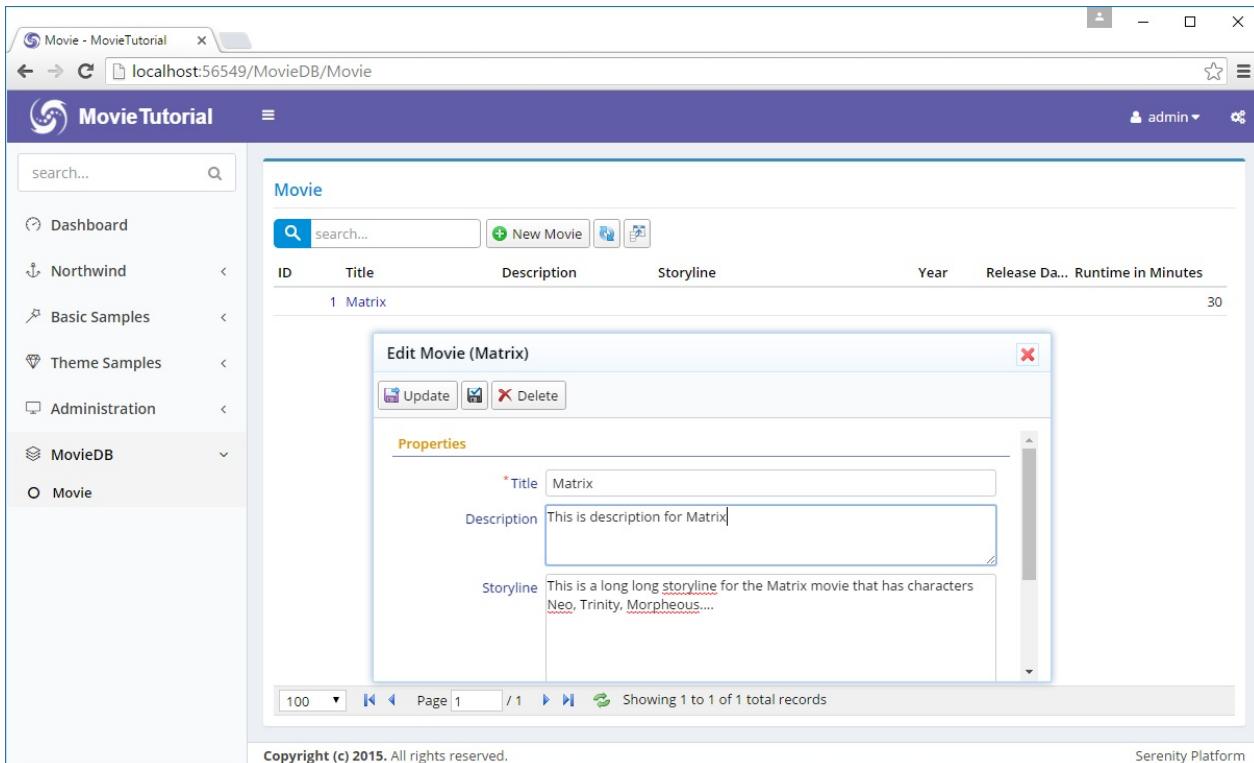
```
namespace MovieTutorial.MovieDB.Forms
{
    //...
    [FormScript("MovieDB.Movie")]
    [BasedOnRow(typeof(Entities.MovieRow))]
    public class MovieForm
    {
        public String Title { get; set; }
        public String Description { get; set; }
        public String Storyline { get; set; }
        public Int32 Year { get; set; }
        public DateTime ReleaseDate { get; set; }
        public Int32 Runtime { get; set; }
    }
}
```

在这两个属性中添加 `TextAreaEditor` 特性：

```
namespace MovieTutorial.MovieDB.Forms
{
    //...
    [FormScript("MovieDB.Movie")]
    [BasedOnRow(typeof(Entities.MovieRow))]
    public class MovieForm
    {
        public String Title { get; set; }
        [TextAreaEditor(Rows = 3)]
        public String Description { get; set; }
        [TextAreaEditor(Rows = 8)]
        public String Storyline { get; set; }
        public Int32 Year { get; set; }
        public DateTime ReleaseDate { get; set; }
        public Int32 Runtime { get; set; }
    }
}
```

我为剧情（8 行）分配了更多的编辑行，因为与描述（3 行）相比剧情应该更长。

生成并运行项目后，我们得到：



Serene 有几个编辑器类型可供选择。有些是基于字段的数据类型自动选取，而你需要显式地设置它们。

你还可以开发自己的编辑器类型。你可以把现有的编辑器类作为基类，或者从 scratch 中开发自己的编辑器，我们会在后面的章节介绍。

由于编辑器有点高，表单高度超出了 Serenity 默认的表单高度(约260px)，所以现在我们有一个垂直滚动条，让我们删除这个垂直滚动条。

## 使用 **CSS (Less)** 设置对话框的初始大小

Sergen 在 `MovieTutorial.Web/Content/site/site.less` 文件中生成一些影片对话框的 CSS。

如果你打开该文件，并滚动到底部，你将看到：

```
/* -----
----- */
/* APPENDED BY CODE GENERATOR, MOVE TO CORRECT PLACE AND REMOVE
THIS COMMENT */
/* -----
----- */

.s-MovieDB-MovieDialog {
    > .size { width: 650px; }
    .caption { width: 150px; }
}
```

你可以安全地删除这三行注释（由代码生成器添加）。这是提醒你将它们移动到更好的地方，如移到特定于此模块的 `site.movies.less` 文件（推荐）。

这些规则将应用于有 `.s-MovieDB-MovieDialog` 样式类的元素。在默认情况下，我们的对话框具有这样的样式类：`"s-" + 模块名 + "-" + 类名`。

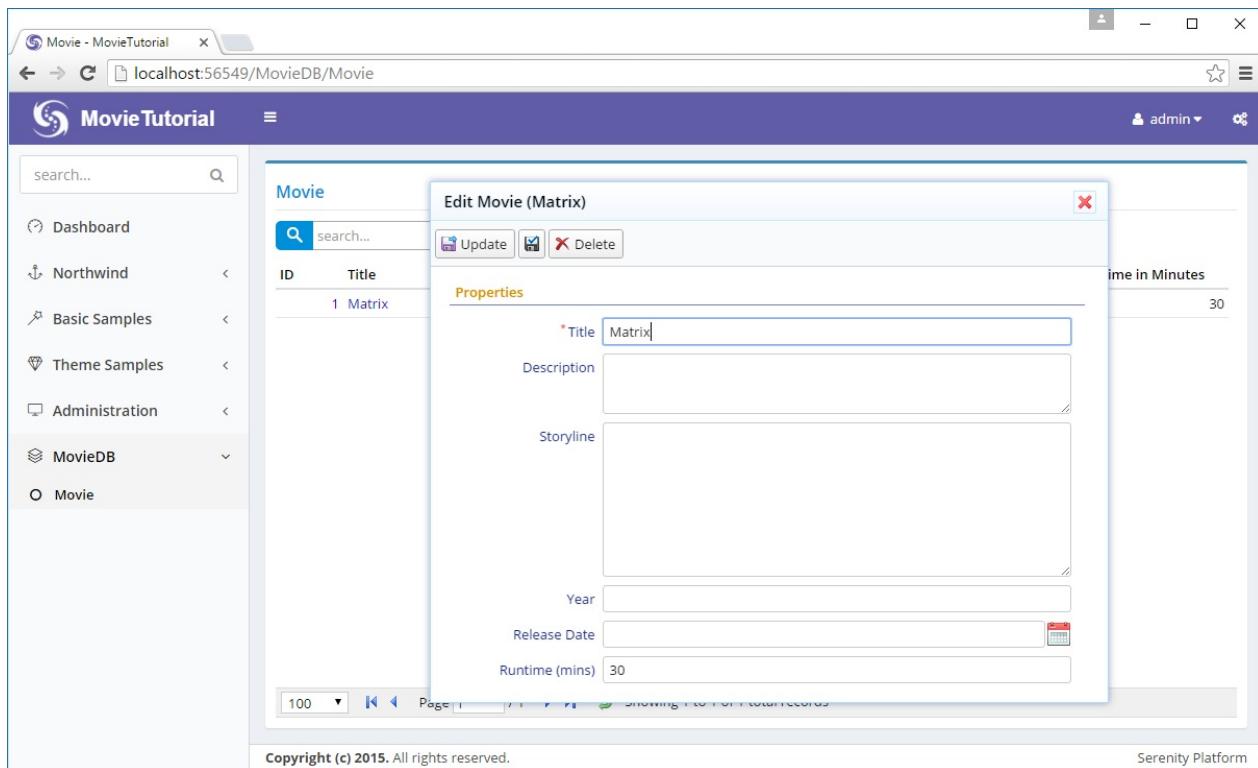
在第二行中指定该对话框的默认宽度是 `650px`。

在第三行中，我们指定该字段标签长度应该是 `150px(@l: 150px)`。

让我们改变对话框初始高度为 `500px`(桌面模式)，这样它就不会出现垂直滚动条：

```
.s-MovieDialog {
    > .size { width: 650px); height: 500px; }
    .caption { width: 150px; }
}
```

要将此更改应用到你的对话框，你需要生成解决方案。在生成项目时，“`site.less`”文件被编译到一个 “`site.css`” 文件。现在生成项目并刷新页面。



上面所说的 桌面模式 将很快变得直观。Serenity 对话框默认是响应式的。让我们把浏览器窗口的宽度调整为 350px 左右。我会用 Chrome 浏览器的移动模式切换到 iphone 6：

## 自定义影片界面

A screenshot of a browser window titled "Movie - MovieTutorial" showing a "Edit Movie (Matrix)" form. The form has fields for Title, Description, Storyline, Year, Release Date, and Runtime (mins). The browser's developer tools Network tab is open, showing a list of requests with their names, start times, types, and durations. The requests include "Lookup.Ad...", "datepicker....", "Retrieve", "cross.png", "disk-arrow....", "disk-black-...", "cross-script....", and "back-arrow....". The timeline shows the sequence of these requests.

现在改为 iPad 横向模式：

A screenshot of the same "Edit Movie (Matrix)" form displayed in横向 mode on an iPad. The form fields are identical to the iPhone version. The developer tools Network tab is also visible, showing the same list of requests and timeline as the iPhone screenshot.

因此，我们在这里对桌面模式设置的高度是很有用意的。对话框将变为响应式，不需要使用 CSS 的 @media 查询就可对移动设备特定大小模式保持敏感的响应。

## 修改页面标题

我们的页面有标题 *Movie*，让我们把它修改为 *Movies*。

再次打开 *MovieRow.cs*。

```
namespace MovieTutorial.MovieDB.Entities
{
    // ...
    [ConnectionKey("Default"), DisplayName("Movie"), InstanceName("Movie"),
     TwoLevelCached]
    public sealed class MovieRow : Row, IIdRow, INameRow
    {
        [DisplayName("Movie Id"), Identity]
        public Int32? MovieId
```

把 *DisplayName* 特性的值改为 *Movies*。这是引用此表时使用的名称，它通常是复数。此特性用于确定默认页面的标题。

也可以在 *MoviePage.Index.cshtml* 文件中重写页面标题。但像往常一样，我们习惯从核心着手，以便在其它地方可以重用这些信息。

实例名对应单数名称，并在网格的新增记录（新电影）按钮中使用，也决定了对话框标题（例如编辑电影）。

```
namespace MovieTutorial.MovieDB.Entities
{
    // ...
    [ConnectionKey("Default"), DisplayName("Movies"), InstanceName("Movie"),
     TwoLevelCached]
    public sealed class MovieRow : Row, IIdRow, INameRow
    {
        [DisplayName("Movie Id"), Identity]
        public Int32? MovieId
```

# 处理 Movie 的导航

## 设置导航项标题和图标

当 Sergen 为影片（Movie）表生成代码时，它同时也创建了一个导航项目实体。在 Serene 中，导航项目使用专门的程序集特性创建。

在同一文件夹中打开 `MoviePage.cs`，在顶部你将找到下面这行代码：

```
[assembly:Serenity.Navigation.NavigationLink(int.MaxValue, "MovieDB/Movie",
    typeof(MovieTutorial.MovieDB.Pages.MovieController))]

namespace MovieTutorial.MovieDB.Pages
{
    //...
```

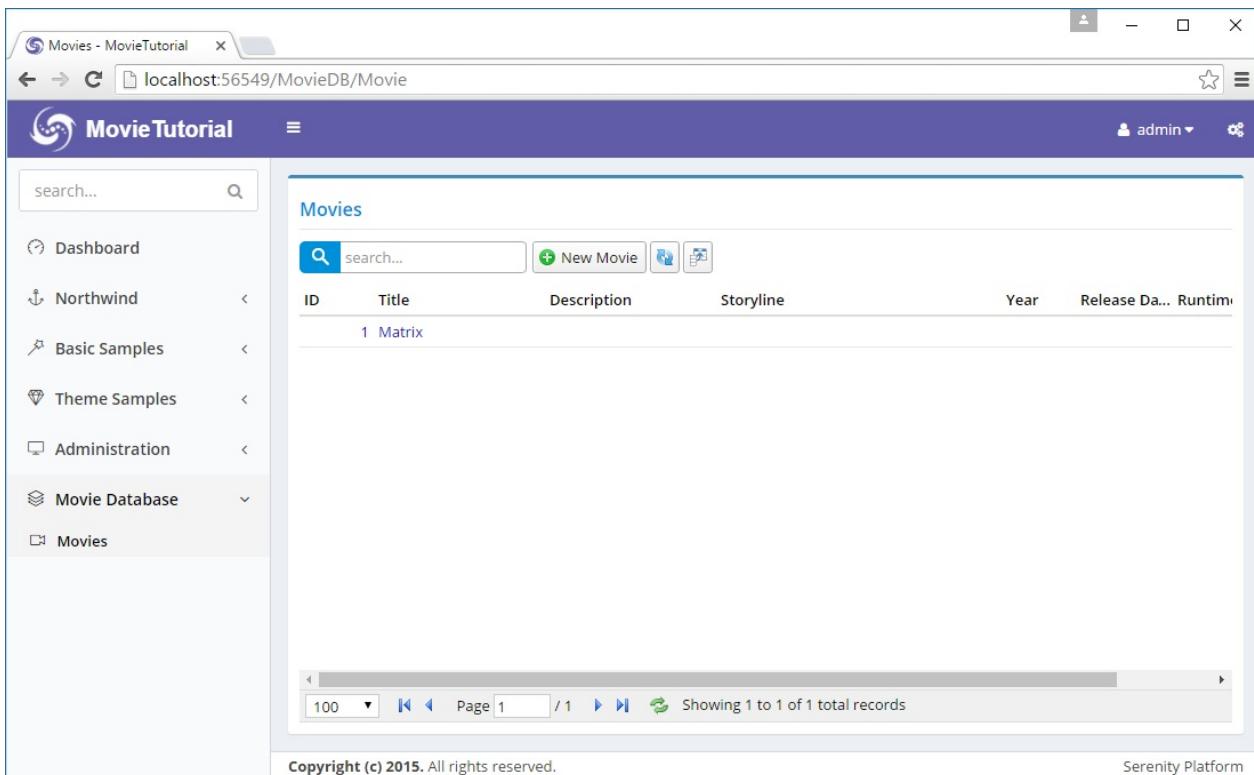
该特性的第一个参数是该导航项目的显示次序。因为我们在 Movie 菜单中只有一个导航项目，所以我们不需要次序。

第二个参数是“节（Section）标题/连接（Link）标题”格式的导航标题。节和导航项目是以斜杠(/)分隔。

让我们将其更改为 `Movie Database/Movies`。

```
[assembly:Serenity.Navigation.NavigationLink(int.MaxValue, "Movie Database/Movies",
    typeof(MovieTutorial.MovieDB.Pages.MovieController), icon: "icon-camrecorder")]

namespace MovieTutorial.MovieDB.Pages
{
    //...
```



我们也将导航项的图标修改为 *icon-camcorder*。Serene 模板有两种字体图标的设计：

更多 simple line icons 和其 css 类，请访问下面的连接：

<http://thesabbr.github.io/simple-line-icons/>

FontAwesome 可在这里查看：

<https://fontawesome.github.io/Font-Awesome/icons/>

Serene 中 *Theme Samples / UI Elements / Icons* 下有一个包含这些图标集的列表页面。

## 对导航节（Navigation Sections）排序

由于我们的 *Movie Database* 节是最后自动生成的，所以它显示在导航菜单的底部。

我们将把它移到 *Northwind* 菜单前面。

正如我们最近看到的，Sergen 在 *MoviePage.cs* 创建了一个导航项目。如果导航项目都分散到像这样的页面，将很难看到大图(所有导航项目的列表)并难以对它们排序。

因此我们把它移到一个集中的位

置：*MovieTutorial.Web/Modules/Common/Navigation/NavigationItems.cs*。

我们只须从 *MoviePage.cs* 剪切下面的行：

```
[assembly: Serenity.Navigation.NavigationLink(int.MaxValue, "Movie Database/Movies",
    typeof(MovieTutorial.MovieDB.Pages.MovieController), icon: "icon-camrecorder")]
```

把它移到 *NavigationItems.cs*，并作如下修改：

```
using Serenity.Navigation;
using Northwind = MovieTutorial.Northwind.Pages;
using Administration = MovieTutorial.Administration.Pages;
using MovieDB = MovieTutorial.MovieDB.Pages;

[assembly: NavigationLink(1000, "Dashboard", url: "~/", permission: "",
    icon: "icon-speedometer")]

[assembly: NavigationMenu(2000, "Movie Database", icon: "icon-film")]
[assembly: NavigationLink(2100, "Movie Database/Movies",
    typeof(MovieDB.MovieController), icon: "icon-camcorder")]

[assembly: NavigationMenu(8000, "Northwind", icon: "icon-anchor")]
[assembly: NavigationLink(8200, "Northwind/Customers",
    typeof(Northwind.CustomerController), icon: "icon-wallet")]
[assembly: NavigationLink(8300, "Northwind/Products",
    typeof(Northwind.ProductController), icon: "icon-present")]
// ...
```

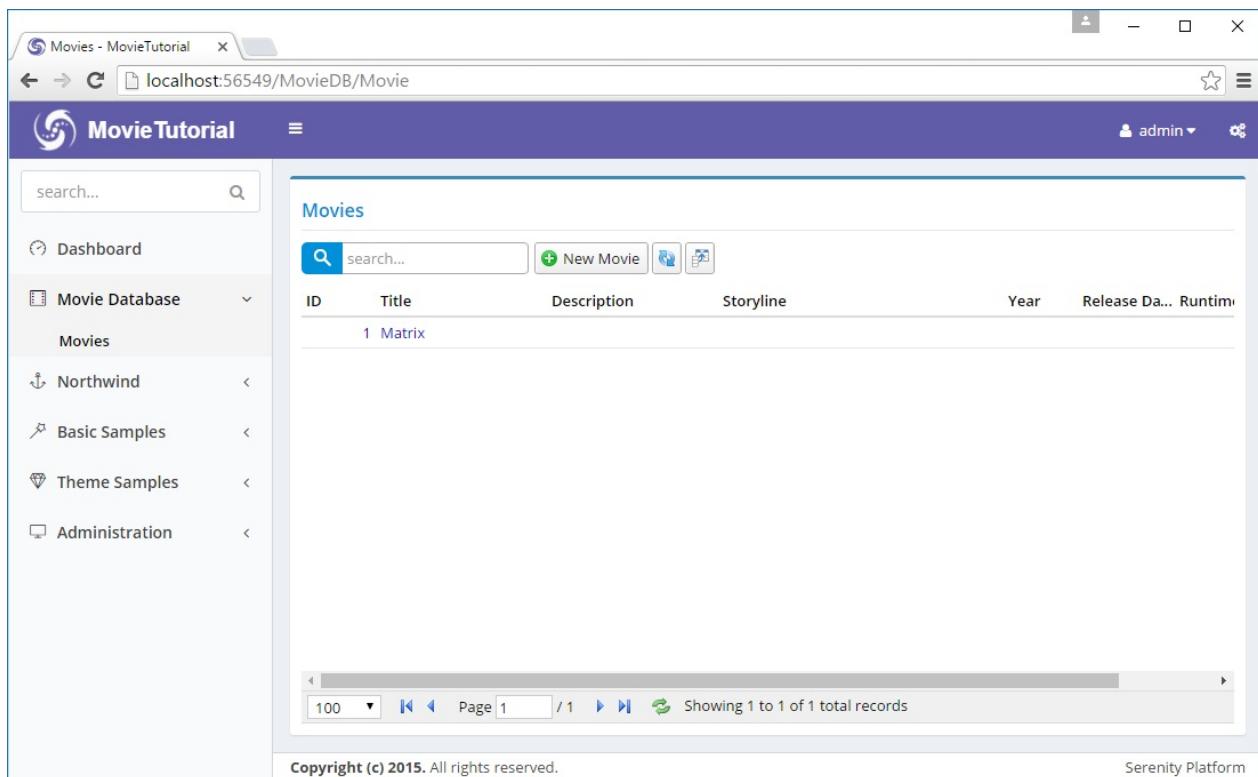
我们在这里还声明导航菜单(Movie Database)和使用 *film* 图标。当你没有明确定义导航菜单，Serenity 将隐式创建一个菜单，但在这种情况下，你不能自己对菜单排序或设置菜单图标。

我们指定它的显示顺序为 2000，所以这个菜单将显示在 Dashboard(1000) 连接之后，但在 Northwind(8000) 菜单之前。

我们指定 **Movies** 连接的显示顺序为 2100，但现在也不重要了，因为我们 **Movie Database** 菜单下只有一个导航项目。

第一级的连接和导航菜单首先根据它们的显示顺序排序，然后第二级兄弟结点间的连接排序。

这是更改之后的样子：



## 使用 Visual Studio 解决一些问题

你可能已经注意到，Visual Studio 在站点运行时，不允许你修改代码。当你停止调试时，网站也停止了，所以你不能在重新生成之后打开浏览器并刷新页面。

为了解决这个问题，我们需要禁用 编辑并继续（*Edit And Continue*）（不知道为什么）。

在 **MovieTutorial.Web** 项目中右键，点击 属性，在 **Web** 选项卡中，取消 *Enable Edit And Continue under Debuggers*。

不幸的是，在 visual studio 2015 Update 2 中该方法不能工作。唯一的变通方案是在不调试状态下执行，例如，使用 **Ctrl+F5** 替换 **F5**。

此外，在这种情况下，你的网站顶部的蓝色进度栏（即 Pace.js 动画）总是保持运行就像它仍在加载东西。这是由于 Visual Studio 的 *Browser Link* 功能。要禁用它，需要在 Visual Studio 工具栏找到看上去像刷新的按钮（在含 Chrome 名称的调试图标旁边），单击其下拉列表，并取消勾选 *Enable Browser Link*。

也可以在 `web.config` 设置中禁用它。

```
<appsettings>
  <add key="vs:EnableBrowserLink" value="false" />
</appsettings>
```

Serene 1.5.4 及后续版本都已经在 `web.config` 中默认配置了此设置，因此你将不会遇到这样的问题。

# 自定义快速检索

## 添加几条影片记录

在下面的章节中，我们需要一些示例数据，可以从 IMDB 复制一些数据过来。

如果你不想浪费时间输入这些示例数据，可以从下面的链接中获取迁移类：

[https://github.com/volkanceylan/MovieTutorial/blob/master/MovieTutorial/MovieTutorial.Web/Modules/Common/Migrations/DefaultDB/DefaultDB\\_20160519\\_135200\\_SampleMovies.cs](https://github.com/volkanceylan/MovieTutorial/blob/master/MovieTutorial/MovieTutorial.Web/Modules/Common/Migrations/DefaultDB/DefaultDB_20160519_135200_SampleMovies.cs)

ID	Title	Description	Storyline	Year	Release Da...	Runtime in Minutes
8	Fight Club	An insomniac office w...	A ticking-time-bomb insomniac and a s...	1999	10/15/1999	139
6	Pulp Fiction	The lives of two mob ...	Jules Winnfield and Vincent Vega are t...	1994	10/14/1994	154
5	The Godfather	The aging patriarch of...	When the aging head of a famous crim...	1972	03/24/1972	175
7	The Good, the Bad an...	A bounty hunting sca...	Blondie (The Good) is a professional g...	1969	01/13/1969	161
3	The Lord of the Rings:...	A meek hobbit of the ...	An ancient Ring thought lost for centu...	2001	12/19/2001	178
2	The Matrix	A computer hacker le...	Thomas A. Anderson is a man living tw...	1999	03/31/1999	136
4	The Shawshank Rede...	Two imprisoned men ...	Andy Dufresne is a young and success...	1994	10/14/1994	142

Copyright (c) 2015. All rights reserved. Serenity Platform

如果我们在搜索框中输入 *go*，我们会看到两部影片被筛选出来：*The Good, the Bad and the Ugly* 和 *The Godfather*。

如果我们输入 *Gandalf*，则不会找到任何影片。

默认情况下，Sergen 把表的第一个文本字段作为 名称 字段。在电影（movies）表中是标题（*Title*），该字段有一个 *QuickSearch* 特性，它表示应该在此字段中执行文本搜索。

名称字段还决定初始的排序顺序和编辑对话框显示的标题。

有时候，第一个文本列可能不是名称字段。如果想更改为搜索另一个字段，你应该在 *MovieRow.cs* 中修改：

```
namespace MovieTutorial.MovieDB.Entities
{
    //...
    public sealed class MovieRow : Row, IIdRow, INameRow
    {
        //...
        StringField INameRow.NameField
        {
            get { return Fields.Title; }
        }
    }
}
```

代码生成器决定表的第一个文本（`string`）字段是标题。所以它添加一个 `INameRow` 接口到 `MovieRow` 类，并通过返回标题（`Title`）字段实现该接口。如果想要使用描述（`Description`）作为名称字段，我们只须做对应的替换即可。

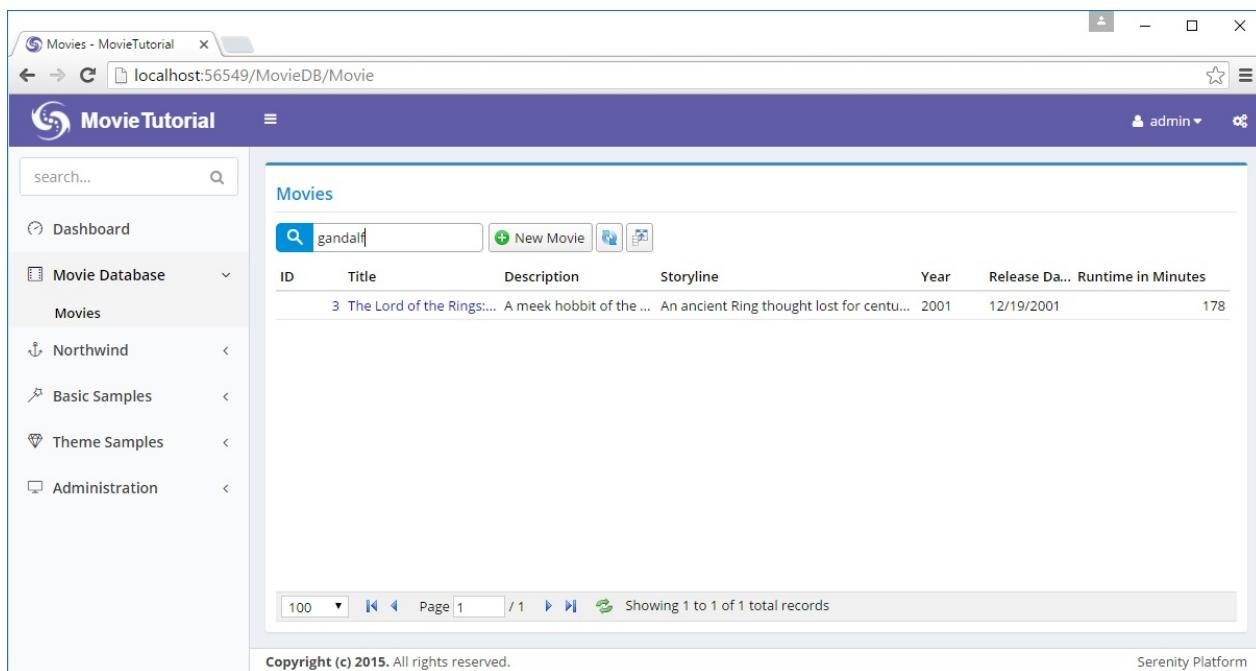
在这里，实际上 `Title` 是 `name` 字段，所以我们让它保持原样。但若想让 Serenity 对 `Description` 和 `Storyline` 字段搜索，我们需要对这两个字段添加 `QuickSearch` 特性，如下所示：

```
namespace MovieTutorial.MovieDB.Entities
{
    //...
    public sealed class MovieRow : Row, IIdRow, INameRow
    {
        //...
        [DisplayName("Title"), Size(200), NotNull, QuickSearch]
        public String Title
        {
            get { return Fields.Title[this]; }
            set { Fields.Title[this] = value; }
        }

        [DisplayName("Description"), Size(1000), QuickSearch]
        public String Description
        {
            get { return Fields.Description[this]; }
            set { Fields.Description[this] = value; }
        }

        [DisplayName("Storyline"), QuickSearch]
        public String Storyline
        {
            get { return Fields.Storyline[this]; }
            set { Fields.Storyline[this] = value; }
        }
        //...
    }
}
```

现在，如果搜索 *Gandalf*，我们将得到 *The Lord of the Rings* 实体：



默认情况下，QuickSearch 特性以 包含 的形式进行过滤。它有一些选项设置其匹配 StartsWith 过滤器或只匹配精确值。

如果我们希望它只显示 *StartsWith* 输入文本的过滤结果行，我们将特性更改为：

```
[DisplayName("Title"), Size(200), NotNull, QuickSearch(SearchType.StartsWith)]
public String Title
{
    get { return Fields.Title[this]; }
    set { Fields.Title[this] = value; }
}
```

在这里，这种快速搜索功能不是非常有用，但对于像 SSN、序列号、身份证号码、电话号码等类型的值将很有效。

如果我们还想在 year 列进行搜索，但只限于确切的整数年份值(匹配 1999，不匹配 19)：

```
[DisplayName("Year"), QuickSearch(SearchType.Equals, numericOnly : 1)]
public Int32? Year
{
    get { return Fields.Year[this]; }
    set { Fields.Year[this] = value; }
}
```

你可能已经注意到，我们不用为这些基本特性的工作写任何 C# 或 SQL 代码。我们只需指定我们想要的东西，而不是如何实现，这是就是声明式的编程。

也可以由用户决定搜索字段。

打开 `MovieTutorial.Web/Modules/MovieDB/Movie/MovieGrid.ts`，并做如下修改：

```
namespace MovieTutorial.MovieDB {

    @Serenity.Decorators.registerClass()
    export class MovieGrid extends
        Serenity.EntityGrid<MovieRow, any> {
        //...
        constructor(container: JQuery) {
            super(container);
        }

        protected getQuickSearchFields():
            Serenity.QuickSearchField[] {
            return [
                { name: "", title: "all" },
                { name: "Description", title: "description" },
                { name: "Storyline", title: "storyline" },
                { name: "Year", title: "year" }
            ];
        }
    }
}
```

一旦你保存该文件，我们的快速搜索将变成以下拉列表的形式输入：

The screenshot shows a web application titled "MovieTutorial" running on "localhost:56549/MovieDB/Movie". The left sidebar has a "Dashboard" section and a "Movie Database" dropdown menu with options like "Movies", "Northwind", "Basic Samples", "Theme Samples", and "Administration". The main content area is titled "Movies" and contains a search bar with dropdown filters for "all", "description", "storyline", and "year". A table lists movie details with columns: ID, Title, Description, Storyline, Year, Release Date, and Runtime in Minutes. The table includes rows for "Fight Club", "Pulp Fiction", "The Godfather", "The Good, the Bad and the Ugly", "The Lord of the Rings: The Fellowship of the Ring", "The Matrix", and "The Shawshank Redemption". At the bottom, there's a navigation bar with "100", a page number input set to "1", and a message "Showing 1 to 7 of 7 total records". The footer says "Copyright (c) 2015. All rights reserved." and "Serenity Platform".

与之前我们修改服务器端代码的示例不同，这一次我们在客户端仅修改 Javascript (TypeScript) 代码。

## 运行 T4 模板 (.tt 文件)

我们在之前的示例中硬编码 *Description*、*Storyline* 等字段，如果我们忘了属性的名称或属性在服务器端的大小写 (javascript 区分大小写)，就可能导致输入错误。

Serene 包含一些 T4(.tt) 文件，可以将此类信息从服务器端 (C# 中的 `rows` 等) 转移到客户端 (TypeScript) 用于智能提示。

运行这些模板之前，请确保你的解决方案能成功生成，以便模板可使用项目输出的 DLL 文件 (*MovieTutorial.Web.dll*) 生成代码。

在生成解决方案之后，点击 生成 菜单，然后选择 转换所有模板。

现在，我们可以使用智能提示的输入方式来替代硬编码字段名称，并在编译时检查版本：

```
namespace MovieTutorial.MovieDB
{
    //...
    public class MovieGrid extends EntityGrid<MovieRow, any>
    {
        constructor(container: JQuery) {
            super(container);
        }

        protected getQuickSearchFields(): Serenity.QuickSearchField[]
        {
            let fld = MovieRow.Fields;
            return [
                { name: "", title: "all" },
                { name: fld.Description, title: "description" },
                { name: fld.Storyline, title: "storyline" },
                { name: fld.Year, title: "year" }
            ];
        }
    }
    //...
}
```

字段标题呢？作为字段名称，它并不那么重要，但可用于本地化（如果我们以后决定翻译它）：

```

namespace MovieTutorial.MovieDB
{
    //...
    public class MovieGrid extends EntityGrid<MovieRow, any>
    {
        constructor(container: JQuery) {
            super(container);
        }

        protected getQuickSearchFields(): Serenity.QuickSearchField[] {
            let fld = MovieRow.Fields;
            let txt = (s) => Q.text("Db." +
                MovieRow.localTextPrefix + "." + s).toLowerCase();
            return [
                { name: "", title: "all" },
                { name: fld.Description, title: txt(fld.Description) },
                { name: fld.Storyline, title: txt(fld.Storyline) },
                { name: fld.Year, title: txt(fld.Year) }
            ];
        }
    }
    //...
}

```

我们使用了在客户端可用的本地文本字典（翻译）。如：

```

{
    // ...
    "Db.MovieDB.Movie.Description": "Description",
    "Db.MovieDB.Movie.Storyline": "Storyline",
    "Db.MovieDB.Movie.Year": "Year"
    // ...
}

```

行字段的本地文本唯一键由 `"Db." + (LocalTextPrefix for Row) + "." + FieldName` 组成。

它们的值从字段的 `[DisplayName]` 特性生成，但也可能是另一种语言的翻译。

`LocalTextPrefix` 默认情况下相当于 `ModuleName + "." + RowClassName`，但是可以在 `Row` 构造函数中修改。

## 添加一个影片类型字段

如果我们还想在影片表中保存电视剧和微电影，那我们就需要另一个字段（MovieKind）来存储它。

我们在创建影片（Movie）表时并没有添加 MovieKind 字段，现在我们用另一个迁移类把它添加到数据库中。

不要修改现有的迁移类，因为现有的迁移类不会再次运行。

在 *Modules/Common/Migrations/DefaultDB/DefaultDB\_20160519\_145500\_MovieKind.cs* 下面创建另一个迁移文件：

```
using FluentMigrator;

namespace MovieTutorial.Migrations.DefaultDB
{
    [Migration(20160519145500)]
    public class DefaultDB_20160519_145500_MovieKind : Migration
    {
        public override void Up()
        {
            Alter.Table("Movie").InSchema("mov")
                .AddColumn("Kind").AsInt32().NotNullable()
                .WithDefaultValue(1);
        }

        public override void Down()
        {
        }
    }
}
```

定义一个影片类型枚举

现在我们把 *Kind* 字段添加到影片（Movie）表中，我们需要一些影片类型作为值。让我们将它定义在 *MovieTutorial.Web/Modules/MovieDB/Movie/MovieKind.cs* 的枚举中：

```
using Serenity.ComponentModel;
using System.ComponentModel;

namespace MovieTutorial.MovieDB
{
    [EnumKey("MovieDB.MovieKind")]
    public enum MovieKind
    {
        [Description("Film")]
        Film = 1,
        [Description("TV Series")]
        TvSeries = 2,
        [Description("Mini Series")]
        MiniSeries = 3
    }
}
```

## 在 **MovieRow** 实体中添加 类型（**Kind**）字段

因为我们没有使用 Sergen，所以我们需要在 *MovieRow.cs* 手工添加一个映射类型（*Kind*）属性。在 *MovieRow.cs* 的 *Runtime* 属性下面添加：

```
[DisplayName("Runtime (mins)")]
public Int32? Runtime
{
    get { return Fields.Runtime[this]; }
    set { Fields.Runtime[this] = value; }
}

[DisplayName("Kind"), NotNull]
public MovieKind? Kind
{
    get { return (MovieKind?)Fields.Kind[this]; }
    set { Fields.Kind[this] = (Int32?)value; }
}
```

我们还需要声明一个 `Int32Field` 对象，该对象用于 Serenity 实体系统。在 `MovieRow.cs` 的底部找到 `RowFields` 类并在 `Runtime` 字段后添加 `Kind` 字段：

```
public class RowFields : RowFieldsBase
{
    // ...
    public readonly Int32Field Runtime;
    public readonly Int32Field Kind;

    public RowFields()
        : base("[mov].Movie")
    {
        LocalTextPrefix = "MovieDB.Movie";
    }
}
```

## 为影片表单添加影片类型

如果我们现在生成并运行项目，尽管我们已经在 `MovieRow` 添加了 `Kind` 字段的映射，但在影片表单中还是看不到任何的变化。这是由于表单中字段的显示/编辑是在 `MovieForm.cs` 中控制的。

于是我们对 `MovieForm.cs` 做如下修改：

```
namespace MovieTutorial.MovieDB.Forms
{
    // ...
    [FormScript("MovieDB.Movie")]
    [BasedOnRow(typeof(Entities.MovieRow))]
    public class MovieForm
    {
        // ...
        public MovieKind Kind { get; set; }
        public Int32 Runtime { get; set; }
    }
}
```

现在，生成并运行应用程序。当你尝试编辑或添加影片时，什么也不会发生。这是一个预料之中的情况。如果你打开浏览器的开发者工具（F12），在控制台中将出现一个错误：

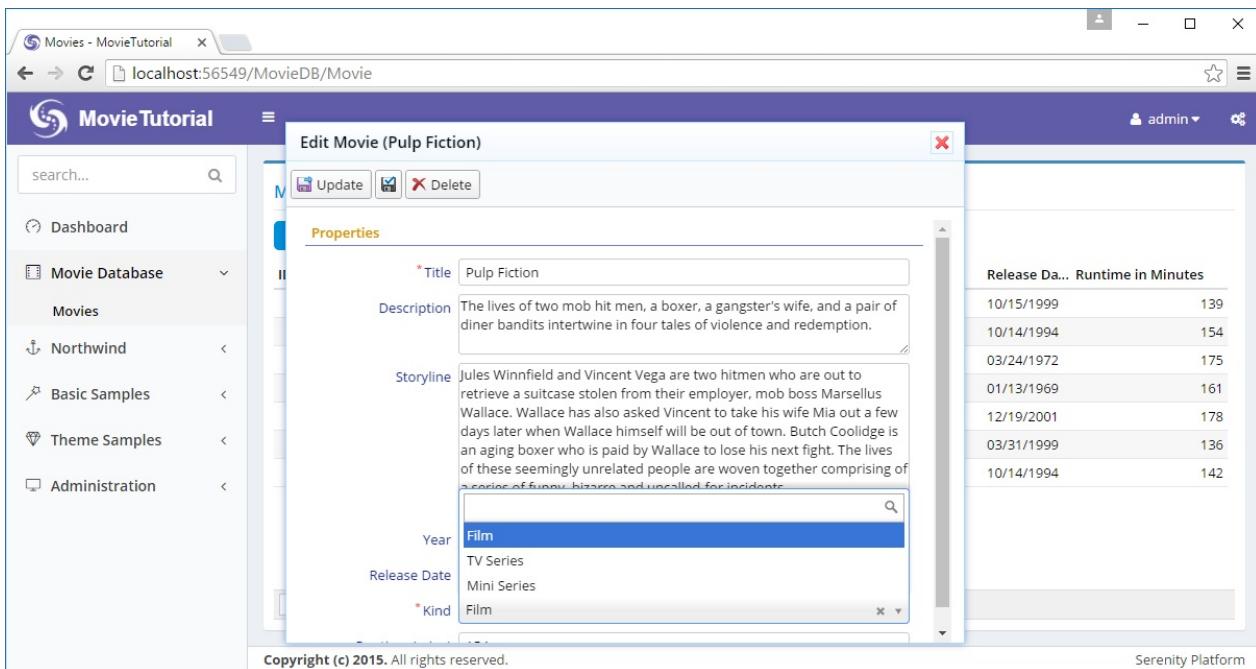
```
Uncaught Can't find MovieTutorial.MovieDB.MovieKind enum type!
```

每当发生这样的事，请先检查浏览器控制台报告的错误信息。

这个错误是由于 `MoveKind` 枚举在客户端无效而导致的，因此，在这种情况下，我们应该在运行程序之前运行我们的 T4 模板。

现在在 Visual Studio 中再次点击 生成 -> 转换所有模板。

重新生成解决方案并运行它。现在就可以在表单中使用下拉列表选择影片类型了。



## 为影片类型定义一个默认值

*Kind* 是一个必填字段，我们必须在新增 *Movie* 对话框中填写该字段，否则将得到一个验证错误。

但是我们大部分影片都是故事片，所以应该把该值设为默认值。

若要为 *Kind* 属性添加默认值，需要像这样添加 *DefaultValue* 特性：

```
[DisplayName("Kind"), NotNull, DefaultValue(MovieKind.Film)]
public MovieKind? Kind
{
    get { return (MovieKind?)Fields.Kind[this]; }
    set { Fields.Kind[this] = (Int32?)value; }
}
```

现在，在新增 *Movie* 对话框中，*Kind* 字段的值将预设为 *Film*。

# 添加影片流派 (Movie Genres)

## 添加流派 (Genre) 字段

我们需要一个检索表来保存影片流派 (Movie genres)。在影片类型中我们使用一个枚举，但这次流派可能不是静态的，不能再把它定义成枚举。

像往常一样，我们从迁移类开始：

*Modules/Common/Migrations/DefaultDB/  
DefaultDB\_20160519\_154700\_GenreTable.cs:*

```

using FluentMigrator;
using System;

namespace MovieTutorial.Migrations.DefaultDB
{
    [Migration(20160519154700)]
    public class DefaultDB_20160519_154700_GenreTable : Migration
    {
        public override void Up()
        {
            Create.Table("Genre").InSchema("mov")
                .WithColumn("GenreId").AsInt32().NotNullable()
                .PrimaryKey().Identity()
                .WithColumn("Name").AsString(100).NotNullable();

            Alter.Table("Movie").InSchema("mov")
                .AddColumn("GenreId").AsInt32().Nullable()
                .ForeignKey("FK_Movie_GenreId", "mov", "Genre",
                           "GenreId");
        }

        public override void Down()
        {
        }
    }
}

```

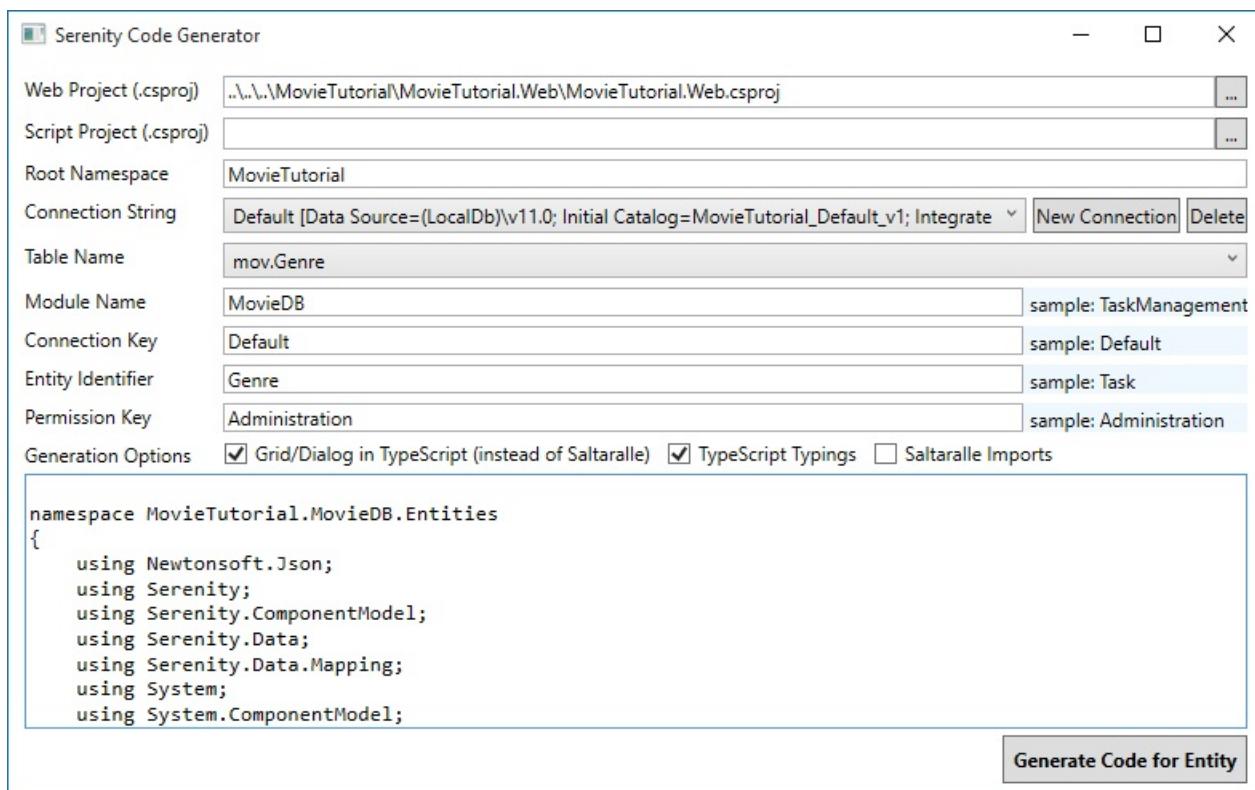
我们也要在电影 (*Movie*) 表中添加 *GenreId* 字段。

实际上，一部电影可以属于多个流派，因此我们应该把它保存在一张单独的影片流派 (*MovieGenres*) 表中。但是现在，我们假设一部电影只属于一个流派。我们将在后面演示如何将它改为可属于多个流派。

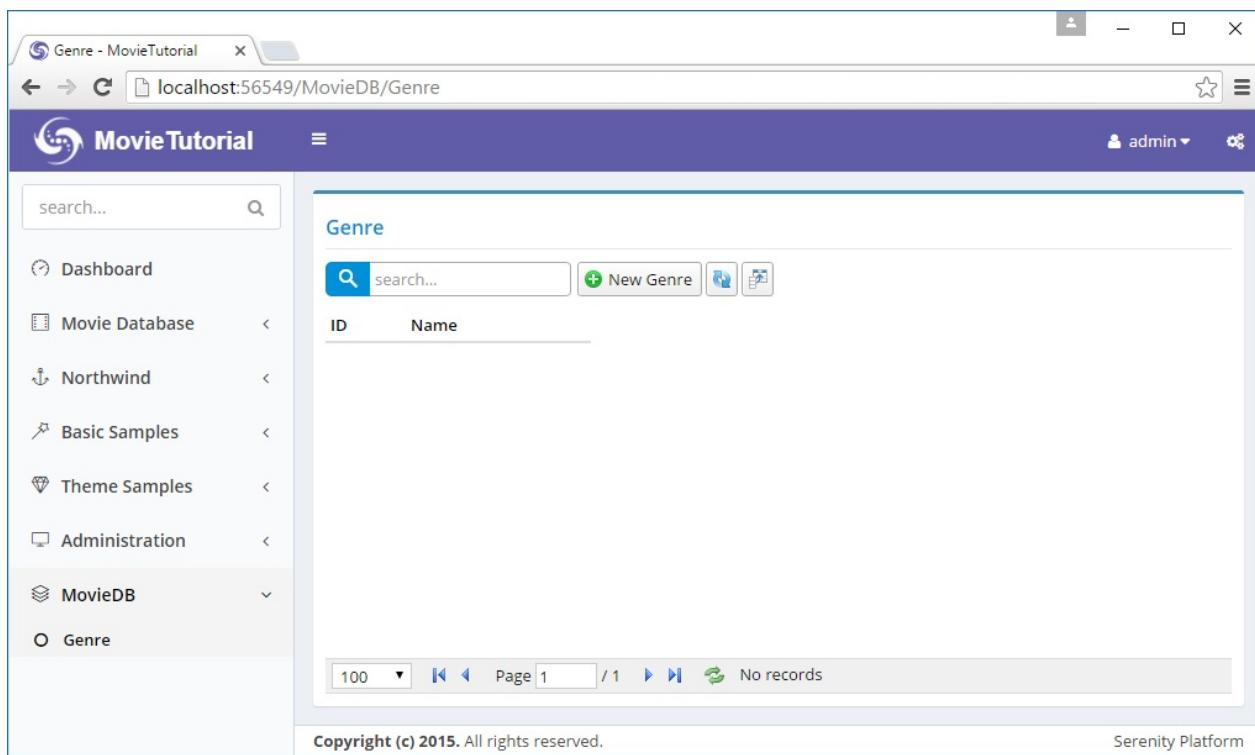
## 为影片流派表生成代码

再次使用程序包管理器控制台打开 `sergen.exe`，并为 *Genre* 表设置生成代码的参数，如下所示：

## 添加影片流派 (Movie Genres)



重新生成解决方案并运行项目，我们将得到这样的新页面：



正如你在截图中看到，它在左侧导航菜单中重新生成 **MovieDB** 菜单，而不是生成到我们最近重命名的 **Movie Database** 菜单下。

这是由于 **Sergen** 不知道我们对 **Movie** 页面的定制化修改，所以我们需要手工把它移到 **Movie Database** 下面。

打开 *Modules/Movie/GenrePage.cs*，剪切下面的导航连接：

```
[assembly: Serenity.Navigation.NavigationLink(int.MaxValue, "MovieDB/Genre",
    typeof(MovieTutorial.MovieDB.Pages.GenreController))]
```

、

并把它移到 *Modules/Common/Navigation/NavigationItems.cs*：

```
//...
[assembly: NavigationMenu(2000, "Movie Database", icon: "icon-film")]
[assembly: NavLink(2100, "Movie Database/Movies",
    typeof(MovieDB.MovieController), icon: "icon-camcorder")]
[assembly: NavLink(2200, "Movie Database/Genres",
    typeof(MovieDB.GenreController), icon: "icon-pin")]
//...
```

## 添加多个流派的定义

现在，让我们添加一些流派示例。我将通过迁移类来完成，而不是在不同计算机中重复该操作，但是你可以通过影片流派页面手工添加。

```
using FluentMigrator;
using System;

namespace MovieTutorial.Migrations.DefaultDB
{
    [Migration(20160519181800)]
    public class DefaultDB_20160519_181800_SampleGenres : Migration
    {
        public override void Up()
        {
            Insert.IntoTable("Genre").InSchema("mov")
                .Row(new
                {
```

```

        Name = "Action"
    })
    .Row(new
{
    Name = "Drama"
})
.Row(new
{
    Name = "Comedy"
})
.Row(new
{
    Name = "Sci-fi"
})
.Row(new
{
    Name = "Fantasy"
})
.Row(new
{
    Name = "Documentary"
});
}

public override void Down()
{
}
}
}

```

## 在 MovieRow 映射 **GenreId** 字段

像我们之前在添加 *Kind* 字段一样，*GenreId* 字段也需要在 *MovieRow.cs* 中映射。

```

namespace MovieTutorial.MovieDB.Entities
{
    // ...
    public sealed class MovieRow : Row, IIdRow, INameRow
    {

```

```

    [DisplayName("Kind"), NotNull, DefaultValue(1)]
    public MovieKind? Kind
    {
        get { return (MovieKind?)Fields.Kind[this]; }
        set { Fields.Kind[this] = (Int32?)value; }
    }

    [DisplayName("Genre"), ForeignKey("[mov].Genre", "GenreId"),
     LeftJoin("g")]
    public Int32? GenreId
    {
        get { return Fields.GenreId[this]; }
        set { Fields.GenreId[this] = value; }
    }

    [DisplayName("Genre"), Expression("g.Name")]
    public String GenreName
    {
        get { return Fields.GenreName[this]; }
        set { Fields.GenreName[this] = value; }
    }

    // ...

    public class RowFields : RowFieldsBase
    {
        // ...
        public readonly Int32Field Kind;
        public readonly Int32Field GenreId;
        public readonly StringField GenreName;

        public RowFields()
            : base("[mov].Movie")
        {
            LocalTextPrefix = "MovieDB.Movie";
        }
    }
}

```

在这里，我们映射 `GenreId` 字段并使用 `ForeignKey` 特性定义它与 `[mov].Genre` 表中的 `GenreId` 有外键关系。

如果我们在添加流派表之后为电影表生成代码，Serenge 通过在数据库中检查外键定义来理解这种关系，并为我们生成类似的代码。

我们还添加了另一个字段，实际上，`GenreName` 并不是电影表 (`Movie`) 中的字段，而是流派 (`Genre`) 表中字段的。

Serenity 实体更像是 SQL 视图。你可以使用关联 (`joins`) 获取其他表的字段。

可在 `MovieId` 属性中添加 `LeftJoin("g")` 特性。每当流派 (`Genre`) 表需要被关联时，我们定义它的别名为 `g`。

因此，当 Serenity 需要从电影 (`Movies`) 表查询时，它生成这样的 SQL 查询：

```
SELECT t0.MovieId, t0.Kind, t0.GenreId, g.Name as GenreName
FROM Movies t0
LEFT JOIN Genre g on t0.GenreId = g.GenreId
```

该关联 (`join`) 操作只在需要流派 (`Genre`) 表字段时才执行，如在网格列表需要显示流派信息时。

通过在 `GenreName` 属性上面添加 `Expression("g.Name")`，指定该字段有一个 `g.Name` 的 SQL 表达式，表明这是一个来自 `g` 的关联字段。

### 为影片窗体添加流派选择

让我们把 `GenreId` 字段添加到 `MovieForm.cs`：

```
namespace MovieTutorial.MovieDB.Forms
{
    //...
    [FormScript("MovieDB.Movie")]
    [BasedOnRow(typeof(Entities.MovieRow))]
    public class MovieForm
    {
        //...
        public Int32 GenreId { get; set; }
        public MovieKind Kind { get; set; }
    }
}
```

现在，如果我们生成并运行应用程序，我们将在表单中看到一个 **Genre** 字段。现在的问题是 **Genre** 允许输入整型数值，而我们想使用下拉列表。

很显然，我们需要为 **GenreId** 字段修改编辑器类型。

## 为流派 (**Genres**) 声明一个检索脚本 (**Lookup Script**)

若要为流派 (**Genre**) 字段显示一个编辑器，在客户端列出数据库中的流派列表应该是可行的做法。

如果是枚举值，我们只需简单地运行 T4 模板就可以把枚举定义复制到客户端脚本。

但我们不能在这里用同样的方法。流派列表是基于数据库的动态列表。

Serenity 有 动态脚本 (*Dynamic scripts*) 的概念，将动态数据在运行时以生成脚本的形式提供给脚本端。

动态脚本类似于 Web 服务，但是它们产出的是可以在客户端缓存的动态 javascript 脚本。

这里的 动态 对应于它们包含的数据，而不是它们的行为。不像 Web 服务，动态脚本不能接受任何参数，并且网站的所有用户都共享动态脚本产生的数据，动态脚本更像是单例或静态变量。

你不应该试图写像 Web 服务行为(例如，检索)那样的动态脚本。

为流派 (Genre) 表声明一个动态检索脚本 (dynamic lookup script) ，打开 **GenreRow.cs** 并作做下修改：

```
namespace MovieTutorial.MovieDB.Entities
{
    // ...

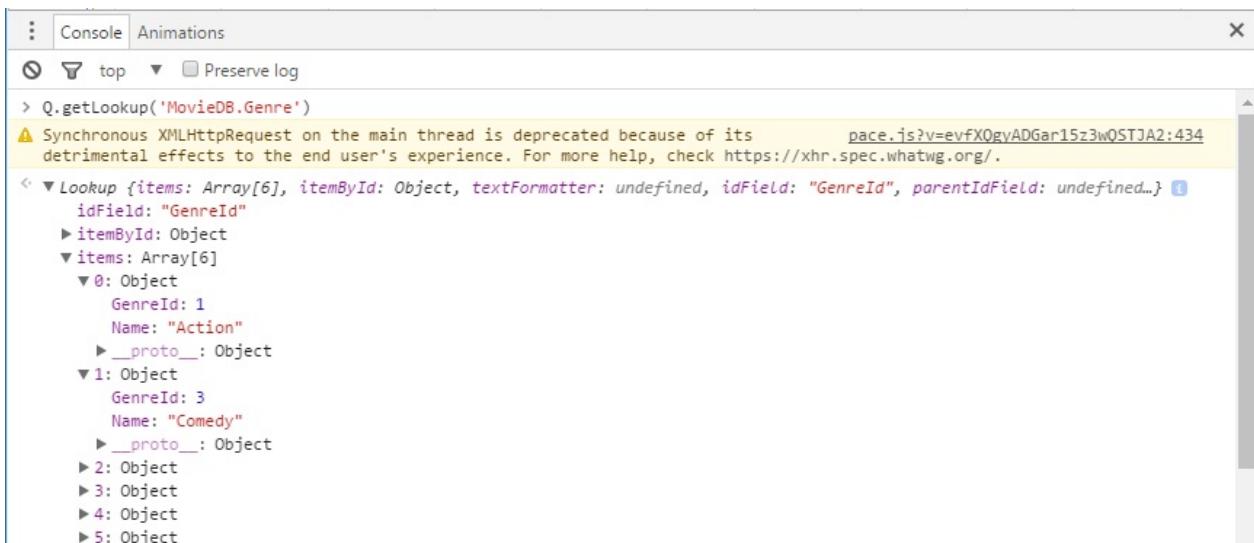
    [ConnectionKey("Default"), DisplayName("Genre"), InstanceName("Genre"),
     TwoLevelCached]
    [ReadPermission("Administration")]
    [ModifyPermission("Administration")]
    [JsonConverter(typeof(JsonRowConverter))]
    [LookupScript("MovieDB.Genre")]
    public sealed class GenreRow : Row, IIdRow, INameRow
    {
        // ...
    }
}
```

我们只添加了一行代码：`[LookupScript("MovieDB.Genre")]`。

重新生成并启动项目，在登录系统之后，使用 **F12** 打开开发者工具的控制台标签：

输入 `Q.getLookup('MovieDB.Genre')`：

你会得到一些像这样的信息：



这里，我们在声明 `LookupScript` 特性时为其指定 `MovieDB.Genre` 参数：

```
[LookupScript("MovieDB.Genre")]
```

这一步骤只是为了演示如何检查一个检索脚本在客户端是否可用。

"`MovieDB.Genre`" 的检索键 (Lookup key) 是区分大小写的，所以请确保你输入大小写一致的参数。

## 对流派表字段使用 `LookupEditor`

这里有两个地方为 `GenreId` 字段设置编辑器类型：一是在 `MovieForm.cs`，另一个是 `MovieRow.cs`。

我更喜欢后者，因为它是核心组件类。但如果编辑器类型仅应用于表单，你可以选择在表单类上设置它。

定义在表单中的信息不能被重用。例如，网格列表使用 `XYZColumn.cs` / `XYZRow.cs`，而对话框使用 `XYZForm.cs` / `XYZRow.cs`。由此可见，把可重用信息定义在 `XYZRow.cs` 更好。

打开 `MovieRow.cs` 并为 `GenreId` 属性添加 `LookupEditor` 特性：

```
[DisplayName("Genre"), ForeignKey("[mov].Genre", "GenreId"),
LeftJoin("g")]
[LookupEditor("MovieDB.Genre")]
public Int32? GenreId
{
    get { return Fields.GenreId[this]; }
    set { Fields.GenreId[this] = value; }
}
```

在我们生成和启动项目之后，在 Genre 字段中，我们现在有一个可搜索的（Select2.js）下拉列表。

ID	Title	Description	Storyline	Year	Release Da...	Runtime in Minutes
8	Fight Club					139
6	Pulp Fiction					154
5	The Godfather					175
7	The Good, the Bad and the Ugly					161
3	The Lord of the Rings: The Return of the King					178
2	The Matrix					136
4	The Shawshank Redemption					142

在定义 [LookupEditor] 时，我们硬编码检索键。我们还可以使用 GenreRow 进行信息重用：

```
[DisplayName("Genre"), ForeignKey("[mov].Genre", "GenreId"),
LeftJoin("g")]
[LookupEditor(typeof(GenreRow))]
public Int32? GenreId
{
    get { return Fields.GenreId[this]; }
    set { Fields.GenreId[this] = value; }
}
```

它们在功能上是等效的，但我更喜欢后者。后面这种方式，Serenity 将在 `GenreRow` 中找到 `[LookupScript]` 特性，并从中获取检索键信息。如果我们的 `GenreRow` 没有 `[LookupScript]` 特性，在应用程序启动时，你会得到错误：

```
Server Error in '/' Application.

'MovieTutorial.MovieDB.Entities.GenreRow' type doesn't have a
[LookupScript] attribute, so it can't be used with a LookupEditor
r!

Parameter name: lookupType
```

在应用程序启动时扫描表单，因此没有修复该问题时，是没有办法处理这个错误的。

## 在影片表格中显示流派

目前，影片流派可以在表单中编辑，但没有显示在影片列表。我们可以通过编辑 `MovieColumns.cs` 来显示 `GenreName` (而不是 `GenreId`) 。

```

namespace MovieTutorial.MovieDB.Columns
{
    // ...
    public class MovieColumns
    {
        //...
        [Width(100)]
        public String GenreName { get; set; }
        [DisplayName("Runtime in Minutes"), Width(150), AlignRight]
        public Int32 Runtime { get; set; }
    }
}

```

现在 `GenreName` 可以在网格列表中显示了。

ID	Title	Description	Storyline	Year	Release Da...	Genre	Runtime in Minutes
8	Fight Club	An insomniac office w...	A ticking-time-bomb insomniac and a s...	1999	10/15/1999	Action	139
6	Pulp Fiction	The lives of two mob ...	Jules Winnfield and Vincent Vega are t...	1994	10/14/1994	Action	154
5	The Godfather	The aging patriarch of...	When the aging head of a famous crim...	1972	03/24/1972	Drama	175
7	The Good, the Bad an...	A bounty hunting sca...	Blondie (The Good) is a professional g...	1969	01/13/1969	Drama	161
3	The Lord of the Rings:...	A meek hobbit of the ...	An ancient Ring thought lost for centu...	2001	12/19/2001	Fantasy	178
2	The Matrix	A computer hacker le...	Thomas A. Anderson is a man living tw...	1999	03/31/1999	Sci-fi	136
4	The Shawshank Redem...	Two imprisoned men ...	Andy Dufresne is a young and success...	1994	10/14/1994	Drama	142

## 让就地定义新的流派成为可能

当我们为示例影片设置流派时，我们注意到影片《黄金三镖客》（*The Good, the Bad and the Ugly*）是属于西部片（Western），但在流派下拉列表中还没有该选项（所以我不得不把流派选为剧情）。

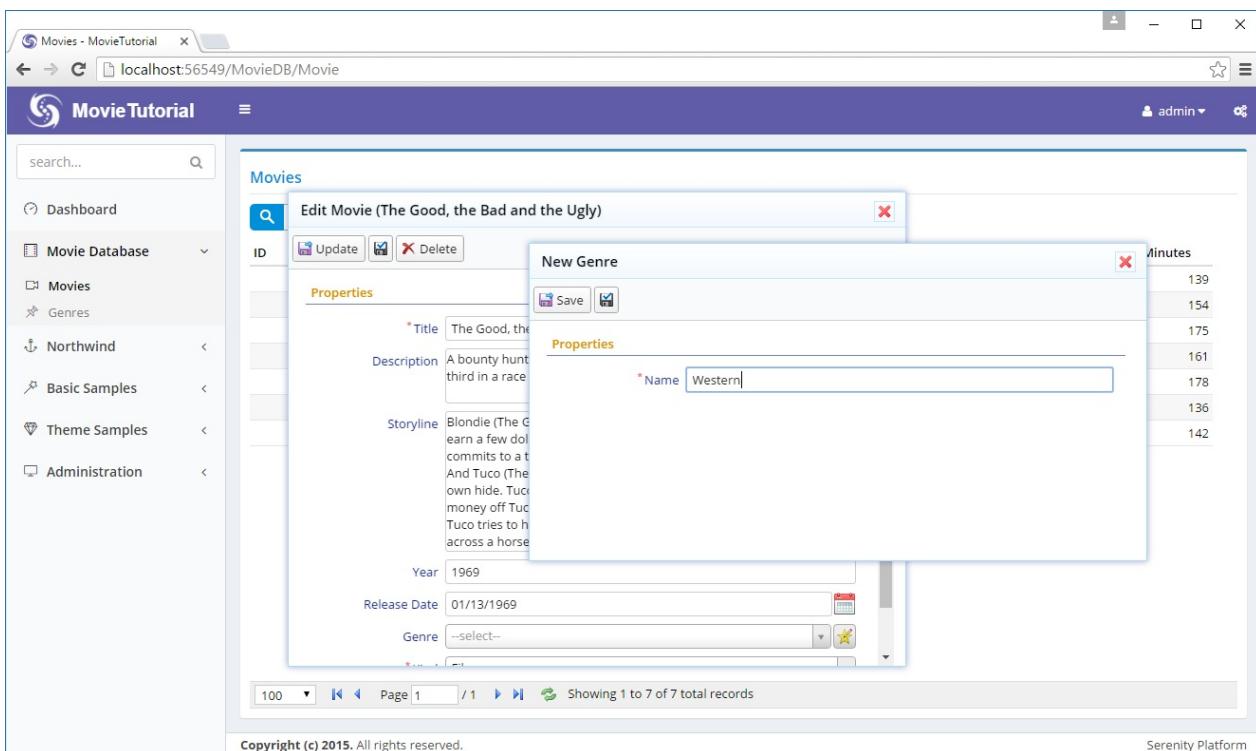
要为该影片添加正确的流派，其中一个做法是打开流派页面，添加该当流派，并再次回到影片表单……然而这并不是很好的操作体验。

幸运的是，Serenity 集成了就地为编辑器声明新内容的能力。

打开 MovieRow.cs，并对 *LookupEditor* 特性做如下修改：

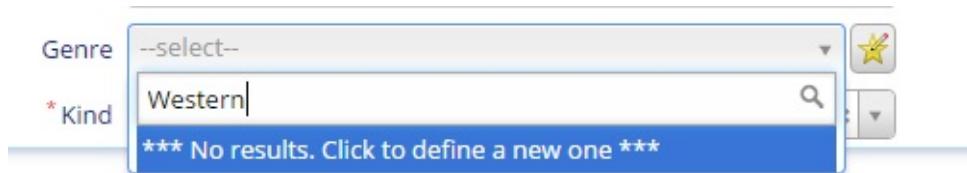
```
[DisplayName("Genre"), ForeignKey("[mov].Genre", "GenreId"), LeftJoin("g")]
[LookupEditor(typeof(GenreRow), InplaceAdd = true)]
public Int32? GenreId
{
    get { return Fields.GenreId[this]; }
    set { Fields.GenreId[this] = value; }
}
```

现在，我们可以点击流派 (Genre) 字段旁边的图标添加新的流派。



在这里，我们可看到，可以在电影页面打开一个对话框，并在对话框中打开另一个页面 (GenreDialog)。在 Serenity 应用程序中，所有的客户端对象（对话框、网格列表、编辑器、格式化器等）都是自包含的可重用组件（部件），它们并没有绑定到任何页面。

还可以在流派编辑器中输入内容，它将为你提供一个选项来添加一个新的流派。



## 它是如何确定要使用的对话框类型

你可能还没有注意到这个细节：在我们想就地添加一个新的流派时，我们的流派检索编辑器（lookup editor）为选择流派而自动打开一个新的 流派对话框（*GenreDialog*）。

在这里，我们的检索编辑器使用了一项约定。因为它的检索键是 *MovieDB.Genre*，所以它使用下面的完整名称搜索对话框类：

```
MovieDB.GenreDialog
MovieTutorial.MovieDB.GenreDialog
...
...
```

幸运的是，我们有一个流派对话框（*GenreDialog*），它定义在 *Modules/Genre/GenreDialog.ts*，并且它的完整名称是 *MovieTutorial.MovieDB.GenreDialog*。

```

namespace MovieTutorial.MovieDB {

    @Serenity.Decorators.registerClass()
    @Serenity.Decorators.responsive()
    export class GenreDialog extends Serenity.EntityDialog<Genre
Row, any> {
        protected getKey() { return GenreForm.formKey; }
        protected getIdProperty() { return GenreRow.idProperty;
    }
        protected getLocalTextPrefix() { return GenreRow.localTe
xtPrefix; }
        protected getNameProperty() { return GenreRow.nameProper
ty; }
        protected getService() { return GenreService.baseUrl; }

        protected form = new GenreForm(this.idPrefix);
    }
}

```

如果 `GenreRow` 的检索键和其对话框类不匹配，当我们单击就地添加按钮时，我们会在浏览器控制台得到一个错误：

```
Uncaught MovieDB.GenreDialog dialog class is not found!
```

在名称不匹配的情况下，你可使用一个兼容的检索键，如 "`ModuleName.RowType`"，或者显式指定对话框类型：

```

[DisplayName("Genre"), ForeignKey("[mov].Genre", "GenreId"), Left
Join("g")]
[LookupEditor(typeof(GenreRow), InplaceAdd = true, DialogType =
"MovieDB.Genre")]
public Int32? GenreId
{
    get { return Fields.GenreId[this]; }
    set { Fields.GenreId[this] = value; }
}

```

你不应该把 *Dialog* 作为对话框的后缀，也不能有完整的命名空间，如 *MovieTutorial.MovieDB.Genre*，因为 Serenity 会自动搜索它们。

## 为网格列表添加流派的快速过滤

随着我们影片列表数据变得越来越多，除了快速搜索功能，我们可能还需要基于某字段值来筛选影片。

Serenity 有几个过滤方法，快速过滤器 (Quick Filter) 就是其中之一。我们将其应用到流派 (Genre) 字段。

编辑 *Modules/MovieDB/Movie/MovieColumns.cs* 文件，在 *GenreName* 属性上面添加一个 [QuickFilter] 特性：

```
public class MovieColumns
{
    //...
    public DateTime ReleaseDate { get; set; }
    [Width(100), QuickFilter]
    public String GenreName { get; set; }
    [DisplayName("Runtime in Minutes"), Width(150), AlignRight]
    public Int32 Runtime { get; set; }
}
```

生成并导航到影片页面。你可以通过流派 (Genre) 字段的下拉列表快速过滤：

The screenshot shows the 'Movies' list page in the MovieTutorial application. On the left, there's a sidebar with navigation links like 'Dashboard', 'Movie Database', 'Movies', 'Genres', etc. The main area has a search bar and a table with movie data. A dropdown menu for 'Genre' is open, showing options like Action, Comedy, Documentary, Drama, Fantasy, and Sci-fi. 'Comedy' is currently selected. The table below lists several movies with their details.

	Description	Storyline	Year	Release Da...	Gen	
1	An insomniac office w...	A ticking-time-bomb insomniac and a s...	1999	10/15/1999	Action	
2	The lives of two mob ...	Jules Winnfield and Vincent Vega are t...	1994	10/14/1994	Action	
3	The aging patriarch of...	When the aging head of a famous crim...	1972	03/24/1972	Drama	
4	Bad a... A bounty hunting sca...	Blondie (The Good) is a professional g...	1969	01/13/1969	Drama	
5	Rings;... A meek hobbit of the ...	An ancient Ring thought lost for centu...	2001	12/19/2001	Fant	
6	2 The Matrix	A computer hacker le...	Thomas A. Anderson is a man living tw...	1999	03/31/1999	Sci-f
7	4 The Shawshank Red...	Two imprisoned men ...	Andy Dufresne is a young and success...	1994	10/14/1994	Drama

被过滤的字段实际上是 `GenreId`，而不是我们附加 QuickFilter 特性的 `GenreName`。Serenity 足够聪明能理解这种关系，并通过查看 `GenreRow.cs` 的 `GenreId` 属性确定使用编辑器的类型。

## 重新运行 T4 模板

由于我们在应用程序中添加了一个新实体，我们应该在生成解决方案之后运行 T4 模板。

## 更新 Serenity 程序包

当我开始写这个教程时，Serenity（NuGet 程序包中含 Serenity 程序集和标准的脚本库）和 Serene（应用程序模板）是 2.1.8 版本。

当你阅读本教程时，你可能在使用更高的版本，所以此时你还不需要更新 serenity。

但是，我想向你演示如何更新 Serenity 的 NuGet 程序包，以便你在将来能顺利更新版本。

比起使用 Nuget 程序包管理器，我更喜欢使用程序包管理控制台，因为命令行操作更快。

所以，选择 视图 -> 其他窗体 -> 程序包管理控制台。

输入：

```
Update-Package Serenity.Web
```

由于依赖引用，也将更新 *MovieTutorial.Web* 项目的下列 NuGet 程序包：

```
Serenity.Core  
Serenity.Data  
Serenity.Data.Entity  
Serenity.Services
```

要更新 Serenity.CodeGenerator（含 sergen.exe），请输入：

```
Update-Package Serenity.CodeGenerator
```

Serenity.CodeGenerator 也安装在 *MovieTutorial.Web* 项目中。

在更新期间，如果 NuGet 提示要覆盖修改一些脚本文件，你可以放心地同意，除非你想手工修改 Serenity 的脚本文件(我建议你避免这样做)。

现在重新生成你的解决方案，它应该能生成成功。

Serenity 可能会不时地发生重大改变，但尽量把变化保持在最低限度，尽管如此，你可能还是需要在应用程序代码中进行几处手工的修改。

这些更改都被记录在更新日志的 [BREAKING CHANGE] 标签中：

<https://github.com/volkanceylan/Serenity/blob/master/CHANGELOG.md>

在更新后，如果仍有问题，欢迎使用 issue 反馈给我们：

<https://github.com/volkanceylan/Serenity/issues>

## 哪些内容会更新

更新 Serenity 的 NuGet 程序包以获取最新版本的 Serenity 程序集。

也可以更新一些其他的第三方程序包，例如：ASP.NET MVC、FluentMigrator、Select2.js、SlickGrid 等。

请不要更新 Select2.js 3.5.1 之后的版本，因为它有一些 jQuery 验证的兼容问题。

Serenity.Web 程序包还提供一些静态脚本和 css 资源，如：

```
Content/serenity/serenity.css  
Scripts/saltarelle/mscorlib.js  
Scripts/saltarelle/linq.js  
Scripts/serenity/Serenity.CoreLib.js  
Scripts/serenity/Serenity.Script.UI.js
```

因此，这些或更多的资源都将在 MovieApplication.Web 中更新。

## 哪些内容不会更新（或不能自动更新）

更新 Serenity 程序包，是更新了 Serenity 程序集和大部分静态脚本，但并不是所有的 Serene 模板内容都会被更新。

我们努力让更新程序变得尽可能地简单，但 Serene 只是一个项目模板，而不是一个静态的程序包，你的应用程序是一个可定制的 Serene 副本。

你可能修改了应用程序的源代码，所以更新一个有旧版本 Serene 模板的应用程序并不像听起来那么简单。

所以有时你可能需要创建一个含最新 **Serene** 模板的新应用程序，用它和你的应用程序进行比较，并手工合并你需要的功能。

通常情况下，更新 **Serenity** 程序包就足够了，**Serene** 本身是不需要更新的，除非你需要最新版本 **Serene** 的功能。

我们有把 **Serene** 模板也集成到一个 NuGet 程序包的计划，但是我们不知道在更新应用程序时如何处理你的更改，例如，公共代码的导航菜单项，如果你删除了 **Northwind** 相关导航代码，但是在更新时，我们是要重新加上它吗？欢迎你给我们提出宝贵建议.....

## 允许选择多个流派

需求发生了变更，我们现在想让影片允许选择多个流派。

为此，我们需要一个 m-n 映射表，它将让我们的电影关联到多个流派。

### 创建影片流派 (**MovieGenres**) 表

和往常一样，我们从迁移类开始：

**Modules/Common/Migrations/DefaultDB/  
DefaultDB\_20160528\_115400\_MovieGenres.cs:**

```

using FluentMigrator;

namespace MovieTutorial.Migrations.DefaultDB
{
    [Migration(20160528115400)]
    public class DefaultDB_20160528_115400_MovieGenres : Migrati
on
    {
        public override void Up()
        {
            Create.Table("MovieGenres").InSchema("mov")
                .WithColumn("MovieGenreId").AsInt32()
                    .Identity().PrimaryKey().NotNullable()
                .WithColumn("MovieId").AsInt32().NotNullable()
                    .ForeignKey("FK_MovieGenres_MovieId",
                        "mov", "Movie", "MovieId")
                .WithColumn("GenreId").AsInt32().NotNullable()
                    .ForeignKey("FK_MovieGenres_GenreId",
                        "mov", "Genre", "GenreId");

            Execute.Sql(
                @"INSERT INTO mov.MovieGenres (MovieId, GenreId)
                    SELECT m.MovieId, m.GenreId
                    FROM mov.Movie m
                    WHERE m.GenreId IS NOT NULL");
        }

        Delete.ForeignKey("FK_Movie_GenreId")
            .OnTable("Movie").InSchema("mov");
        Delete.Column("GenreId")
            .FromTable("Movie").InSchema("mov");
    }

    public override void Down()
    {
    }
}

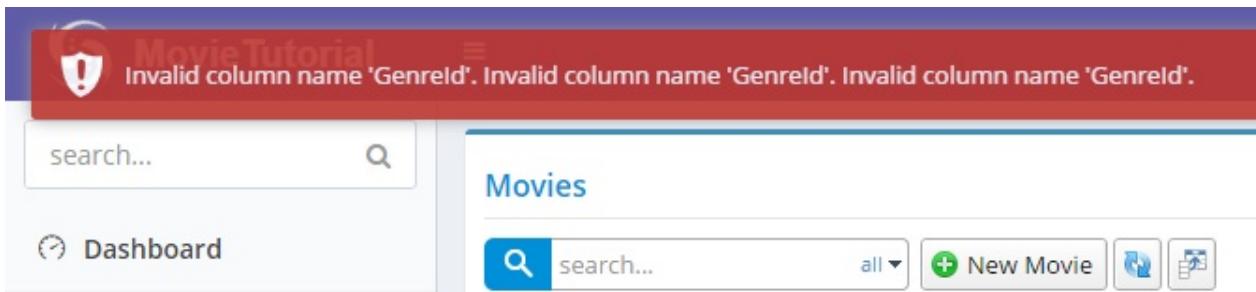
```

我尝试在影片（Movie）表保存保存现有的流派声明，把他们复制到新的影片流派（MovieGenres）表，上面的 `Execute.Sql` 语句就是做这事。

然后，我们应该删除 `GenreId` 字段，但首先应该删除先前定义的外键 `FK_Movie_GenreId`。

## 删除 `GenreId` 字段的映射

一旦你生成并打开影片页面，你会得到错误：



这是因为我们行（row）中还有 `GenreId` 字段的映射，上面的错误来自 AJAX 调用处理电影（Movie）表的列表服务。

重复的错误消息来源于 SQL server。生成的动态 SQL 多次使用列名 `MovieId`。

删除 `GenreId` 和 `GenreName` 属性以及 `MovieRow.cs` 的相关字段对象：

```
// remove this
public Int32? GenreId
{
    get { return Fields.GenreId[this]; }
    set { Fields.GenreId[this] = value; }
}

// remove this
public String GenreName
{
    get { return Fields.GenreName[this]; }
    set { Fields.GenreName[this] = value; }
}

public class RowFields : RowFieldsBase
{
    // and remove these
    public Int32Field GenreId;
    public StringField GenreName;
}
```

在 *MovieColumns.cs* 中删除 *GenreName* 属性：

```
// remove this
[Width(100), QuickFilter]
public String GenreName { get; set; }
```

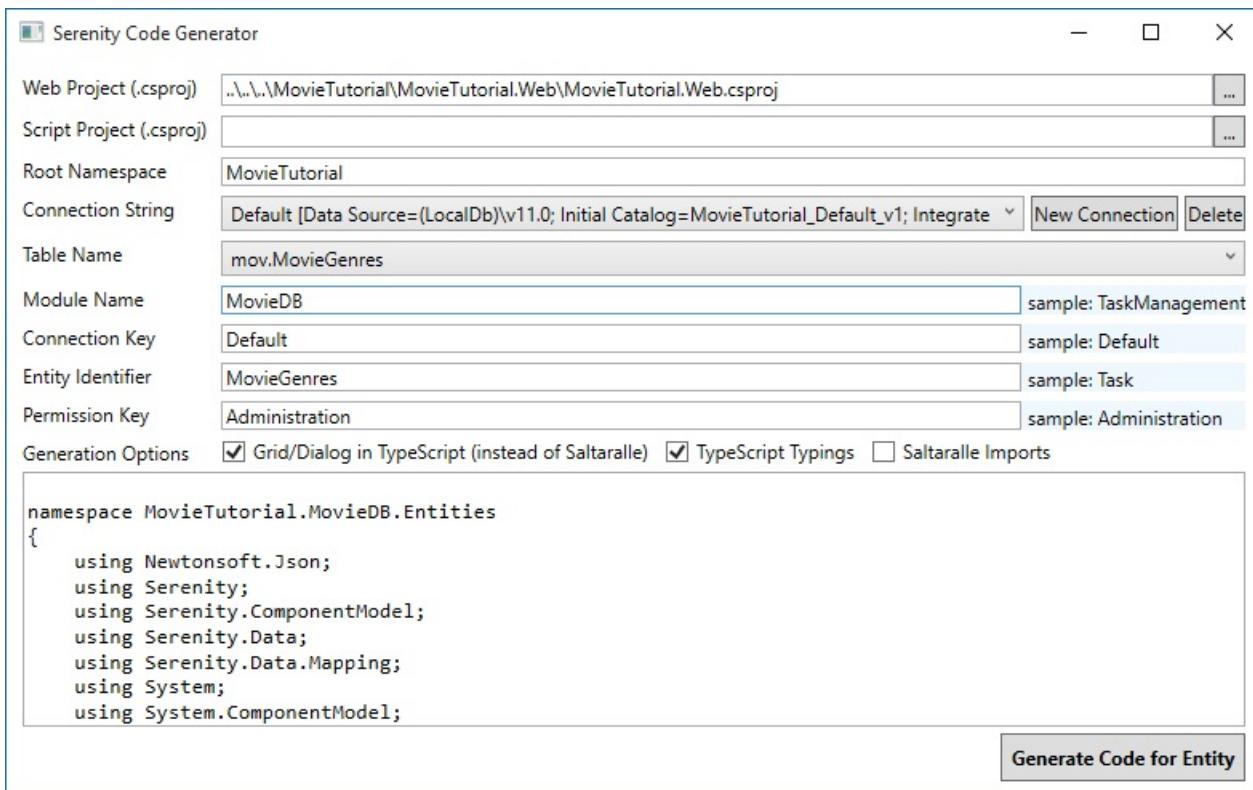
在 *MovieForm.cs* 中删除 *GenreId* 属性：

```
// remove this
public Int32 GenreId { get; set; }
```

生成项目之后，我们再次有一个可工作的影片页面。

## 为影片流派 (**MovieGenres**) 表生成代码

像往常一样，启动 *sergen* 并为影片流派 (*MovieGenres*) 表生成代码：



我们不打算在一个单独的页面编辑影片的流派，你可以放心地删除以下的生成文件：

```
MovieGenresColumns.cs
MovieGenresDialog.ts
MovieGenresEndpoint.cs
MovieGenresForm.cs
MovieGenresGrid.cs
MovieGenresIndex.cshtml
MovieGenresPage.cs
```

你也可以删除 `site.less` 的 `s-MovieDB-MovieGenresDialog` 样式。

最后只留下两个文件：`MovieGenresRow.cs` 和 `MovieGenresRepository.cs`。

生成项目之后，记得运行 T4 模板，若遗忘这个步骤，T4 将不会为我们生成与 `MovieGenresForm` 等相关的文件。

## 添加流派列表（GenreList）字段

由于现在一部电影可能有多个流派，所以不能使用一个 Int32 类型属性保存流派信息，我们需要一个 Int32 类型的列表，例如 `List<Int32>`。因此，我们在 `MovieRow.cs` 添加 `GenreList` 属性：

可能需要在文件添加 `System.Collections.Generic` 引用。

```
//...
[DisplayName("Kind"), NotNull, DefaultValue(MovieKind.Film)]
public MovieKind? Kind
{
    get { return (MovieKind?)Fields.Kind[this]; }
    set { Fields.Kind[this] = (Int32?)value; }
}

[DisplayName("Genres")]
[LookupEditor(typeof(GenreRow), Multiple = true), ClientSide]
[LinkingSetRelation(typeof(MovieGenresRow), "MovieId", "GenreId")]
public List<Int32> GenreList
{
    get { return Fields.GenreList[this]; }
    set { Fields.GenreList[this] = value; }
}

public class RowFields : RowFieldsBase
{
    //...
    public Int32Field Kind;
    public ListField<Int32> GenreList;
```

我们的属性与 `GenreId` 属性一样添加了 `[LookupEditor]` 特性，但是有个不一样的地方：这次允许可多选流派。所以，我们设置 `Multiple = true` 作为特性的参数。

`GenreList` 属性还有 `ClientSide` 标识，该标识在 Serenity 中类似 `Unmapped` 特性，表示该属性不是数据库中的字段。

我们在电影（Movie）表中并没有 *GenreList* 字段，所以我们应该把它设置为未映射（unmapped）字段。另外，Serenity 会尝试选择该字段，因此我们会得到一个 SQL 错误。

下面这行代码，我们使用另一个 *LinkingSetRelation* 新特性：

```
[LinkingSetRelation(typeof(MovieGenresRow), "MovieId", "GenreId")]
```

该特性表示此表的一行关联另一表的多行，说明这两个表是 M-N 关系。

第一个参数是 m-n 映射行的类型，在这里是 MovieGenresRow。

第二个参数是行（MovieGenresRow）字段中匹配该行 ID 属性的属性名称，例如 MovieId。

第三个参数是该行（MovieGenresRow）中关联多个流派 id 的字段属性名称，例如 GenreId。

*LinkingSetRelation* 具有相关的 Serenity 服务行为（Serenity service behavior），名为 *LinkingSetRelationBehavior*，所有含 *LinkingSetRelation* 特性的属性将自动激活该行为。

此行为，将拦截创建、更新、删除、检索和列表的处理服务，并注入代码来填充或更新我们的流派列表（GenreList）列及其相关影片流派（MovieGenres）表。

我们将在后面的章节谈论 Serenity 的服务行为。

## 在表单中添加流派列表

编辑 *MovieForm.cs* 并添加 *GenreList* 属性：

```
public class MovieForm
{
    //...
    public List<Int32> GenreList { get; set; }
    public MovieKind Kind { get; set; }
    public Int32 Runtime { get; set; }
}
```

现在我们可以为影片添加多个流派：

The screenshot shows the 'MovieTutorial' application interface. On the left is a sidebar with navigation links like 'Dashboard', 'Movie Database' (selected), 'Movies', 'Genres', 'Northwind', 'Basic Samples', 'Theme Samples', and 'Administration'. The main area has a title 'Movies' and a search bar. A modal dialog titled 'Edit Movie (Fight Club)' is open, showing a list of movies (ID 8: Fight Club, ID 6: Pulp Fiction, etc.) and input fields for 'Description', 'Storyline', 'Year' (1999), 'Release Date' (10/15/1999), 'Genres' (with 'Comedy' selected), and 'Kind' (with 'Documentary', 'Fantasy', and 'Sci-fi' listed). The bottom right of the dialog shows runtime values (139, 154, 175, 161, 178, 136, 142).

## 在列中显示所选的流派

以前，我们每部影片只有一个流派时，通过将可视字段添加到 `MovieRow.cs`，就可以在列中显示所选的流派，但这次就不会这么简单。

我们先把 `GenreList` 属性添加到 `MovieColumns.cs`：

```

public class MovieColumns
{
    //...
    [Width(200)]
    public List<Int32> GenreList { get; set; }
    [DisplayName("Runtime in Minutes"), Width(150), AlignRight]
    public Int32 Runtime { get; set; }
}

```

这是我们所得到的：

ID	Title	Description	Storyline	Year	Release Da...	Genres	Runtime in Minutes
8	Fight Club	An insomniac office w...	A ticking-time-bomb insomniac and a s...	1999	10/15/1999	1,2	139
6	Pulp Fiction	The lives of two mob ...	Jules Winnfield and Vincent Vega are t...	1994	10/14/1994	1	154
5	The Godfather	The aging patriarch of...	When the aging head of a famous crim...	1972	03/24/1972	2	175
7	The Good, the Bad an...	A bounty hunting sca...	Blondie (The Good) is a professional g...	1969	01/13/1969	2	161
3	The Lord of the Rings:...	A meek hobbit of the ...	An ancient Ring thought lost for centu...	2001	12/19/2001	5	178
2	The Matrix	A computer hacker le...	Thomas A. Anderson is a man living tw...	1999	03/31/1999	4	136
4	The Shawshank Redem...	Two imprisoned men...	Andy Dufresne is a young and success...	1994	10/14/1994	2	142

GenreList 列包含一个 Int32 类型的数值列表，该列表对应于 Javascript 数组。庆幸的是，Javascript 数组的 `.toString()` 方法返回用逗号分隔的项目，所以我们得到了影片 *Fight Club* 对应的流派 “1,2”。

我们更喜欢看到流派的名称而不是流派 ID，所以很显然，我们需要通过对这些值进行格式化来将流派 ID 转换为对应的流派名称。

## 创建 `GenreListFormatter`

现在是时候写一个 SlickGrid 列的格式化器了，在 `GenreListFormatter.ts` 旁边创建 `MovieGrid.ts`：

```

namespace MovieTutorial.MovieDB {

    @Serenity.Decorators.registerFormatter()
    export class GenreListFormatter implements Slick.Formatter {
        format(ctx: Slick.FormatterContext) {
            let idList = ctx.value as number[];
            if (!idList || !idList.length)
                return "";

            let byId = GenreRow.getLookup().itemById;

            return idList.map(x => {
                let g = byId[x];
                if (!g)
                    return x.toString();

                return Q.htmlEncode(g.Name);
            }).join(", ");
        }
    }
}

```

我们在这里定义了一个新的格式化器 *GenreListFormatter*，并在 Serenity 类型系统中使用 *@Serenity.Decorators.registerFormatter* 装饰注册它，装饰类似于 .NET 特性。

所有格式化器应该实现 *Slick.Formatter* 接口，该接口有一个 *format* 方法，此方法有一个 *Slick.FormatterContext* 类型的 *ctx* 参数。

参数 *ctx* 是格式化器上下文，具有一些成员的对象，其中有一个 *value* 成员，它包含当前正在格式化的网格行/列的列值。

因为我们知道将要对一个类型为 `List<Int32>` 的列值使用此格式化器，所以我们开始把值强制转换为 `number[]`。

在 Javascript 中没有 `Int32` 类型。`Int32`、`Double`、`Single` 等对应 Javascript 的 `number` 类型。另外，C# 中的泛型 `List<>` 对应于 Javascript 数组。

如果数据是空或者为 `null`，我们就安全地返回一个空字符串。

```
let idList = ctx.value as number[];
if (!idList || !idList.length)
    return "";
```

然后我们获得流派检索的引用，在其 *itemById* 属性中有一个流派字典：

```
let byId = GenreRow.getLookup().itemById;
```

接下来，我们开始使用 Javascript 的 *Array.map* 函数把 *idList* 的 ID 值映射为对应的流派名称，这非常类似于 LINQ 的 *Select* 语句：

```
return idList.map(x => {
```

我们在流派字典中检索 ID，ID 应该都在流派字典中，但我们这里为了安全起见，如果在字典中找不到该流派，就返回其数值。

```
let g = byId[x];
if (!g)
    return x.toString();
```

如果我们找到对应 ID 的流派，就返回它的名称，我们还对流派名称使用 *HTML* 编码，转换其可能含有的无效字符（如 `<`，`>` 或 `&`）。

```
return Q.htmlEncode(g.Name);
```

我们还可以编写一个泛型的格式化器，使其适用于任何类型的列表，但它已经超出了本教程的范围。

## 在流派列表列中使用 **GenreListFormatter**

我们定义一个新的格式化类，应该生成项目并转换 *T4* 模板文件，这样我们就可以在服务器端代码中引用 *GenreListFormatter*。

在生成和转换模板之后，打开 *MovieColumns.cs* 并添加该格式化器到 *MovieList* 属性：

```

public class MovieColumns
{
    //...
    [Width(200), GenreListFormatter]
    public List<Int32> GenreList { get; set; }
    [DisplayName("Runtime in Minutes"), Width(150), AlignRight]
    public Int32 Runtime { get; set; }
}

```

现在我们可以在流派列中看到其名称：

The screenshot shows a web-based movie database application. On the left, there's a sidebar with navigation links: Dashboard, Movie Database (Movies, Genres), Northwind, Basic Samples, Theme Samples, and Administration. The main content area is titled 'Movies' and displays a table of movie records. The columns are ID, Title, Description, Storyline, Year, Release Da..., Genres, and Runtime in Minutes. The 'Genres' column lists multiple genre names separated by commas. At the bottom of the page, there's a copyright notice 'Copyright (c) 2015. All rights reserved.' and a footer note 'Serenity Platform'.

ID	Title	Description	Storyline	Year	Release Da...	Genres	Runtime in Minutes
8	Fight Club	An insomniac office w...	A ticking-time-bomb insomniac and a s...	1999	10/15/1999	Action, Drama	139
6	Pulp Fiction	The lives of two mob ...	Jules Winnfield and Vincent Vega are t...	1994	10/14/1994	Action	154
5	The Godfather	The aging patriarch of...	When the aging head of a famous crim...	1972	03/24/1972	Drama	175
7	The Good, the Bad an...	A bounty hunting sca...	Blondie (The Good) is a professional g...	1969	01/13/1969	Drama	161
3	The Lord of the Rings:...	A meek hobbit of the ...	An ancient Ring thought lost for centu...	2001	12/19/2001	Fantasy	178
2	The Matrix	A computer hacker le...	Thomas A. Anderson is a man living tw...	1999	03/31/1999	Sci-fi	136
4	The Shawshank Redem...	Two imprisoned men...	Andy Dufresne is a young and success...	1994	10/14/1994	Drama	142

# 筛选具有多个流派的列表

记得当每部影片只有一个流派时，我们可以很容易实现快速过滤：在 `GenreId` 属性加入了 [QuickFilter] 特性即可。

让我们试着在 `MovieColumns.cs` 做类似修改：

```
[ColumnScript("MovieDB.Movie")]
[BasedOnRow(typeof(Entities.MovieRow))]
public class MovieColumns
{
    //...
    [Width(200), GenreListFormatter, QuickFilter]
    public List<Int32> GenreList { get; set; }
}
```

只要在 `Genres` 中输入流派就会得到该错误：

The screenshot shows a web application interface. At the top, there is a red error bar with the text "MovieTutorial Invalid column name 'GenreList'. Invalid column name 'GenreList'." Below the error bar, the main content area has a header "Movies". Underneath the header, there is a search bar and a "New Movie" button. A sidebar on the left is titled "Movie Database" and includes "Movies", "Genres", and "Actors-Actresses" sections. The main content area displays a table of movies with columns: ID, Title, Description, Storyline, Year, Release Da., Genres, and Runtime in Minutes. The "Genres" column header has a dropdown menu open with "Comedy" selected. The table contains 7 rows of movie data. At the bottom of the page, there is a navigation bar with "100", "Page 1 / 1", and "Showing 1 to 7 of 7 total records".

`ListHandler` 尝试使用 `GenreList` 字段过滤，但是在数据库中并没有这样的字段，因此我们得到这个错误。

事实上，`LinkingSetRelation` 会拦截此过滤器并把它转换为 `EXISTS` 子查询，但是列行为（list behaviors）还没有实现这样的功能，也许会在以后版本……

因此，我们现在就用某种方式来处理它。

## 声明 **MovieListRequest** 类型

因为我们打算做一些不规范的事，例如通过关联表（linking set table）的值来过滤时，我们需要防止 ListHandler 在 GenreList 属性中过滤自身。

我们可以使用一个访问者模式处理请求条件（*Criteria*）对象（它类似于表达式目录树）和处理 GenreList 自身，但这有点复杂。所以现在我会使用一个简单的方式。

让我们看一个含标准的 *ListRequest* 对象的子类，我们将在这里添加流派过滤器参数。在 *MovieRepository.cs* 文件旁边添加 *MovieListRequest.cs* 文件：

```
namespace MovieTutorial.MovieDB
{
    using Serenity.Services;
    using System.Collections.Generic;

    public class MovieListRequest : ListRequest
    {
        public List<int> Genres { get; set; }
    }
}
```

在列表请求对象中，我们添加一个 *Genres* 属性，它将保存我们在影片中过滤的流派（*Genres*）选项。

## 为新请求类型修改 **Repository/Endpoint**

为使我们的列表处理器（list handler）和服务使用新的列表请求类型，需要在几个地方做修改。

先从 *MovieRepository.cs* 开始：

```
public class MovieRepository
{
    //...
    public ListResponse<MyRow> List(IDbConnection connection, MovieListRequest request)
    {
        return new MyListHandler().Process(connection, request);
    }

    //...
    private class MyListHandler : ListRequestHandler<MyRow, MovieListRequest> { }
}
```

若要使用新类型，而不是 ListRequest，在 List 方法中，我们把 ListRequest 改为 MovieListRequest，并在 MyListHandler 中添加一个泛型参数。

在另一文件 *MovieEndpoint.cs* 中也做一些小修改，该类实际上是 web 服务：

```
public class MovieController : ServiceEndpoint
{
    //...
    public ListResponse<MyRow> List(IDbConnection connection, MovieListRequest request)
    {
        return new MyRepository().List(connection, request);
    }
}
```

现在是时候生成和转换模板，以使我们的 MovieListRequest 对象及相关服务方法能在客户端生效。

## 将快速过滤器移到流派参数

我们仍然有同样的错误，因为快速过滤器并不知道我们刚添加到列表的请求类型，还一直使用着 *Criteria* 参数。

需要拦截快速过滤项并将流派列表移到 *MovieListRequest* 的流派 (*Genres*) 属性。

编辑 *MovieGrid.ts* :

```
export class MovieGrid extends Serenity.EntityGrid<MovieRow, any> {

    //...
    protected getQuickFilters() {
        let items = super.getQuickFilters();

        var genreListFilter = Q.first(items, x =>
            x.field == MovieRow.Fields.GenreList);

        genreListFilter.handler = h => {
            var request = (h.request as MovieListRequest);
            var values = (h.widget as Serenity.LookupEditor).values;
            request.Genres = values.map(x => parseInt(x, 10));
            h.handled = true;
        };

        return items;
    }
}
```

**getQuickFilters** 是一个获取此网格列表的快速过滤器对象列表的方法。

默认情况下，网格列表枚举 *MovieColumns.cs* 中所有含 [QuickFilter] 特性的属性，并为其创建合适的快速过滤器对象。

我们从基类获取 QuickFilter 对象列表开始。

```
let items = super.getQuickFilters();
```

然后找到 *GenreList* 属性的快速过滤器对象：

```
var genreListFilter = Q.first(items, x =>
  x.field == MovieRow.Fields.GenreList);
```

实际上现在只有一个快速过滤器。

下一步是设置 *handler* 方法。在提交到列表服务之前，快速过滤器对象读取编辑器值并将其应用到请求的 *Criteria*（如果有多个）或 *EqualityFilter*（如果单个值）参数。

```
genreListFilter.handler = h => {
```

然后我们获得当前 *ListRequest* 引用：

```
var request = (h.request as MovieListRequest);
```

并读取检索编辑器（*LookupEditor*）中的当前值：

```
var values = (h.widget as Serenity.LookupEditor).values;
```

把该值设置到 *request.Genres* 属性：

```
request.Genres = values.map(x => parseInt(x, 10));
```

这是一个字符串列表的值，我们需要将它们转换为整数。

最后一步是设置 *handled* 为 *true*，要禁用快速过滤器对象的默认行为，因此它将不会使用自己设置的 *Criteria* 或 *EqualityFilter*：

```
h.handled = true;
```

现在，我们将不再有 无效的列名 *GenreList* 的错误，但 *Genres* 过滤器还没有应用到服务器端。

## 在仓储（**Repository**）中处理流派的过滤

在 *MovieRepository.cs* 文件对 *MyListHandler* 做如下修改：

```
private class MyListHandler : ListRequestHandler<MyRow, MovieListRequest>
{
    protected override void ApplyFilters(SqlQuery query)
    {
        base.ApplyFilters(query);

        if (!Request.Genres.IsEmptyOrNull())
        {
            var mg = Entities.MovieGenresRow.Fields.As("mg");

            query.Where(Criteria.Exists(
                query.SubQuery()
                    .From(mg)
                    .Select("1")
                    .Where(
                        mg.MovieId == fld.MovieId &&
                        mg.GenreId.In(Request.Genres))
                    .ToString())));
        }
    }
}
```

*ApplyFilters* 是一个应用过滤器指定的 *Criteria* 和 *EqualityFilter* 请求参数表的方法。这里是应用自定义过滤器的好地方。

如果需要做任何过滤，我们首先要检查 *Request.Genres* 是否是 null 或空列表。

接下来，我们获得一个别名为 *mg* 的字段 *MovieGenresRow* 引用。

```
var mg = Entities.MovieGenresRow.Fields.As("mg");
```

这里需要说明一下，我们还没有覆盖 Serenity 实体系统。

让我们从还没有别名的 *MovieGenresRow.Fields* 开始：

```
var x = MovieGenresRow.Fields;
new SqlQuery()
    .From(x)
    .Select(x.MovieId)
    .Select(x.GenreId);
```

如果我们写类似上述的查询，它输出的 SQL 会是这样的：

```
SELECT t0.MovieId, t0.GenreId FROM MovieGenres t0
```

除非特别指出，Serenity 总是分配 *t0* 到行的主表。即使我们命名 *MovieGenresRow.Fields* 为变量 *x*，它的别名仍将是 *t0*。

因为在编译时，*x* 不会存在并且 Serenity 已没有办法知道其变量的名称。Serenity 实体系统没有使用像 LINQ to SQL 或 Entity Framework 那样的表达式树。它使用非常简单的字符串/查询生成器。

所以，如果想要使用 *x* 作为别名，我们必须明确地声明：

```
var x = MovieGenresRow.Fields.As("x");
new SqlQuery()
    .From(x)
    .Select(x.MovieId)
    .Select(x.GenreId);
```

结果为：

```
SELECT x.MovieId, x.GenreId FROM MovieGenres x
```

在 *MovieRow* 实体的 *MyListHandler* 中，*t0* 已经被 *MovieRow* 字段使用。因此，为防止 *MovieGenresRow* 字段（名为 *fld*）的冲突，我需要把 *MovieGenresRow* 别名指定为 *mg*。

```
var mg = Entities.MovieGenresRow.Fields.As("mg");
```

我想实现的是这样的一个查询（就像我们会使用纯SQL）：

```
SELECT t0.MovieId, t0.Title, ... FROM Movies t0
WHERE EXISTS (
    SELECT 1
    FROM MovieGenres mg
    WHERE
        mg.MovieId = t0.MovieId AND
        mg.GenreId IN (1, 3, 5, 7)
)
```

因此，我向 query 对象的 Where 方法添加 WHERE 过滤器，使用 EXISTS 条件：

```
query.Where(Criteria.Exists(
```

然后开始写子查询：

```
query.SubQuery()
    .From(mg)
    .Select("1")
```

为子查询添加 where 声明：

```
.Where(
    mg.MovieId == fld.MovieId &&
    mg.GenreId.In(Request.Genres))
```

其实这里 fld 包含 MovieRow 字段的别名 t0。

由于 *Criteria.Exists* 方法需要一个字符串，所以我需要在末尾使用 *.ToString()* 方法把子查询转换为字符串。

是的，我注意到应该添加一个接受子查询的重载.....

```
.ToString()));
```

开始使用时，这种写法看上去可能有点陌生，但花点时间你就会明白，  
Serenity 查询系统与 SQL 有 99% 的相似。但它不能是具体的 SQL，因为我们  
需要在不同的语言（C#）工作。

现在，我们的 *GenreList* 属性过滤器就可以很好地工作了……

# 演员和角色

如果我们要像这样保存演员和角色记录：

<b>Actor/Actress</b>	<b>Character</b>
Keanu Reeves	Neo
Laurence Fishburne	Morpheus
Carrie-Anne Moss	Trinity

我们需要一张演员（MovieCast）表，其内容如：

<b>MovieCastId</b>	<b>Movield</b>	<b>PersonId</b>	<b>Character</b>
...	...	...	...
11	2 (Matrix)	77 (Keanu Reeves)	Neo
12	2 (Matrix)	99 (Laurence Fisburne)	Morpheus
13	2 (Matrix)	30 (Carrie-Anne Moss)	Trinity
...	...	...	...

很显然，我们还需要一张人员（Person）表，因为我们要通过其 id 关联演员信息。

这里用人员（Person）表示演员会更好，因为演员后来可能成为导演，编剧等。

## 创建人员（Person）和演员（MovieCast）表

现在是时候创建这两张表的迁移类（migration）：

MovieTutorial.Web/Modules/Common/Migrations/DefaultDB/  
DefaultDB\_20160528\_141600\_PersonAndMovieCast.cs:

```
using FluentMigrator;
using System;

namespace MovieTutorial.Migrations.DefaultDB
```

```

{
    [Migration(20151025170200)]
    public class DefaultDB_20160528_141600_PersonAndMovieCast : Migration
    {
        public override void Up()
        {
            Create.Table("Person").InSchema("mov")
                .WithColumn("PersonId").AsInt32().Identity()
                    .PrimaryKey().NotNullable()
                .WithColumn("Firstname").AsString(50).NotNullable()
                .WithColumn("Lastname").AsString(50).NotNullable()
                .WithColumn("BirthDate").AsDateTime().Nullable()
                .WithColumn("BirthPlace").AsString(100).Nullable()
                .WithColumn("Gender").AsInt32().Nullable()
                .WithColumn("Height").AsInt32().Nullable();

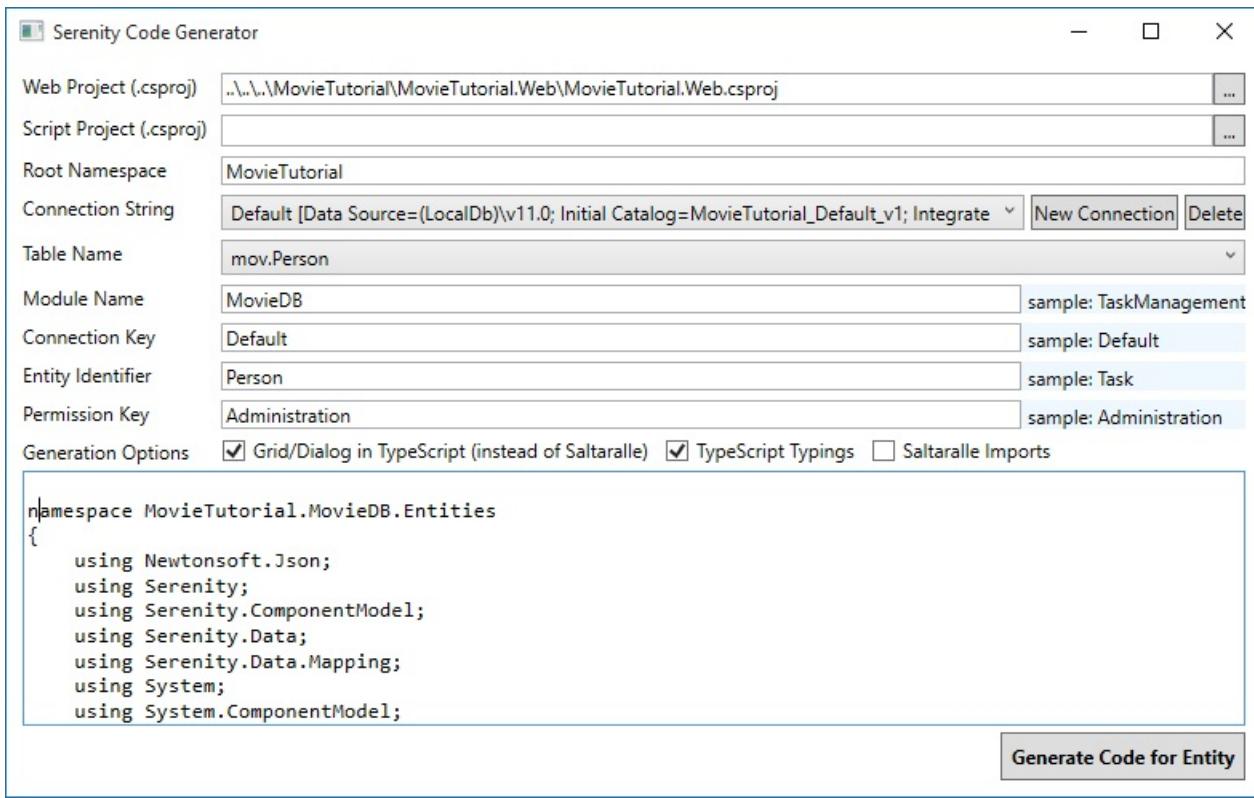
            Create.Table("MovieCast").InSchema("mov")
                .WithColumn("MovieCastId").AsInt32().Identity()
                    .PrimaryKey().NotNullable()
                .WithColumn("MovieId").AsInt32().NotNullable()
                    .ForeignKey("FK_MovieCast_MovieId", "mov", "Movie", "MovieId")
                .WithColumn("PersonId").AsInt32().NotNullable()
                    .ForeignKey("FK_MovieCast_PersonId", "mov", "Person", "PersonId")
                .WithColumn("Character").AsString(50).Nullable();
        }

        public override void Down()
        {
        }
    }
}

```

## 为人员（Person）表生成代码

我们首先为人员（Person）表生成代码：



## 把性别（Gender）修改为枚举

人员（Person）表的性别（Gender）列应该存放枚举类型的数据，在 PersonRow.cs 所在文件夹创建文件 Gender.cs，并定义一个 Gender 枚举：

```
using Serenity.ComponentModel;
using System.ComponentModel;

namespace MovieTutorial.MovieDB
{
    [EnumKey("MovieDB.Gender")]
    public enum Gender
    {
        [Description("Male")]
        Male = 1,
        [Description("Female")]
        Female = 2
    }
}
```

修改 PersonRow.cs 定义的 Gender 属性，如：

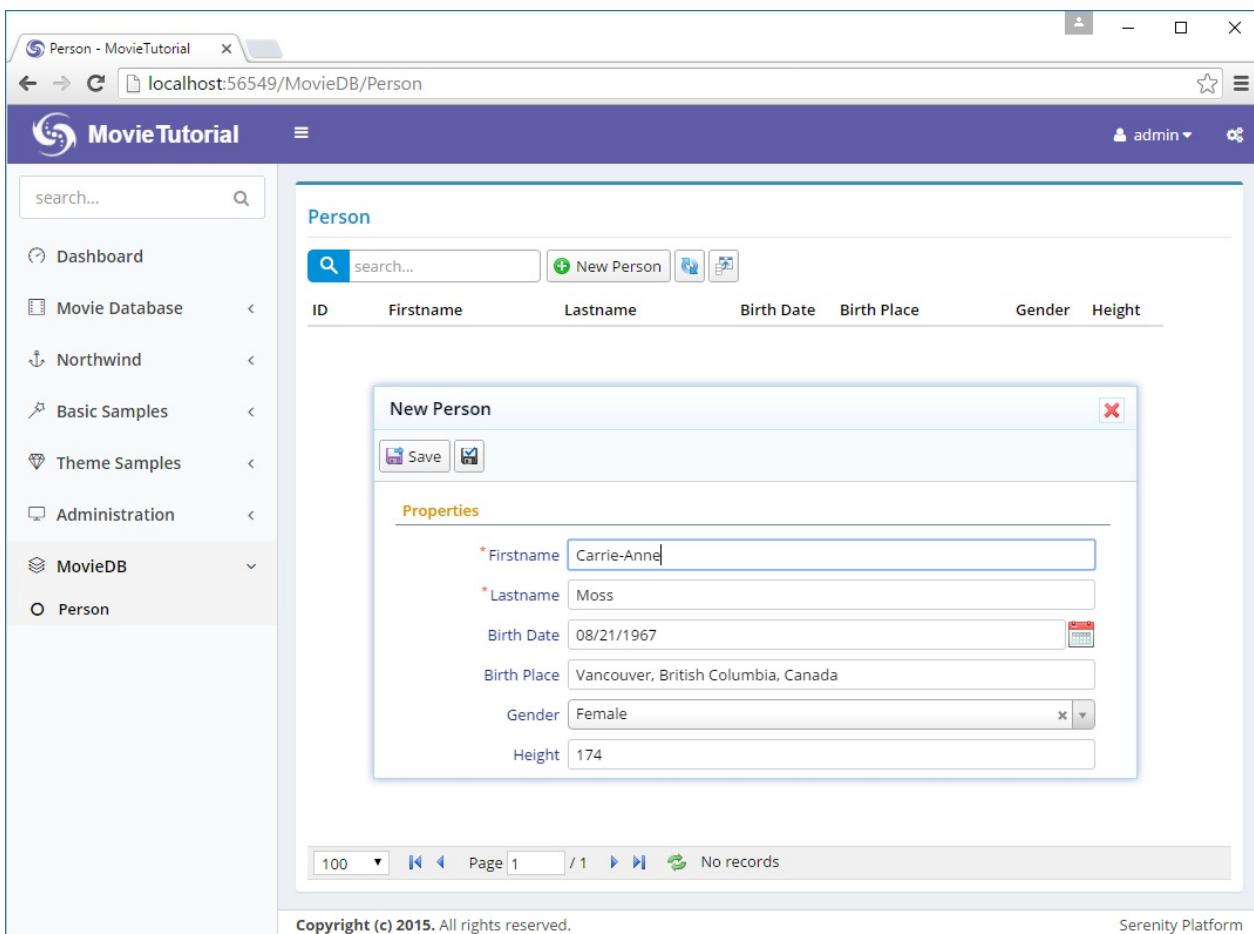
```
//...
[DisplayName("Gender")]
public Gender? Gender
{
    get { return (Gender?)Fields.Gender[this]; }
    set { Fields.Gender[this] = (Int32?)value; }
}
//...
```

为保持一致性，把 PersonForm.cs 和 PersonColumns.cs 的 Gender 属性类型 Int32 改为 Gender 枚举类型。

## 重新生成 T4 模板

当我们定义并使用一个新的枚举，我们应该重新生成解决方案，以便 T4 模板重新生成内容：

现在当你启动项目后，你应该可以输入演员信息：



## 声明 FullName 字段

编辑对话框的标题只显示人员的 姓氏(Carrie-Anne)，显示完整名字会更好，并且搜索也应该按列表中的全名进行搜索。

因此，让我们编辑 PersonRow.cs：

```
namespace MovieTutorial.MovieDB.Entities
{
    //...
    public sealed class PersonRow : Row, IIdRow, INameRow
    {
        //... remove QuickSearch from FirstName
        [DisplayName("First Name"), Size(50), NotNull]
        public String Firstname
        {
            get { return Fields.Firstname[this]; }
            set { Fields.Firstname[this] = value; }
        }
    }
}
```

```

[DisplayName("Last Name"), Size(50), NotNull]
public String Lastname
{
    get { return Fields.Lastname[this]; }
    set { Fields.Lastname[this] = value; }
}

[DisplayName("Full Name"),
 Expression("(t0.Firstname + ' ' + t0.Lastname)"), Quick
Search]
public String Fullname
{
    get { return Fields.Fullname[this]; }
    set { Fields.Fullname[this] = value; }
}

//... change NameField to Fullname
StringField INameRow.NameField
{
    get { return Fields.Fullname; }
}

//...

public class RowFields : RowFieldsBase
{
    public readonly Int32Field PersonId;
    public readonly StringField Firstname;
    public readonly StringField Lastname;
    public readonly StringField Fullname;
    //...
}
}
}

```

我们在 Fullname 属性上面添加 SQL 表达式 `Expression("(t0.Firstname + ' ' + t0.Lastname)")`，因此，这是一个在服务器端计算的字段。

通过给 FullName 属性添加 QuickSearch 特性，列表现在将默认以 Fullname 字段进行搜索，而不再以 Firstname 搜索。

但对话框显示的仍然是 Firstname。要显示 Fullname，我们需要生成项目让 T4 模板转换模板。

## 为什么必须转换模板吗？

查看 *PersonDialog.ts* 文件后你就会明白：

```
namespace MovieTutorial.MovieDB {  
  
    @Serenity.Decorators.registerClass()  
    @Serenity.Decorators.responsive()  
    export class PersonDialog extends Serenity.EntityDialog<PersonRow, any> {  
        protected getFormKey() { return PersonForm.formKey; }  
        protected getIdProperty() { return PersonRow.idProperty; }  
        protected getLocalTextPrefix() { return PersonRow.localTextPrefix; }  
        protected getNameProperty() { return PersonRow.nameProperty; }  
        protected getService() { return PersonService.baseUrl; }  
  
        protected form = new PersonForm(this.idPrefix);  
    }  
}
```

在这里我们看到 getNameProperty() 方法返回 PersonRow.nameProperty。TypeScript 文件(MovieDB.PersonRow.ts) 中的 PersonRow 就是由我们的 T4 模板生成的。

因此，除非我们转换 T4 模板，否则我们在 PersonRow.cs 中对 name 属性的更改将不会反映在 \*Modules/Common/Imports/ServerTypings/ ServerTypings.tt" 下的 MovieDB.PersonRow.ts 文件：

```

namespace MovieTutorial.MovieDB {
    export interface PersonRow {
        PersonId?: number;
        Firstname?: string;
        Lastname?: string;
        Fullname?: string;
        //...
    }

    export namespace PersonRow {
        export const idProperty = 'PersonId';
        export const nameProperty = 'Fullname';
        export const localTextPrefix = 'MovieDB.Person';

        export namespace Fields {
            export declare const PersonId: string;
            //...
        }
        //...
    }
}

```

此元数据（`PersonRow` 的 `nameProperty` 常量）是由 `ServerTypings.tt` 生成 `TypeScript` 文件(`MovieDB.PersonRow.ts`)的代码。

同样，`idProperty`、`localTextPrefix`、`Enum Types` 等也是由 `ServerTypings.tt` 文件生成的。因此，当你的更改对生成文件中的元数据有影响时，你就应该转换 T4 模板以将该修改信息应用到对应的 `TypeScript` 文件中。

你应该总是在转换模板之前生成项目，因为 T4 模板文件引用了 `MovieTutorial.Web` 项目输出的 DLL。否则你将会为一个旧版本的 Web 项目生成代码。

## 声明 **PersonRow** 检索脚本(Lookup Script)

让我们继续在 `PersonRow.cs` 文件中为 人员 (Person) 表添加一个 `LookupScript` 特性：

```
namespace MovieTutorial.MovieDB.Entities
{
    //...
    [LookupScript("MovieDB.Person")]
    public sealed class PersonRow : Row, IIdRow, INameRow
    //...
```

我们会在稍后的编辑影片中使用它。

再次生成项目，你将看到 *MovieDB.PersonRow.ts* 中有一个返回 `lookupKey` 类型的 `getLookup()` 方法。

```
namespace MovieTutorial.MovieDB {
    export interface PersonRow {
        //...
    }

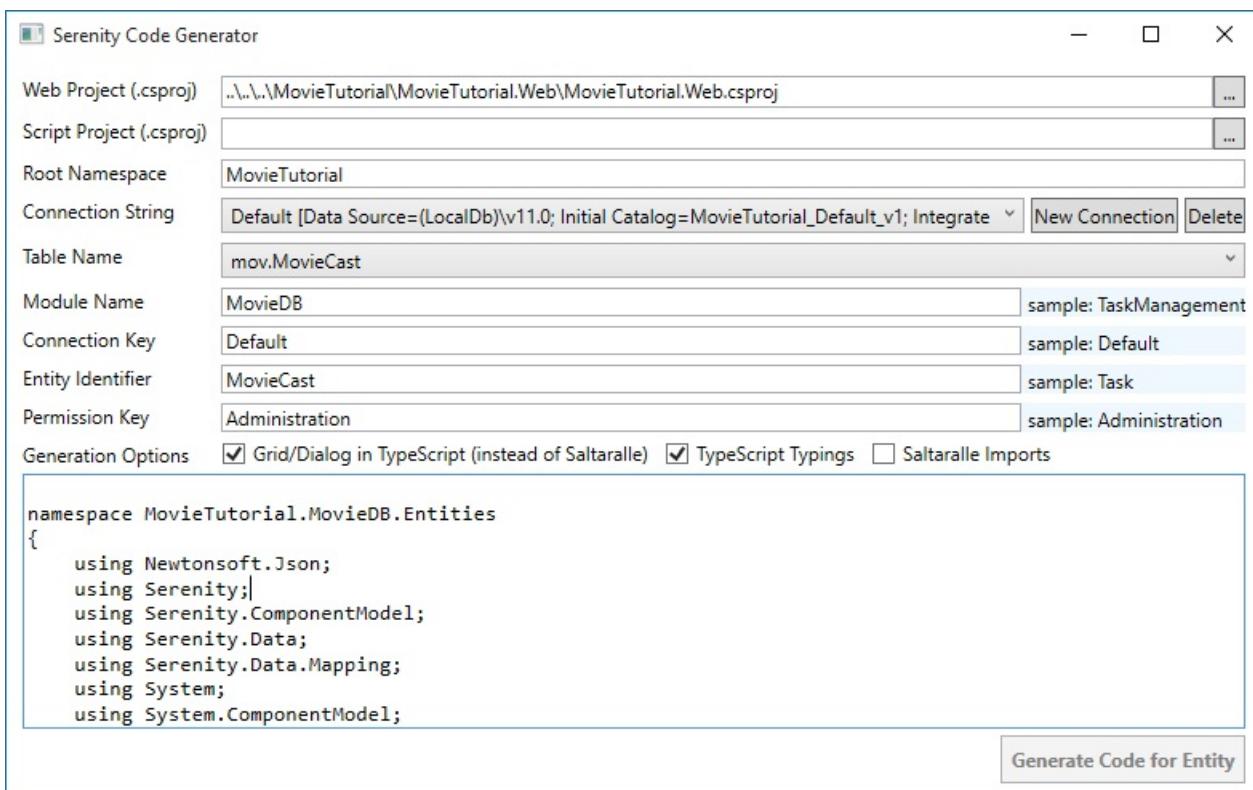
    export namespace PersonRow {
        export const idProperty = 'PersonId';
        export const nameProperty = 'Fullname';
        export const localTextPrefix = 'MovieDB.Person';
        export const lookupKey = 'MovieDB.Person';

        export function getLookup(): Q.Lookup<PersonRow> {
            return Q.getLookup<PersonRow>('MovieDB.Person');
        }

        //...
    }
}
```

## 为 演员 (**MovieCast**) 表生成代码

使用 `sergen` 为 演员 (**MovieCast**) 表生成代码：



生成代码之后，由于我们不需要一个单独的页面编辑 演员（MovieCast）表，所以你可以删除下列的文件：

```
MovieCastIndex.cshtml
MovieCastPage.cs
MovieDialog.ts
MovieGrid.ts
```

再次生成项目。

## 演员（MovieCast）表主从关系的编辑逻辑

到目前为止，我们为每张表创建一个页面，并在该页面显示列表和编辑记录。这一次，我们将使用不同的策略。

我们将在影片对话框中列出演员信息，并允许他们与影片一起进行编辑。此外，演员也将与影片实体在同一个事务中保存。

因此，编辑演员的信息将保存在内存中，当用户点击影片对话框中的保存按钮时，影片和其演员会同时（同一事务）保存到数据库。

也有可能需要单独编辑演员信息的情况，但我们这里仅演示其中一种实现方式。

对于一些主从关系，如订单/订单详细，由于需要保持主从表的一致性，从表不应该允许进行独立编辑。Serene 已经在 Northwind/Order 的编辑对话框中为该情况提供了示例。

## 创建演员（MovieCast）列表的编辑器

在 MovieCastRow.cs (位于 MovieTutorial.Web/Modules/MovieDB/MovieCast/) 目录下面，创建文件 MovieCastEditor.ts，其内容如下：

```
/// <reference path="../../Common/Helpers/GridEditorBase.ts" />

namespace MovieTutorial.MovieDB {
    @Serenity.Decorators.registerEditor()
    export class MovieCastEditor
        extends Common.GridEditorBase<MovieCastRow> {
        protected getColumnsKey() { return "MovieDB.MovieCast";
    }
        protected getLocalTextPrefix() { return MovieCastRow.localTextPrefix; }

        constructor(container: JQuery) {
            super(container);
        }
    }
}
```

此编辑器继承自 Serene 的 Common.GridEditorBase 类，这是一种在内存中编辑内容的特殊网格类型。它也是订单对话框中订单详细信息编辑器的基类。

在文件顶部的 `<reference />` 很重要。TypeScript 有输入文件的顺序问题。如果我们不把它放在那里，TypeScript 有时会在输出 MovieCastEditor 之后再输出 GridEditorBase，导致出现运行时错误。

作为一个经验法则，如果你有一些类继承自另一个项目（不是 Serenity 的类），你应该在文件中包含该基类文件的引用。

这有助于在其他类使用 GridEditorBase 之前，TypeScript 把它转换为 javascript。

若要从服务器端引用此新的编辑器类型，则需要生成并转换所有模板。

此基类可能会在以后的版本中集成到 Serenity。在这种情况下，其命名空间可能会变为 Serenity，代替当前的 Serene 或 MovieTutorial。

## 在影片窗体中使用 MovieCastEditor

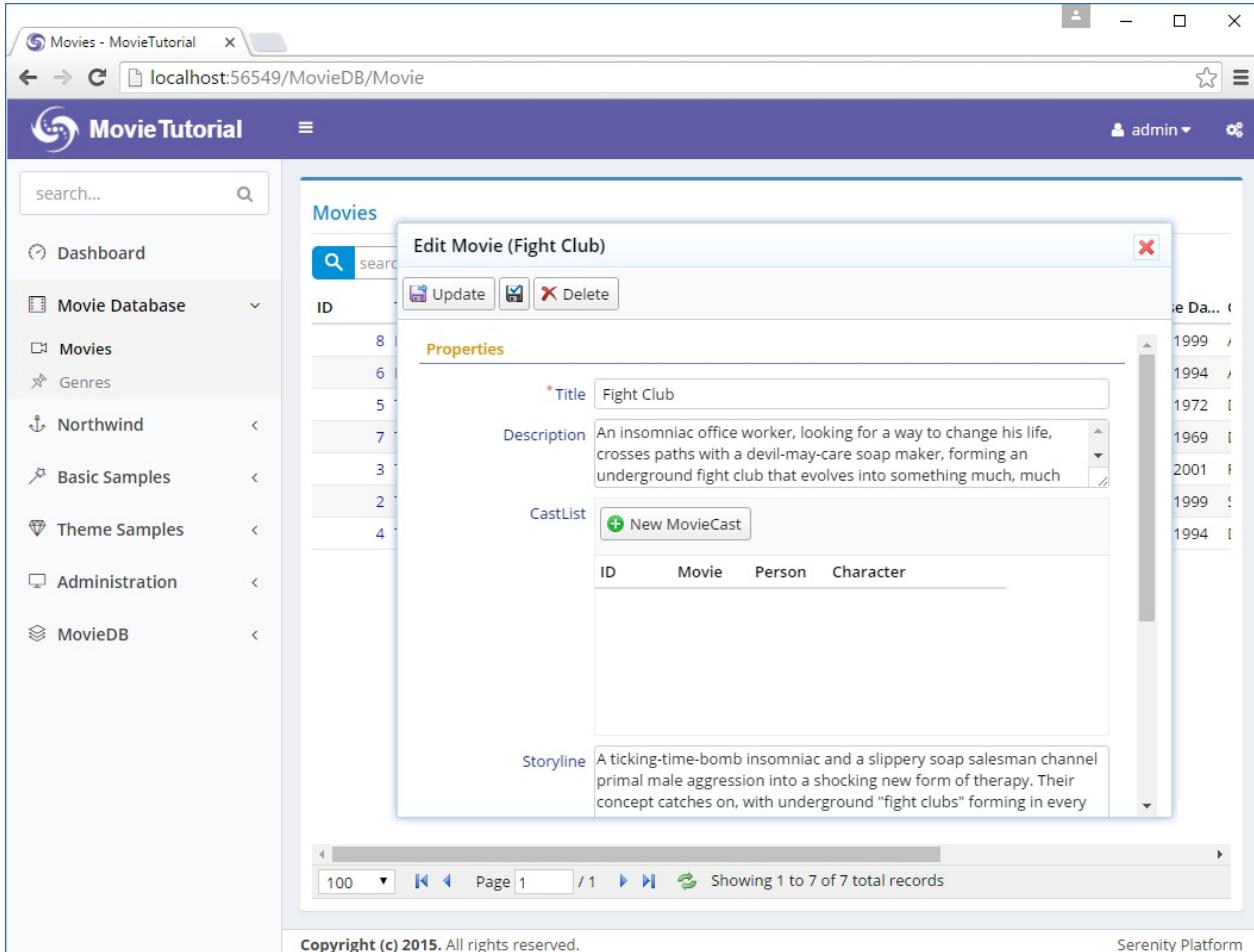
打开 MovieForm.cs，在 *Description* 和 *Storyline* 属性之间添加 *CastList* 属性，如：

```
namespace MovieTutorial.MovieDB.Forms
{
    //...
    public class MovieForm
    {
        public String Title { get; set; }
        [TextAreaEditor(Rows = 3)]
        public String Description { get; set; }
        [MovieCastEditor]
        public List<Entities.MovieCastRow> CastList { get; set;
    }
    [TextAreaEditor(Rows = 8)]
    public String Storyline { get; set; }
    //...
}
```

通过把 `[MovieCastEditor]` 特性放在 *CastList* 属性上，我们指定该属性将由新的 MovieCastEditor 类型所定义的 TypeScript 代码编辑。

我们也可以像这样添加特性 `[EditorType("MovieDB.MovieCast")]`，但谁真的喜欢硬编码的字符串呢？我可不喜欢...

现在生成并启动你的应用程序。打开一个影片编辑对话框，你将看到我们的新编辑器：



OK，这看起来很容易完成该功能，但我实话告诉你，我们连功能的一半都还没完成。

*New MovieCast* 按钮不能工作，需要为它定义一个对话框；网格显示的列不是我要的并且字段和按钮的标题很不友好.....

此外，我们还有更多的细节需要处理，如在服务器端保存和加载演员列表（我们先用手工方式演示其有多困难，然后再用服务行为（service behavior）来演示其有多简单）

## 配置演员编辑器（MovieCastEditor）使用编辑演员对话框（MovieCastEditDialog）

在 *MovieCastEditor.ts* 所在文件夹中创建文件 *MovieCastEditDialog.ts*，并对其做如下修改：

```
/// <reference path="../../Common/Helpers/GridEditorDialog.ts" />

namespace MovieTutorial.MovieDB {

    @Serenity.Decorators.registerClass()
    export class MovieCastEditDialog extends
        Common.GridEditorDialog<MovieCastRow> {
        protected getFormKey() { return MovieCastForm.formKey; }
        protected getNameProperty() { return MovieCastRow.namePr
        operty; }
        protected getLocalTextPrefix() { return MovieCastRow.loc
        alTextPrefix; }

        protected form: MovieCastForm;

        constructor() {
            super();
            this.form = new MovieCastForm(this.idPrefix);
        }
    }
}
```

我们使用另外一个来自 Serene 的基类 *Common.GridEditorDialog*，编辑订单详细对话框（*OrderDetailEditDialog*）也是使用该基类。

再次打开 *MovieCastEditor.ts* 文件，添加一个 *getDialogType* 方法并重写 *getAddButtonCaption* 方法：

```
/// <reference path="../../Common/Helpers/GridEditorBase.ts" />

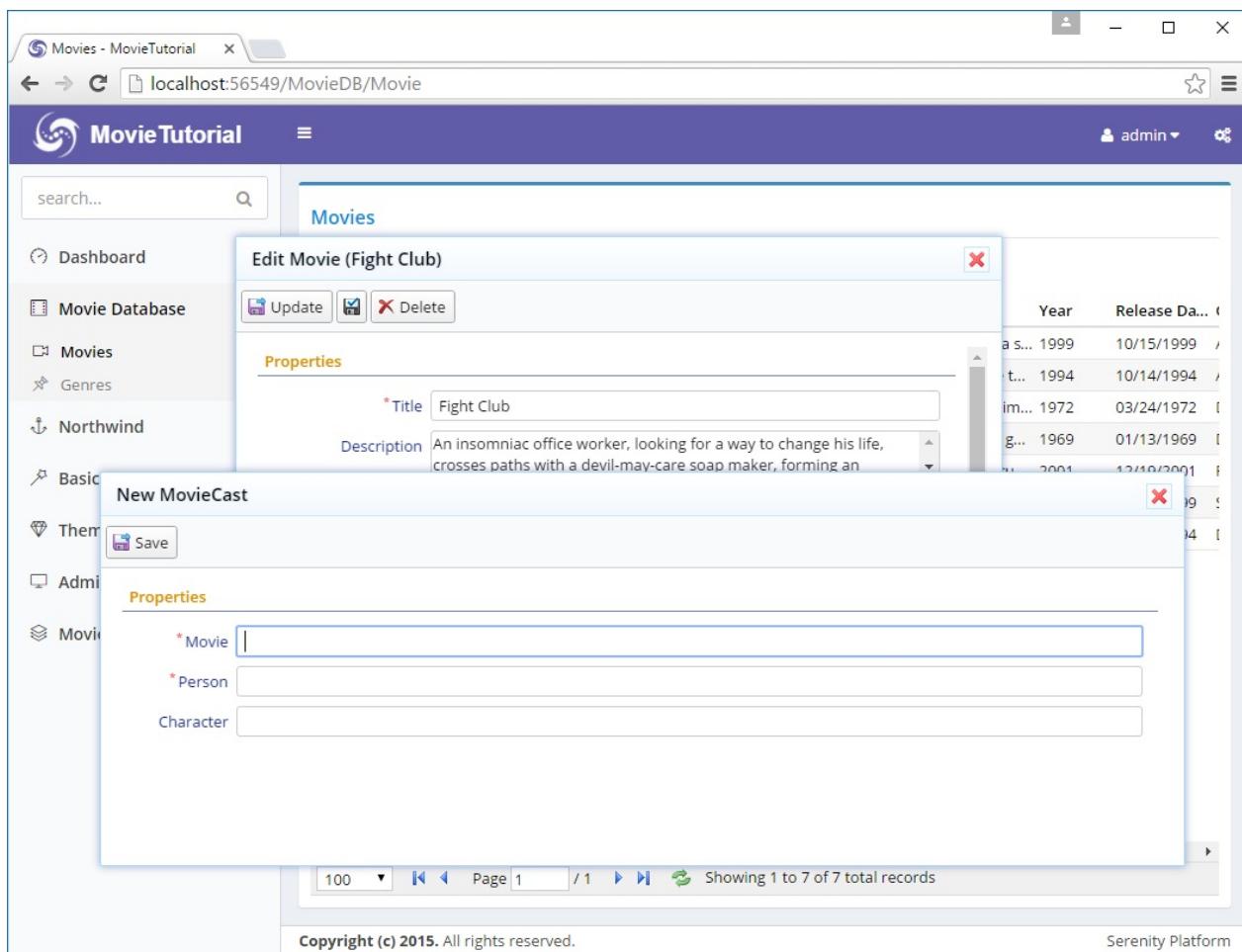
namespace MovieTutorial.MovieDB {
    @Serenity.Decorators.registerEditor()
    export class MovieCastEditor
        extends Common.GridEditorBase<MovieCastRow> {
        protected getColumnsKey() { return "MovieDB.MovieCast";
    }
        protected getDialogType() { return MovieCastEditDialog;
    }
        protected getLocalTextPrefix() { return MovieCastRow.localTextPrefix; }

        constructor(container: JQuery) {
            super(container);
        }

        protected getAddButtonCaption() {
            return "Add";
        }
    }
}
```

我们指定演员编辑器（MovieCastEditor）默认使用编辑演员对话框（MovieCastEditDialog），*Add* 按钮也使用该对话框。

现在，*Add* 按钮将显示一个对话框，而不再什么也没做。



此对话框需要一些 CSS 美化、电影标题和人员名字段接受数字输入（因为它们实际上是 `MovieId` 和 `PersonId` 字段）。

## 编辑 `MovieCastForm.cs` 文件

`MovieCastEditDialog` 的 `getFormKey()` 方法返回 `MovieCastForm.formKey`, 所以它当前使用 Sergen 生成的 `MovieCastForm.cs`。

在 Serenity 中也有可能一个实体有多个窗体。如果我想要一些其他独立的对话框保存 `MovieCastForm`, 例如 演员对话框 (`MovieCastDialog`, 我们已经把它删除了), 我更愿意定义一个像 `MovieCastEditForm` 的新窗体, 但在这里我没有这样做。

打开并编辑 `MovieCastForm.cs` :

```

namespace MovieTutorial.MovieDB.Forms
{
    using Serenity.ComponentModel;
    using System;
    using System.ComponentModel;

    [FormScript("MovieDB.MovieCast")]
    [BasedOnRow(typeof(Entities.MovieCastRow))]
    public class MovieCastForm
    {
        public Int32 PersonId { get; set; }
        public String Character { get; set; }
    }
}

```

我已经删除了 **Movield**，因为这个表单在 编辑演员对话框 (**MovieCastEditDialog**) 中使用，所以在 影片对话框 (**MovieDialog**) 中，演员 (**MovieCast**) 实体自动带有当前被编辑电影的 **Movield**，打开 《魔戒》 (*Lord of the Rings*)，并添加《黑客帝国》 (*the Matrix*) 的演员列表是没有意义的。

下一步，编辑 **MovieCastRow.cs**：

```

[ConnectionKey("Default"), TwoLevelCached]
[DisplayName("Movie Casts"), InstanceName("Cast")]
[ReadPermission("Administration")]
[ModifyPermission("Administration")]
public sealed class MovieCastRow : Row, IIdRow, INameRow
{
    //...
    [DisplayName("Actor/Actress"), NotNull, ForeignKey("[mov
].[Person]", "PersonId")]
    [LeftJoin("jPerson"), TextualField("PersonFirstname")]
    [LookupEditor(typeof(PersonRow))]
    public Int32? PersonId
    {
        get { return Fields.PersonId[this]; }
        set { Fields.PersonId[this] = value; }
    }
}

```

我为 `PersonId` 属性设置 `LookupEditor` 特性，正如我在 `PersonRow` 添加 `LookupScript` 特性一样，我可以重用这些信息来设置检索键。

我们也可以写成

```
[LookupEditor("MovieDB.Person")]
```

把 `PersonId` 的显示名称修改为 `Actor/Actress`。

同时修改行的 `DisplayName` 和 `InstanceName` 特性来设置对话框标题。

生成解决方案，并启动项目，现在 `MovieCastEditDialog` 有更好的编辑体验。但对话框宽度和高度仍然太大了。

## 美化 编辑演员对话框 (`MovieCastEditDialog`)

让我们检查 `site.less` 来了解为什么我们的 `MovieCastEditDialog` 不会应用样式。

```
.s-MovieDB-MovieCastDialog {  
    > .size { width: 650px; }  
    .caption { width: 150px; }  
}
```

`Site.less` 底部 CSS 是 `MovieCastDialog`，而不是 `MovieCastEditDialog`，因为该类样式是我们自己定义的，而不是用代码生成器生成。

我们创建了一个名为 `MovieCastEditDialog` 新对话框，所以现在我们的新对话框只有一个 `s-MovieDB-MovieCastEditDialog` 样式类，但是代码生成器只生成样式 `s-MovieDB-MovieCastDialog`。

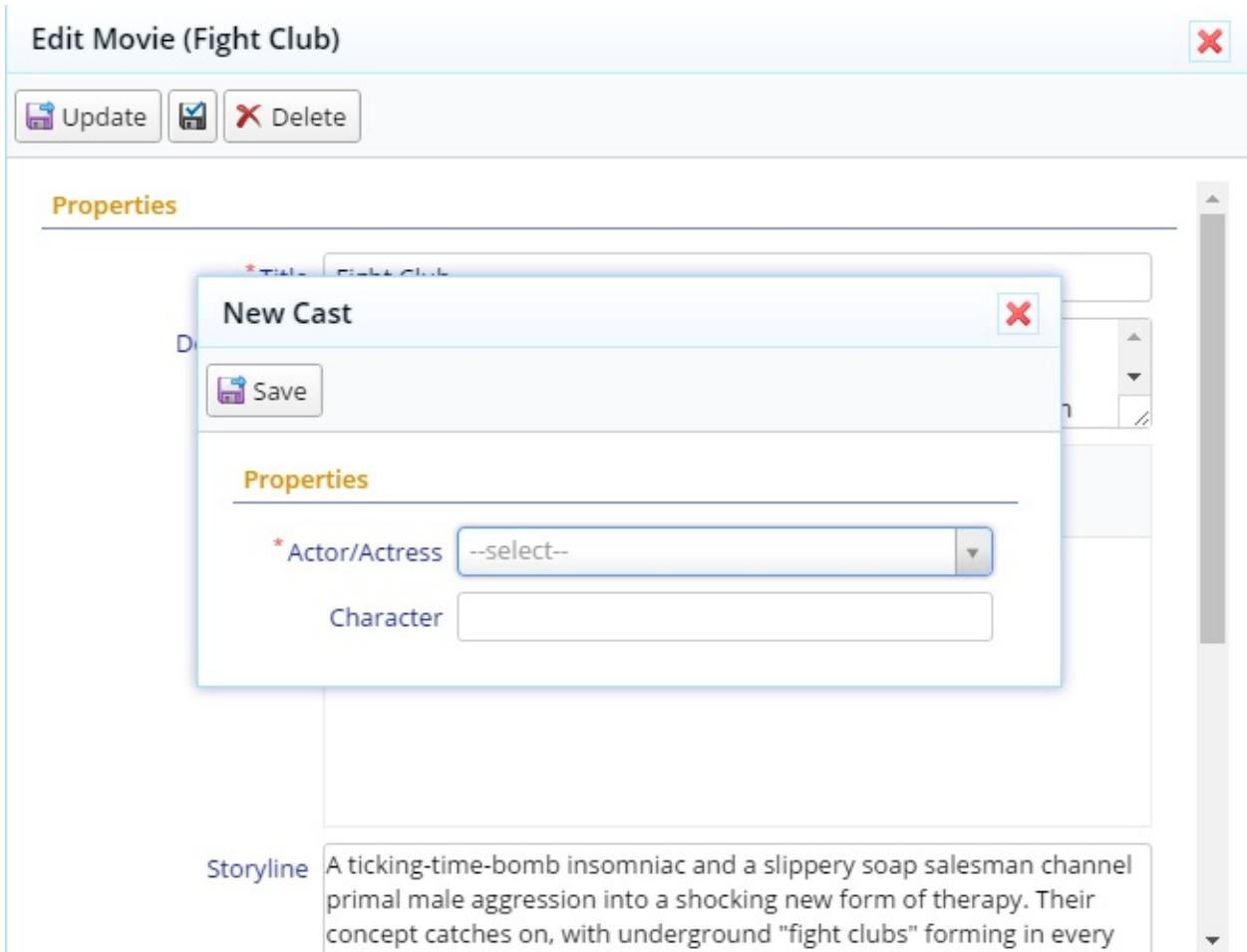
Serenity 将自动为对话框元素分配以“`s-`”为前缀的 CSS 类名。你可以在开发人员工具中通过该特点检查对话框。`MovieCastEditDialog` 有 `s-MovieCastEditDialog` 和 `s-MovieDB-MovieCastEditDialog` 及一些其它类似 `ui-dialog` 的 CSS 类。

`s-ModuleName-TypeName` CSS 类帮助区分两个具有相同名称模块的样式。

由于我们实际上并没有打算使用演员对话框 (`MovieCastDialog`，我们已经把它删除了)，让我们在 `site.less` 重命名一个类：

```
.s-MovieDB-MovieCastEditDialog {  
    > .size { width: 450px; }  
    .caption { width: 120px; }  
    .s-PropertyGrid .categories { height: 120px; }  
}
```

现在 编辑演员对话框 (*MovieCastEditDialog*) 更为美观了：



## 调整演员编辑器 (**MovieCastEditor**) 的列

演员编辑器 (*MovieCastEditor*) 目前使用在 *MovieCastColumns.cs* 定义的列 (因为它在 *getColumnsKey()* 方法中返回 "MovieDB.MovieCast" ) 。

我们这里有 *MovieCastId*、*MovielId*、*PersonId* (显示为 *Actor/Actress*) 和 *Character* 字段，只显示 *Actor/Actress* 和 *Character* 字段会更好。

我们想显示演员的全名而不是整数值 (*PersonId* 是整数)，所以我们首先在 *MovieCastRow.cs* 中定义该属性：

```
namespace MovieTutorial.MovieDB.Entities
{
    //...
    public sealed class MovieCastRow : Row, IIdRow, INameRow
    {
        // ...

        [DisplayName("Person Firstname"), Expression("jPerson.Firstname")]
        public String PersonFirstname
        {
            get { return Fields.PersonFirstname[this]; }
            set { Fields.PersonFirstname[this] = value; }
        }

        [DisplayName("Person Lastname"), Expression("jPerson.Lastname")]
        public String PersonLastname
        {
            get { return Fields.PersonLastname[this]; }
            set { Fields.PersonLastname[this] = value; }
        }

        [DisplayName("Actor/Actress"),
         Expression("(jPerson.Firstname + ' ' + jPerson.Lastname)")]
        public String PersonFullname
        {
            get { return Fields.PersonFullname[this]; }
            set { Fields.PersonFullname[this] = value; }
        }

        // ...

        public class RowFields : RowFieldsBase
        {
            // ...
            public readonly StringField PersonFirstname;
            public readonly StringField PersonLastname;
            public readonly StringField PersonFullname;
        }
    }
}
```

```
// ...
}
}
}
```

并修改 MovieCastColumns.cs :

```
namespace MovieTutorial.MovieDB.Columns
{
    using Serenity.ComponentModel;
    using System;

    [ColumnsScript("MovieDB.MovieCast")]
    [BasedOnRow(typeof(Entities.MovieCastRow))]
    public class MovieCastColumns
    {
        [EditLink, Width(220)]
        public String PersonFullname { get; set; }
        [EditLink, Width(150)]
        public String Character { get; set; }
    }
}
```

重新生成项目，现在网格列表有更友好的列：

Edit Movie (The Matrix) ✖

Update Delete

**Properties**

* Title	The Matrix				
Description	A computer hacker learns from mysterious rebels about the true nature of his reality and his role in the war against its controllers.				
CastList	<span style="border: 1px solid #ccc; padding: 2px 10px; margin-right: 10px;">+ Add</span>				
<table border="1" style="width: 100%;"><thead><tr><th>Actor/Actress</th><th>Character</th></tr></thead><tbody><tr><td></td><td></td></tr></tbody></table>		Actor/Actress	Character		
Actor/Actress	Character				
Storyline	Thomas A. Anderson is a man living two lives. By day he is an average computer programmer and by night a hacker known as Neo. Neo has always questioned his reality, but the truth is far				

现在尝试添加演员信息，例如，Keanu Reeves / Neo :

Edit Movie (The Matrix) ✖

Update Delete

**Properties**

* Title	The Matrix				
Description	A computer hacker learns from mysterious rebels about the true nature of his reality and his role in the war against its controllers.				
CastList	<span style="border: 1px solid #ccc; padding: 2px 10px; margin-right: 10px;">+ Add</span> <table border="1" style="width: 100%; border-collapse: collapse; text-align: center;"> <thead> <tr> <th>Actor/Actress</th> <th>Character</th> </tr> </thead> <tbody> <tr> <td></td> <td>Neo</td> </tr> </tbody> </table>	Actor/Actress	Character		Neo
Actor/Actress	Character				
	Neo				
Storyline	Thomas A. Anderson is a man living two lives. By day he is an average computer programmer and by night a hacker known as Neo. Neo has always questioned his reality, but the truth is far				

为什么 Actor/Actress 列是空的？

## 解决 Actor/Actress 列为空的问题

请记住，我们的编辑是在内存中进行的。这里有没有涉及服务的调用。因此，网格列表显示从对话框传给它的任何实体。

当你点击保存按钮时，对话框生成像这样的实体进行保存：

```
{
  PersonId: 7,
  Character: 'Neo'
}
```

这些字段对应你之前在 MovieCastForm.cs 设置的表单字段：

```
public class MovieCastForm
{
    public Int32 PersonId { get; set; }
    public String Character { get; set; }
}
```

但是在网格列表中，我们显示的是这些列：

```
public class MovieCastColumns
{
    public String PersonFullname { get; set; }
    public String Character { get; set; }
}
```

在该实体中并没有 **PersonFullname** 属性，所以网格列表不会显示它的值。

我们需要自己设置 **PersonFullname** 属性。首先，转换 T4 模板以获得我们最近添加的 **PersonFullname** 属性，然后编辑 **MovieCastEditor.ts**：

```

/// <reference path="../../Common/Helpers/GridEditorBase.ts" />

namespace MovieTutorial.MovieDB {
    @Serenity.Decorators.registerEditor()
    export class MovieCastEditor extends Common.GridEditorBase<MovieCastRow> {
        // ...

        protected validateEntity(row: MovieCastRow, id: number) {
            if (!super.validateEntity(row, id))
                return false;

            row.PersonFullname = PersonRow.getLookup()
                .itemById[row.PersonId].Fullname;

            return true;
        }
    }
}

```

`ValidateEntity` 是一个 Serene 的 `GridEditorBase` 类的方法，点击保存按钮，在实体添加到网格列表之前调用该方法对实体进行验证，但是我们需要重写它，让其设置 `PersonFullname` 属性的值而不是验证。

正如我们之前看到的，我们的实体有 `PersonId` 和 `Character` 字段。我们可以使用 `PersonId` 字段的值来确定角色的全名。

为此，我们需要一个把 `PersonId` 对应到角色全名的字典。幸运的是，人员检索 (`lookup`) 有这样的字典。我们可以通过其 `getLookup` 方法来访问 `PersonRow` 的检索 (`lookup`)。

另外一种访问人员检索 (`lookup`) 的方法是使用 `Q.getLookup('MovieDB.Person')`。`PersonRow` 中的这个方法是 T4 模板定义的快捷方式。

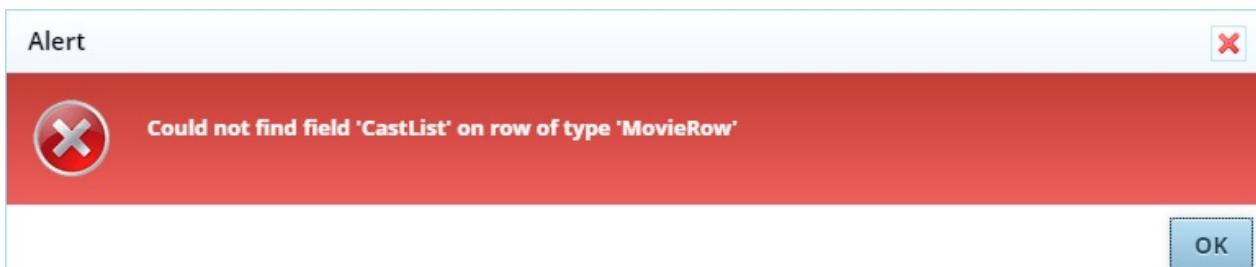
所有的检索 (`lookups`) 都有一个 `itemById` 字典，以允许你通过其 ID 访问该实体类。

检索（lookups）是一个简单的服务器端与客户端共享数据的方法。但他们只适用于小型数据集。

如果一个表有成千上万条记录，就没有理由为其定义检索(lookup)。在这情况下，我们会用一个服务器请求并通过其 ID 来查询记录。

## 在 MovieRow 中声明 CastList

当有一个影片对话框打开时，且演员列表（CastList）中没有演员，这时点击保存按钮，你会得到这样的错误：



引发该错误的是服务器端的行反序列化器 (JsonRowConverter for JSON.NET)。

我们在 MovieForm 定义 CastList 属性，但是在 MovieRow 没有对应的字段，所以反序列器不知道如何处理从客户端接收的 CastList 值。

如果你使用 F12 打开开发者工具，选择网格选项卡，并在点击保存按钮之后查看 AJAX 请求，你将看到有这样一个请求：

```
{  
    "Entity": {  
        "Title": "The Matrix",  
        "Description": "A computer hacker...",  
        "CastList": [  
            {  
                "PersonId": "1",  
                "Character": "Neo",  
                "PersonFullscreen": "Keanu Reeves"  
            }  
        ],  
        "Storyline": "Thomas A. Anderson is a man living two lives...",  
        "Year": 1999,  
        "ReleaseDate": "1999-03-31",  
        "Runtime": 136,  
        "GenreId": "",  
        "Kind": "1",  
        "MovieId": 1  
    }  
}
```

在这里，`CastList` 属性不能在服务器端进行反序列化。所以我们要在 `MovieRow.cs` 中声明：

```
namespace MovieTutorial.MovieDB.Entities
{
    // ...
    public sealed class MovieRow : Row, IIdRow, INameRow
    {
        [DisplayName("Cast List"), SetFieldFlags(FieldFlags.ClientSide)]
        public List<MovieCastRow> CastList
        {
            get { return Fields.CastList[this]; }
            set { Fields.CastList[this] = value; }
        }

        public class RowFields : RowFieldsBase
        {
            // ...
            public readonly RowListField<MovieCastRow> CastList;
            // ...
        }
    }
}
```

我们定义一个接受列表中 MovieCastRow 对象的 CastList 属性。用于这种行列属性的 *Field* 类的类型是 *RowListField*。

通过添加 *[SetFieldFlags(FieldFlags.ClientSide)]* 特性，我们指定此字段直接在数据库表中不可用，因此不能通过简单的 SQL 查询来选择。它在其他 ORM 系统中，类似于未映射的字段。

现在，当你点击保存按钮，将不会有错误发生。

但是，重新打开刚保存的《黑客帝国》（the Matrix）实体。这里并没有演员记录，尼欧（Neo）发生了什么事？

因为这是一个未映射的字段，所以电影保存服务忽略了 CastList 属性。

如果你还记得在前一节我们的 `GenreList` 同样也是一个未映射的字段，但不知为何它在那里可以正常工作。这是因为我们在该属性中使用了 `LinkedSetRelationBehavior` 行为（behavior）。

我们在这里演示若没有该服务器行为（service behavior）将会发生什么。

## 处理演员列表（**CastList**）的保存

打开 `MovieRepository.cs`，找到空 `MySaveHandler` 类，并对其做如下修改：

```
private class MySaveHandler : SaveRequestHandler<MyRow>
{
    protected override void AfterSave()
    {
        base.AfterSave();

        if (Row.CastList != null)
        {
            var mc = Entities.MovieCastRow.Fields;
            var oldList = IsCreate ? null :
                Connection.List<Entities.MovieCastRow>(
                    mc.MovieId == this.Row.MovieId.Value);

            new Common.DetailListSaveHandler<Entities.MovieCastR
ow>(
                oldList, Row.CastList,
                x => x.MovieId = Row.MovieId.Value).Process(this
.UnitOfWork);
        }
    }
}
```

`MySaveHandler` 处理影片行（Movie rows）的添加和修改服务请求。其大部分逻辑由基类 `SaveRequestHandler` 处理，所以之前该类是空的。

在添加/修改演员列表之前，我们应该先成功添加/修改影片实体。因此，我们通过重写基方法 `AfterSave` 包含自定义的代码。

如果是添加操作，我们需要在演员（MovieCast）纪录中重用 MovieId 字段的值。由于 MovieId 是一个标识字段，所以它只在添加影片记录之后生效。

由于我们是在内存（客户端）中编辑演员列表信息，所以这将是一个批处理更新。

我们需要比较影片的新旧演员列表记录，并对其进行添加/更新/删除操作。

假设我们数据库中有影片 X，演员有：A、B、C、D。

我们在编辑对话框中对演员列表做了一些修改，现在演员变为：A、B、D、E、F。

因此我们需要更新 A、B、D (角色/演员发生了改变)，删除 C，并添加新的演员记录 E 和 F。

幸运的是，Serene 中定义的 `DetailListSaveHandler` 类可处理所有这些比较，并自动执行插入/更新/删除操作（通过 ID 值）。否则我们需要在这里编写大量的代码。

如果这是一个更新影片的操作，为了获取演员列表中的旧记录，我们需要查询数据库。而如果是新增操作，我们就不需要任何演员记录。

我们使用 `Connection.List<Entities.MovieCastRow>` 扩展方法获取演员列表，这里的 `Connection` 是 `SaveRequestHandler` 的属性，该属性返回当前使用的连接。`List` 选择匹配指定条件 (`mc.MovieId == this.Row.MovieId.Value`) 的记录。

`this.Row` 是指添加/更新当前含有新的字段值的记录（影片记录），因此它包含 MovieId 值（新的或者现有的 ID）。

为了更新演员记录，我们创建了一个 `DetailListHandler` 对象，该对象含旧的演员列表、新的演员列表及设置演员记录 MovieId 字段值的委托。让对象为新的演员记录与当前电影建立联系。

然后我们使用当前的工作单元（unit of work）调用 `DetailListHandler` 方法。

`UnitOfWork` 是一个特殊的对象，它封装了当前连接/事务。

所有的 Serenity 添加/更新/删除 处理都适用于隐式事务(IUnitOfWork)。

## 处理演员列表的检索

我们还没有完成该功能。当在影片列表中点击影片实体时，对话框调用 `Retrieve` 服务加载影片记录。就像演员列表（CastList）字段没有映射的情况，即使我们正确保存了演员，演员也不会加载到对话框。

要解决该问题，我们同样需要在 MovieRepository.cs 文件中编辑 *MyRetrieveHandler* 类：

```
private class MyRetrieveHandler : RetrieveRequestHandler<MyRow>
{
    protected override void OnReturn()
    {
        base.OnReturn();

        var mc = Entities.MovieCastRow.Fields;
        Row.CastList = Connection.List<Entities.MovieCastRow>(q
=> q
            .SelectTableFields()
            .Select(mc.PersonFullscreen)
            .Where(mc.MovieId == Row.MovieId.Value));
    }
}
```

在这里，我们重写 *OnReturn* 方法，以使在检索服务返回之前把 *CastList* 注入影片的行（*Row*）对象中。

我使用 *Connection.List* 的不同扩展，这样我就可以修改 *Select* 查询。

默认情况下，列表选择所有的表字段(除了外键表的字段)，但为了显示演员名字，我还需要选择 *PersonFullName* 字段。

现在生成解决方案，我们终于可以显示/编辑演员了。

## 处理演员列表的删除

当你尝试删除电影实体时，你将获得一个外键错误。你可以在创建演员（MovieCast）表时使用 "级联删除（CASCADE DELETE）" 外键，但是我们选择再次在仓储层（repository level）中处理：

```

private class MyDeleteHandler : DeleteRequestHandler<MyRow>
{
    protected override void OnBeforeDelete()
    {
        base.OnBeforeDelete();

        var mc = Entities.MovieCastRow.Fields;
        foreach (var detailID in Connection.Query<Int32>(
            new SqlQuery()
                .From(mc)
                .Select(mc.MovieCastId)
                .Where(mc.MovieId == Row.MovieId.Value)))
        {
            new DeleteRequestHandler<Entities.MovieCastRow>().Process(this.UnitOfWork,
                new DeleteRequest
                {
                    EntityId = detailID
                });
        }
    }
}

```

我们实现这个主/从处理的方式不是很直观，并且在存储层包含了几个手工步骤。请继续阅读，看我们如何通过使用一个集成的功能 (*MasterDetailRelationAttribute*) 轻松地实现同一逻辑。

## 在行为 (**Behavior**) 中处理 保存/检索/删除

主/从关系是一个综合性的功能（至少在服务器端是），所以我使用 *MasterDetailRelation* 特性替代手工重写 保存/检索和删除操作。

打开 MovieRow.cs 并修改 *CastList* 属性：

```
[MasterDetailRelation(foreignKey: "MovieId", IncludeColumns = "PersonFullname")]
[DisplayName("Cast List"), SetFieldFlags(FieldFlags.ClientSide)]
public List<MovieCastRow> CastList
{
    get { return Fields.CastList[this]; }
    set { Fields.CastList[this] = value; }
}
```

我们指定该字段是主/从关系的详细列表（从表内容），并且详细列表的主 ID 字段（外键）是 *MovieId*。

现在我们撤消在 MovieRepository.cs 中做的所有更改：

```
private class MySaveHandler : SaveRequestHandler<MyRow> { }
private class MyDeleteHandler : DeleteRequestHandler<MyRow> { }
private class MyRetrieveHandler : RetrieveRequestHandler<MyRow>
{ }
```

在我们 *MasterDetailRelation* 特性中，我们指定了一个额外的属性 *IncludeColumns*：

```
[MasterDetailRelation(foreignKey: "MovieId", IncludeColumns = "PersonFullname")]
```

这是确保在检索演员列表时包含 *PersonFullname* 字段。另外，只默认选中表字段的情况下它将不会加载。当你打开一个存在演员信息的影片对话框时，演员的全名将为空。

确保你在网格列中使用到的任何可视字段都添加到 *IncludeColumns*。使用逗号分隔多个字段名称，例如 *IncludeColumns* = "FieldA, FieldB, FieldC"。

现在生成项目，你将看到使用更少的代码完成了同样的工作。

*MasterDetailRelationAttribute* 自动触发一种深层次行为（behavior），*MasterDetailRelationBehavior* 拦截检索/保存/删除处理并执行我们之前已经重写的方法及其他类似的操作。

所以我们做了同样的事情，但这一次，是以声明的方式，而不是命令式（告诉程序应该做什么，而不是如何去做）。

[https://en.wikipedia.org/wiki/Declarative\\_programming](https://en.wikipedia.org/wiki/Declarative_programming)

我们将在下面的章节介绍如何编写你自己的请求处理行为（behaviors）。

## 在人员对话框列出参演影片

为了显示人员参演的影片，我们将在 `PersonDialog` 添加一个选项卡。

默认情况下，所有编辑对话框（都继承自 `EntityDialog`）在 `MovieTutorial.Web/Views/Templates/EntityDialog.Template.html` 使用 `EntityDialog` 模板。

```
<div class="s-DialogContent">
    <div id="~_Toolbar" class="s-DialogToolbar">
        </div>
    <div class="s-Form">
        <form id="~_Form" action="">
            <div class="fieldset ui-widget ui-widget-content ui-corner-all">
                <div id="~_PropertyGrid"></div>
                <div class="clear"></div>
            </div>
        </form>
    </div>
</div>
```

该模板包含占位符，如 工具栏(`~_Toolbar`)、表单(`~_Form`) 和 网格属性(`~_PropertyGrid`)\*。

`~_` 是一个在运行时被替换为唯一对话框 ID 的特殊前缀。这确保了同一个对话框的两个实例不会有同样的 ID 值。

所有模板共享 `EntityDialog` 模板，因此我们把它原样添加到 `PersonDialog` 选项卡。

### 为 `PersonDialog` 添加选项卡模板定义

在 `Modules/MovieDB/Person/` 文件夹下创建 `MovieDB.PersonDialog.Template.html`，其内容为：

```
<div id="~_Tabs" class="s-DialogContent">
    <ul>
        <li><a href="#~_TabInfo"><span>Person</span></a></li>
        <li><a href="#~_TabMovies"><span>Movies</span></a></li>
    </ul>
    <div id="~_TabInfo" class="tab-pane s-TabInfo">
        <div id="~_Toolbar" class="s-DialogToolbar">
        </div>
        <div class="s-Form">
            <form id="~_Form" action="">
                <div class="fieldset ui-widget ui-widget-content ui-corner-all">
                    <div id="~_PropertyGrid"></div>
                    <div class="clear"></div>
                </div>
            </form>
        </div>
    </div>
    <div id="~_TabMovies" class="tab-pane s-TabMovies">
        <div id="~_MoviesGrid">
            </div>
    </div>
</div>
```

我们这里使用的语法是特定于 jQuery UI 选项卡组件，它需要一个包含指向选项卡面板 `div(.tab-pane)` 的 `UL` 列表。

当 `EntityDialog` 在模板中找到 ID 含有 `~_Tabs` 的 `div`，它将自动在模板中把其初始化为选项卡组件。

模板文件的命名是很重要的，它必须以 `.Template.html` 扩展名结尾。所有以该扩展名结束的文件都是通过动态脚本提供给客户端。

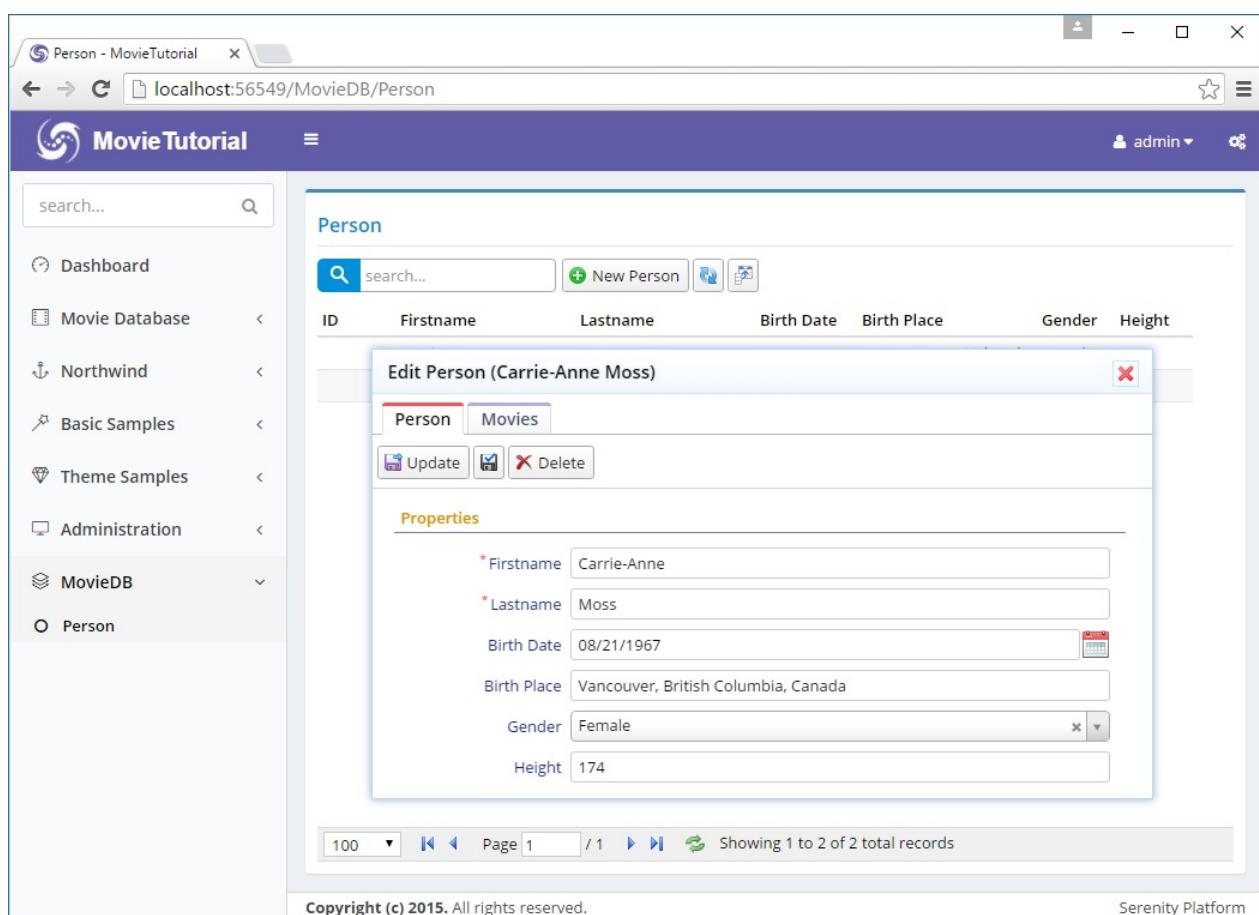
模板文件的文件夹被忽略，但是模板必须在 `Modules` 或者 `Views/Template` 文件夹下面。

默认情况下，所有的模板组件（`EntityDialog` 也继承自 `TemplatedWidget` 类）使用类名查找模板。因此，`PersonDialog` 首先查找名为 `MovieDB.PersonDialog.Template.html` 的模板，然后是 `PersonDialog.Template.html`。

MovieDB 来自于把 `PersonDialog` 命名空间删除根命名空间(MovieTutorial)。你也可以认为是模块名与类名的组合。

如果没有找到含该类名的模板，将继续到基类查找并最终找到一个备用模板，使用 `EntityDialog.Template.html`。

现在，我们将在 `PersonDialog` 中有一个选项卡：



与此同时，我注意到 `Person` 菜单还在 `MovieDB` 下面，并且我们忘了删除 `MovieCast` 菜单，我现在就修正.....

## 创建 PersonMovieGrid

`Movie` 选项卡现在还是空的。我们需要在该选项卡里定义一个有合适列的网格列表。

首先，在 *PersonColumns.cs* 旁边的 *PersonMovieColumns.cs* 文件中，定义需要在网格列表使用的列：

```
namespace MovieTutorial.MovieDB.Columns
{
    using Serenity.ComponentModel;
    using System;

    [ColumnsScript("MovieDB.PersonMovie")]
    [BasedOnRow(typeof(Entities.MovieCastRow))]
    public class PersonMovieColumns
    {
        [Width(220)]
        public String MovieTitle { get; set; }
        [Width(100)]
        public Int32 MovieYear { get; set; }
        [Width(200)]
        public String Character { get; set; }
    }
}
```

然后，在 *PersonGrid.ts* 旁边的 *PersonMovieGrid.ts* 文件中定义一个 *PersonMovieGrid* 类：

```
namespace MovieTutorial.MovieDB {  
  
    @Serenity.Decorators.registerClass()  
    export class PersonMovieGrid extends Serenity.EntityGrid<Mov  
ieCastRow, any>  
    {  
        protected getColumnsKey() { return "MovieDB.PersonMovie"  
; }  
        protected getIdProperty() { return MovieCastRow.idProper  
ty; }  
        protected getLocalTextPrefix() { return MovieCastRow.loc  
alTextPrefix; }  
        protected getService() { return MovieCastService.baseUrl  
; }  
  
        constructor(container: JQuery) {  
            super(container);  
        }  
    }  
}
```

实际上，我们使用 MovieCast 服务查找人员所参演的影片列表。

最后的步骤是在 PersonDialog.ts 实例化该网格列表：

```

@Serenity.Decorators.registerClass()
@Serenity.Decorators.responsive()
export class PersonDialog extends Serenity.EntityDialog<PersonRo
w, any> {
    protected getFormKey() { return PersonForm.formKey; }
    protected getIdProperty() { return PersonRow.idProperty; }
    protected getLocalTextPrefix() { return PersonRow.localTextP
refix; }
    protected getNameProperty() { return PersonRow.nameProperty;
}
    protected getService() { return PersonService.baseUrl; }

    protected form = new PersonForm(this.idPrefix);

    private moviesGrid: PersonMovieGrid;

    constructor() {
        super();

        this.moviesGrid = new PersonMovieGrid(this.byId("MoviesG
rid"));
        this.tabs.on('tabsactivate', (e, i) => {
            this.arrange();
        });
    }
}

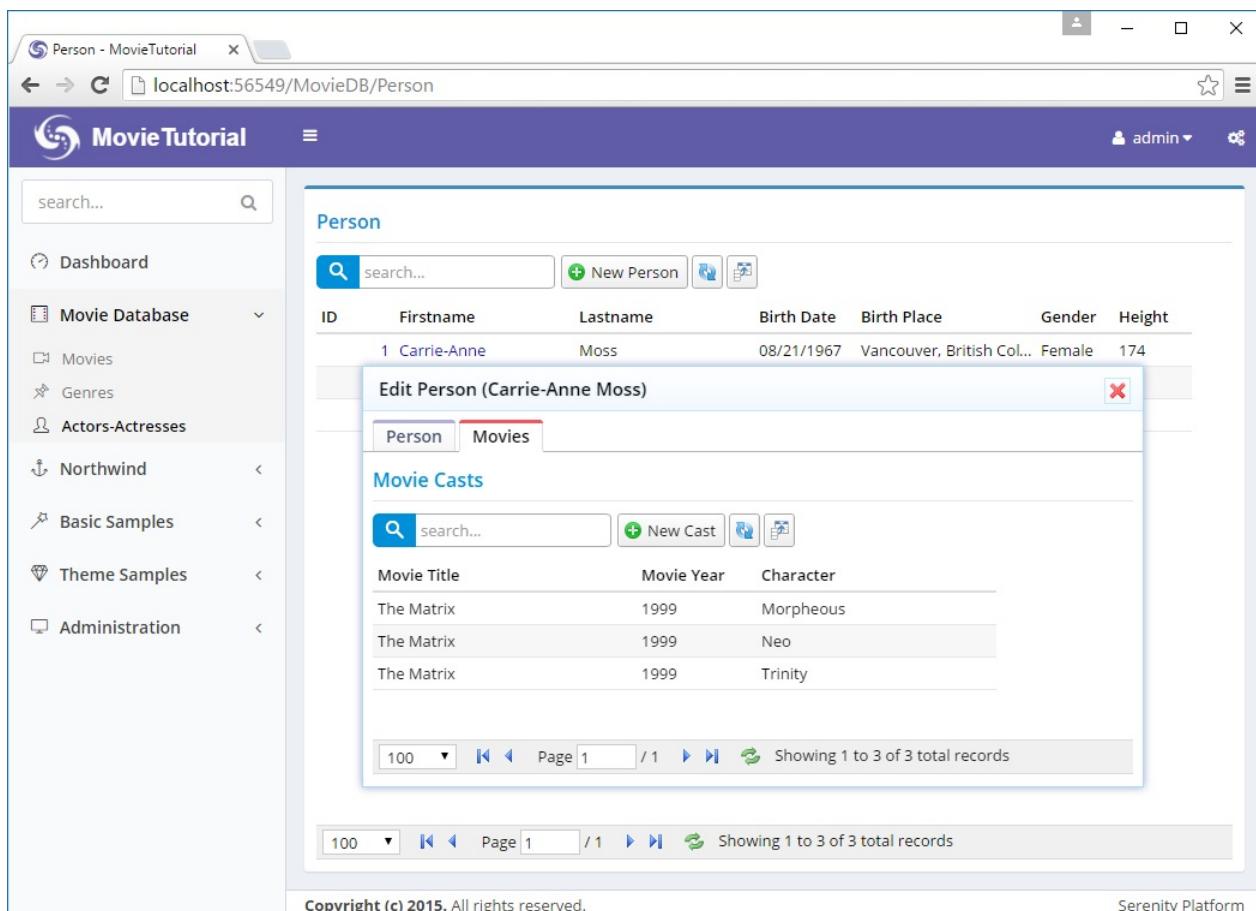
```

记得我们的模板在 movies 选项卡下有一个 id 为 `~_MoviesGrid` 的 div，我们在该 div 中创建了 PersonMovie 网格列表。

`this.ById("MoviesGrid")` 是一个特殊的模板组件方法。`$('#MoviesGrid')` 在这里不能工作，因为实际上 div 有一些像 `PersonDialog17_MoviesGrid` 的 ID。 `~_` 在模板中被替换为容器组件的唯一 ID。

我们还为 jQuery UI 选项卡附加 OnActivate 事件，并调用对话框的 Arrange 方法。这是解决 SlickGrid 初始创建在不可见的选项卡的问题，通过 Arrange 触发 SlickGrid 的布局来解决该问题。

OK，现在我们可以在 Movies 选项卡中看到影片列表，但是有些奇怪：



## 筛选出人员所参演的影片

不，Carrie-Anne Moss 并没有扮演以上三个角色。由于我们还没有告诉该应用什么过滤器，现在这个网格列表显示所有的影片演员记录。

PersonMovieGrid 应该知道它该显示哪个人员的电影记录。因此我们添加一个 *PersonID* 属性到该网格列表。该 *PersonID* 应该通过某种方式传递给列表服务进行过滤。

```
namespace MovieTutorial.MovieDB
{
    @Serenity.Decorators.registerClass()
    export class PersonMovieGrid extends Serenity.EntityGrid<MovieCastRow, any>
    {
        protected getColumnsKey() { return "MovieDB.PersonMovie" }
        protected getIdProperty() { return MovieCastRow.idProperty; }
    }
}
```

```
protected getLocalTextPrefix() { return MovieCastRow.localTextPrefix; }
protected getService() { return MovieCastService.baseUrl; }

constructor(container: JQuery) {
    super(container);
}

protected getButtons() {
    return null;
}

protected getInitialTitle() {
    return null;
}

protected usePager() {
    return false;
}

protected getGridCanLoad() {
    return this.personID != null;
}

private _personID: number;

get personID() {
    return this._personID;
}

set personID(value: number) {
    if (this._personID != value) {
        this._personID = value;
        this.setEquality(MovieCastRow.Fields.PersonId, value);
        this.refresh();
    }
}
```

```
}
```

我们使用 ES5 (EcmaScript 5) 的属性(get/set) 特性。它与 C# 的属性非常相似。

我们把人员 ID 存储在一个私有变量中。当它改变时，我们还使用 `SetEquality` 方法（将被发送到列表服务）为 `PersonId` 字段设置一个相等过滤器，并刷新获取更新。

相等过滤器是使用快速过滤器项目的列表请求参数。

当网格列表调用列表服务时，重写 `GetGridCanLoad` 方法允许我们做一些控制。如果我们没有重写它，当创建一个新的人员时，由于还没有 `PersonID`（它为 `null`），网格列表将加载所有的演员记录。

如果相等过滤器的参数为 `null`，List handler 将忽略相等过滤器的参数，就像快速过滤器下拉列表为空时，所有的记录都被显示出来。

通过重写三个方法，我们还做了三个改变：一、从工具栏（`getButtons`）中删除所有按钮；二、从网格列表（`getInitialTitle`）删除作为选项卡的标题，三、删除分页功能（`usePager`），一个人不可能参演一百万部电影吧？。

## 在 `PersonDialog` 设置 `PersonMovieGrid` 的 `PersonID`

如果没有设置网格列表的 `PersonID` 属性，它将总是为 `null`，并且不会加载记录。我们应该在 `Person` 对话框的 `afterLoadEntity` 方法中设置它。

```
namespace MovieTutorial.MovieDB
{
    // ...
    export class PersonDialog extends Serenity.EntityDialog<Person>
{
    onRow>
    {
        // ...
        protected afterLoadEntity()
        {
            super.afterLoadEntity();

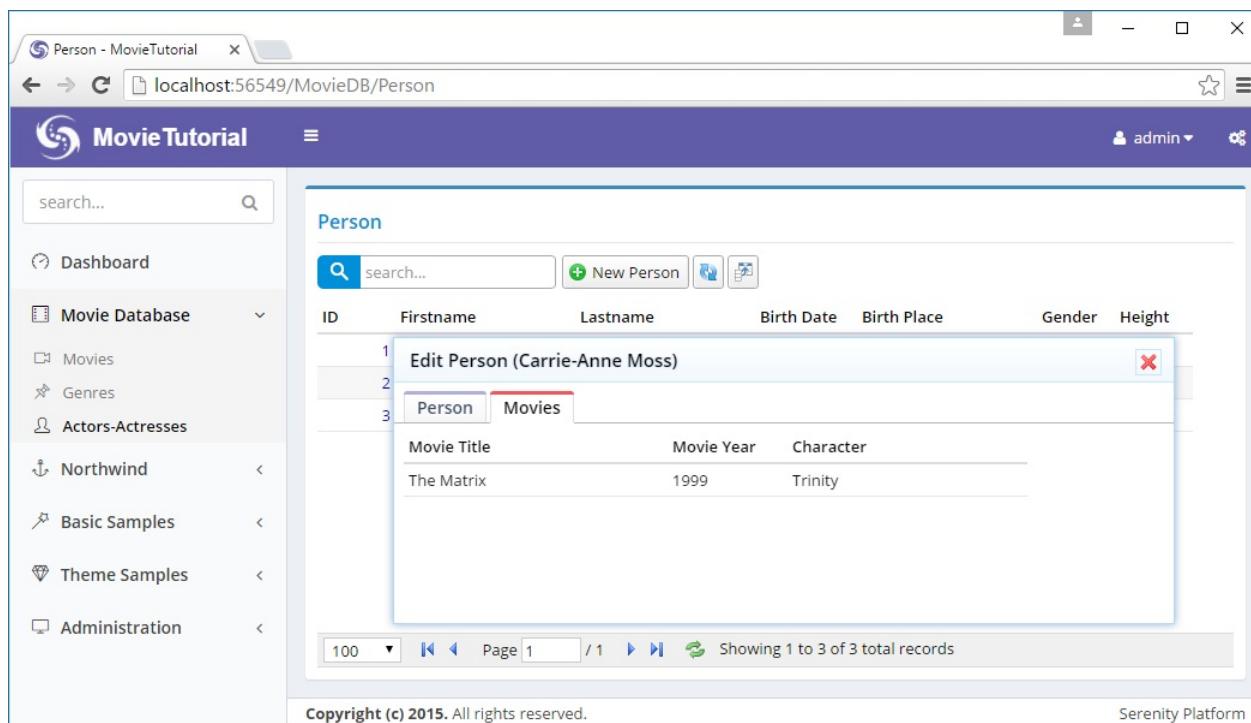
            moviesGrid.personID = this.entityId;
        }
    }
}
```

在实体或新实体被加载到对话框之后调用 *afterLoadEntity*。

请注意，实体是在后一阶段被加载的，因此把它放在对话框的构造函数中是没有用的。

*this.EntityId* 引用当前加载实体的标识值。在新记录的情况下，它的值为 *null*。

在对话框生命周期内，*AfterLoadEntity* 和 *LoadEntity* 可能被调用多次，因此避免在这些事件中创建一些子对象，否则你会有多个创建对象的实例。这就是为什么我们在对话框构造函数中创建网格列表的原因。



## 调整 **Movies** 选项卡大小

你可能已经注意到，当你切换到 **Movies** 选项卡时，对话框高度有点不够。这是由于对话框设置为自动高度，并且网格列表默认高度是 200px。当你切换到 **Movies** 选项卡，表单被隐藏，因此对话框调整为 **Movies** 网格列表高度。

在 site.less 编辑 **s-MovieDB-PersonDialog** css :

```
.s-MovieDB-PersonDialog {
    > .size { width: 650px; }
    .caption { width: 150px; }
    .s-PersonMovieGrid > .grid-container { height: 287px; }
}
```

# 添加海报（Primary）和简介（Gallery）图片

需要分别给人员（Person）和影片（Movie）记录添加一张海报和多张简介图片。  
让我们从迁移类开始：

```
using FluentMigrator;

namespace MovieTutorial.Migrations.DefaultDB
{
    [Migration(20160603205900)]
    public class DefaultDB_20160603_205900_PersonMovieImages : Migration
    {
        public override void Up()
        {
            Alter.Table("Person").InSchema("mov")
                .AddColumn("PrimaryImage").AsString(100).Nullable()
                .AddColumn("GalleryImages").AsString(int.MaxValue).Nullable();
        }

        Alter.Table("Movie").InSchema("mov")
            .AddColumn("PrimaryImage").AsString(100).Nullable()
            .AddColumn("GalleryImages").AsString(int.MaxValue).Nullable();
    }

    public override void Down()
    {
    }
}
```

然后修改 MovieRow.cs 和 PersonRow.cs：

```
namespace MovieTutorial.MovieDB.Entities
{
    // ...
    public sealed class PersonRow : Row, IIdRow, INameRow
    {

        [DisplayName("Primary Image"), Size(100),
         ImageUploadEditor(FilenameFormat = "Person/PrimaryImage
~/")]
        public string PrimaryImage
        {
            get { return Fields.PrimaryImage[this]; }
            set { Fields.PrimaryImage[this] = value; }
        }

        [DisplayName("Gallery Images"),
         MultipleImageUploadEditor(FilenameFormat = "Person/Gall
eryImages/~")]
        public string GalleryImages
        {
            get { return Fields.GalleryImages[this]; }
            set { Fields.GalleryImages[this] = value; }
        }

        // ...

        public class RowFields : RowFieldsBase
        {
            // ...
            public readonly StringField PrimaryImage;
            public readonly StringField GalleryImages;
            // ...
        }
    }
}
```

```
namespace MovieTutorial.MovieDB.Entities
{
    // ...
    public sealed class MovieRow : Row, IIdRow, INameRow
    {
        [DisplayName("Primary Image"), Size(100),
         ImageUploadEditor(FilenameFormat = "Movie/PrimaryImage/
~")]
        public string PrimaryImage
        {
            get { return Fields.PrimaryImage[this]; }
            set { Fields.PrimaryImage[this] = value; }
        }

        [DisplayName("Gallery Images"),
         MultipleImageUploadEditor(FilenameFormat = "Movie/Galle
ryImages/~")]
        public string GalleryImages
        {
            get { return Fields.GalleryImages[this]; }
            set { Fields.GalleryImages[this] = value; }
        }

        // ...
        public class RowFields : RowFieldsBase
        {
            // ...
            public readonly StringField PrimaryImage;
            public readonly StringField GalleryImages;
            // ...
        }
    }
}
```

我们指定这些字段将由 *ImageUploadEditor* 和 *MultipleImageUploadEditor* 类型处理。

*FilenameFormat* 指定上传文件的命名方式。例如，人员的海报图片将上传到 *App\_Data/upload/Person/PrimaryImage/* 目录下面。

你可以通过 web.config 文件的 appSettings 节点下的 *UploadSettings* 配置，把保存上传文件的根目录 (*App\_Data/upload*) 修改为任意文件目录。

~ **FilenameFormat** 的末尾是自动命名方案的快捷方式

{1:00000}/{0:00000000}\_{2} 。

在这里，参数 {0} 替换为记录的标识，如，PersonID。

参数 {1} 是 1000 以内的整数标识。这对在一个目录中的保存文件数进行限制是很有帮助的。

参数 {2} 是一个唯一的字符串，例如 6l55nk6v2tiyi，它用于为每个上传文件生成一个新的文件名，这有助于避免客户端缓存所造成的问题。

它还提供了一些安全措施，因此在没有链接的情况下不能知道文件名称。

因此，我们把上传的人员海报图片文件存放在如下路径：

```
> App_Data\upload\Person\PrimaryImage\00000\00000001_6l55nk6v2tiyi.jpg
```

你不必遵守此命名方案，也可以指定你自己的格式，如

PersonPrimaryImage\_{0}\_{2} 。

下一步，把这些字段添加到表单（MovieForm.cs 和 PersonForm.cs）：

```
namespace MovieTutorial.MovieDB.Forms
{
    //...
    public class PersonForm
    {
        public String Firstname { get; set; }
        public String Lastname { get; set; }
        public String PrimaryImage { get; set; }
        public String GalleryImages { get; set; }
        public DateTime BirthDate { get; set; }
        public String BirthPlace { get; set; }
        public Gender Gender { get; set; }
        public Int32 Height { get; set; }
    }
}
```

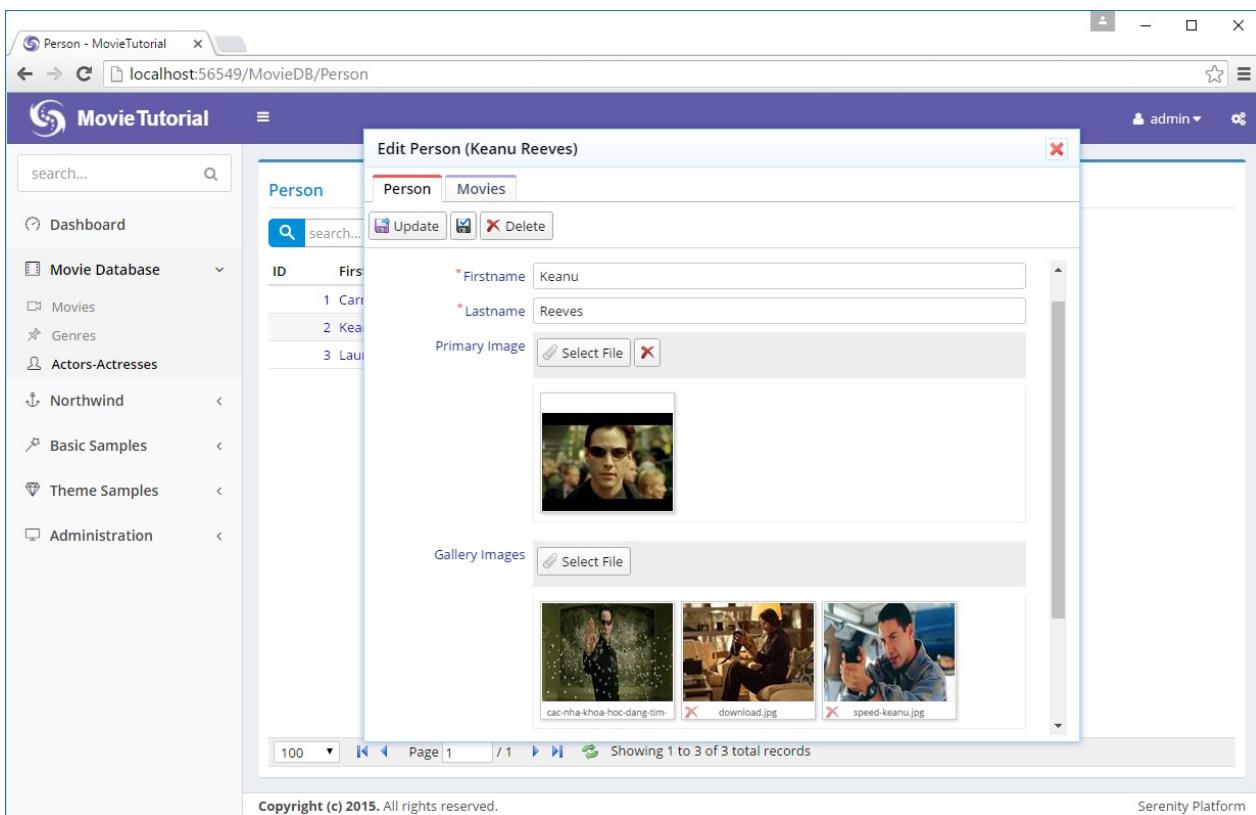
```
namespace MovieTutorial.MovieDB.Forms
{
    //...
    public class MovieForm
    {
        public String Title { get; set; }
        [TextAreaEditor(Rows = 3)]
        public String Description { get; set; }
        [MovieCastEditor]
        public List<Entities.MovieCastRow> CastList { get; set; }

        public String PrimaryImage { get; set; }
        public String GalleryImages { get; set; }
        [TextAreaEditor(Rows = 8)]
        public String Storyline { get; set; }
        public Int32 Year { get; set; }
        public DateTime ReleaseDate { get; set; }
        public Int32 Runtime { get; set; }
        public Int32 GenreId { get; set; }
        public MovieKind Kind { get; set; }
    }
}
```

我还修改了人员对话框的 css，让其有更多的空间：

```
.s-MovieDB-PersonDialog {
    > .size { width: 700px; height: 600px; }
    .caption { width: 150px; }
    .s-PersonMovieGrid > .grid-container { height: 500px; }
}
```

这就是我们现在的窗体：



ImageUploadEditor 直接在一个字符串字段中存储文件名，而  
MultipleImageUpload 编辑器将文件名使用 JSON 数组格式存储在字符串字段  
中。

## 删除 Northwind 和 其他 示例

我认为我们的项目达到了良好的状态，现在我要从 MovieTutorial 项目中删除 Northwind 和其他示例。

请参见如下帮助主题：

[如何删除 Northwind 和 其他 示例](#)

# 多租户

在本教程中，我们把 Norhwind 改为多租户应用程序。

这是多租户软件在维基百科中的定义：

多租户技术（multi-tenancy technology）或称多重租赁技术，是一种软件架构技术，它是在探讨与实现如何于多用户的环境下共用相同的系统或程序组件，并且仍可确保各用户间数据的隔离性。---维基百科

我们在每张表中添加一个 `TenantId` 字段，包含用户（`Users`）表，并只让用户对看到/修改只属于自己的租户记录。因此，如果租户使用自己的数据库工作，租户间是相互独立的。

多租户应用程序有一些优势，如可以减少管理的花销。但是它们也有一些劣势，例如，由于所有租户的数据都保存在一个数据库中，单个租户不能简单地独自获取或者备份自己的数据。当有更多的记录需要处理时通常会降低性能。

随着云应用的趋势，虚拟化的成本逐渐降低，并包含迁移之类的功能，现在很容易在云上安装多实例应用程序。

我个人是避免使用多租户应用程序的。在我看来每个客户最好有一个数据库。

但是，有一些用户询问如何实现多租户的功能，于是便有了该教程，但作为奖励，随着多租户教程，我们将介绍一些高级的 Serenity 主题。

你可以在下面的连接中找到本教程的源代码：

<https://github.com/volkanceylan/Serenity-Tutorials/tree/master/MultiTenancy>

## 创建一个名为 **MultiTenancy** 的教程

在 Visual Studio 中，点击 文件 -> 新项目。确保你选择了 `Serene template`，输入 `MultiTenancy` 作为名称并点击 `OK`。

在解决方案资源管理器中，你应该看到两个项目，分别是 `MultiTenancy.Web` 和 `MultiTenancy.Script`。

确保 *MultiTenancy.Web* 是启动项目（它应该加粗显示），如果它还不是启动项目，你可以在该项目名称中右键并点击 设为启动项目。

## 添加项目依赖

默认情况下，当你按 F5 运行程序时，Visual Studio 只生成 *MultiTenancy.Web* 项目。

这是由 Visual Studio 的 Options -> Projects and Solutions -> Build And Run -> "Only build startup projects and dependencies on Run" 设置控制的，但并不建议你修改该设置。

为了在 Web 项目运行时，也让 Script 项目生成，右键 *MultiTenancy.Web* 项目，点击 *Build Dependencies -> Project Dependencies* 并勾选 *Dependencies* 选项卡下的 *MultiTenancy.Script*。

不幸的是，我们没办法在 Serene 的模板设置该依赖关系。

## 添加租户（Tenants）表和 TenantId 字段

为了租户间相互独立，我们需要把 TenantId 字段添加到所有表中。

因此，我们先添加一个租户（Tenants）表。

因为 Northwind 表已经有记录，我们将定义一个 ID 为 1 的主租户，并把所有现有记录的 TenantId 设为该值。

现在是时候写迁移类，实际上有两个迁移类：一个是 Northwind，另一个是 Default 数据库。

**DefaultDB\_20160110\_092200\_MultiTenant.cs:**

```
using FluentMigrator;

namespace MultiTenancy.Migrations.DefaultDB
{
    [Migration("20160110092200")]
    public class DefaultDB_20160110_092200_MultiTenant
        : AutoReversingMigration
    {
        public override void Up()
        {
            Create.Table("Tenants")
                .WithColumn("TenantId").AsInt32()
                .Identity().PrimaryKey().NotNullable()
                .WithColumn("TenantName").AsString(100)
                .NotNullable();

            Insert.IntoTable("Tenants")
                .Row(new
                {
                    TenantName = "Primary Tenant"
                });

            Insert.IntoTable("Tenants")
                .Row(new
                {

```

```

        TenantName = "Second Tenant"
    });

    Insert.IntoTable("Tenants")
        .Row(new
    {
        TenantName = "Third Tenant"
    });

    Alter.Table("Users")
        .AddColumn("TenantId").AsInt32()
        .NotNullable().WithDefaultValue(1);

    Alter.Table("Roles")
        .AddColumn("TenantId").AsInt32()
        .NotNullable().WithDefaultValue(1);

    Alter.Table("Languages")
        .AddColumn("TenantId").AsInt32()
        .NotNullable().WithDefaultValue(1);
}

}
}

```

我已经在用户（user）表所在的 Default 数据库创建租户（Tenants）表，并在该表添加 3 个预定义的租户。实际上我们只需要 ID 为 1 的第一个租户。

我们没有在一些表（如 UserPermissions、UserRoles、RolePermissions 等）添加 TenantId 列，因为他们可以通过 UserId 或 RoleId 获取 TenantId 信息。

### NorthwindDB\_20160110\_093500\_MultiTenant.cs:

```

using FluentMigrator;

namespace MultiTenancy.Migrations.NorthwindDB
{
    [Migration(20160110093500)]
    public class NorthwindDB_20160110_093500_MultiTenant
        : AutoReversingMigration
    {

```

```
public override void Up()
{
    Alter.Table("Employees")
        .AddColumn("TenantId").AsInt32()
        .NotNullable().WithDefaultValue(1);

    Alter.Table("Categories")
        .AddColumn("TenantId").AsInt32()
        .NotNullable().WithDefaultValue(1);

    Alter.Table("Customers")
        .AddColumn("TenantId").AsInt32()
        .NotNullable().WithDefaultValue(1);

    Alter.Table("Shippers")
        .AddColumn("TenantId").AsInt32()
        .NotNullable().WithDefaultValue(1);

    Alter.Table("Suppliers")
        .AddColumn("TenantId").AsInt32()
        .NotNullable().WithDefaultValue(1);

    Alter.Table("Orders")
        .AddColumn("TenantId").AsInt32()
        .NotNullable().WithDefaultValue(1);

    Alter.Table("Products")
        .AddColumn("TenantId").AsInt32()
        .NotNullable().WithDefaultValue(1);

    Alter.Table("Region")
        .AddColumn("TenantId").AsInt32()
        .NotNullable().WithDefaultValue(1);

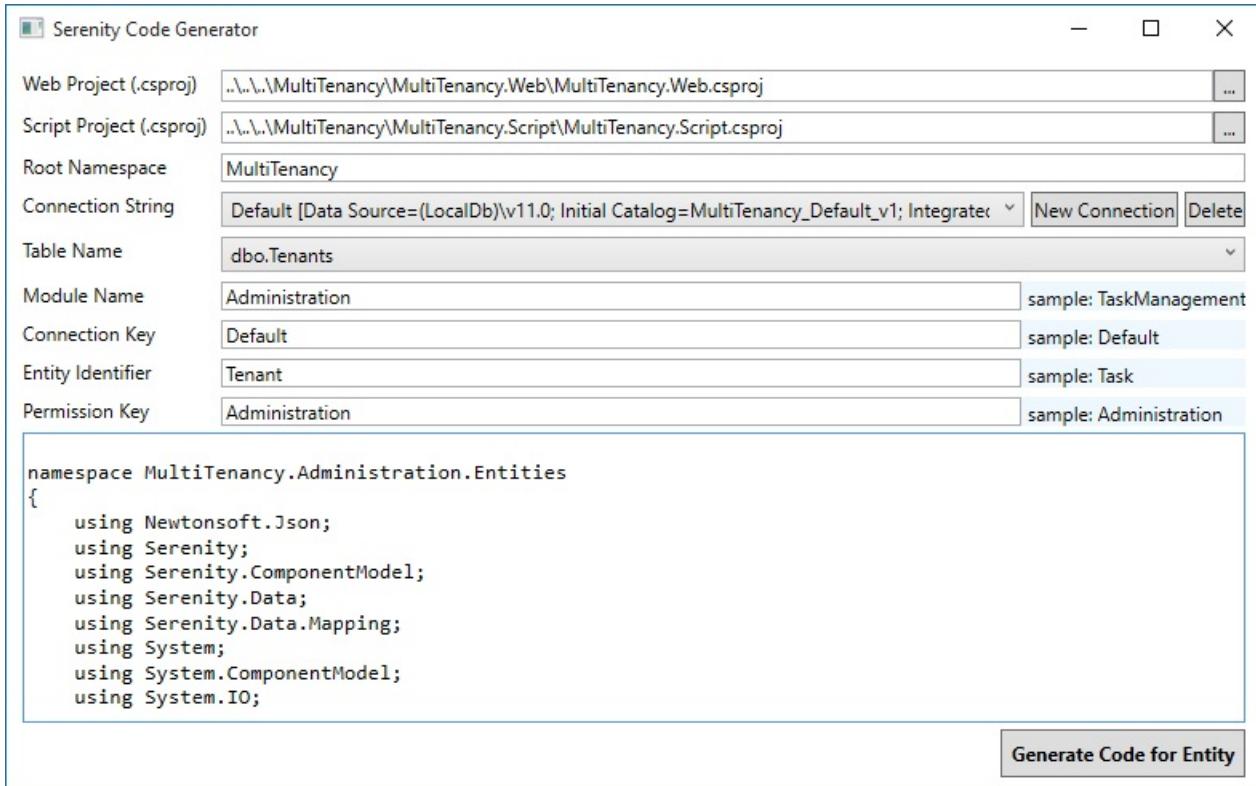
    Alter.Table("Territories")
        .AddColumn("TenantId").AsInt32()
        .NotNullable().WithDefaultValue(1);
}

}
```



# 为租户（Tenants）生成代码

启动 *Sergen*，并使用 *Default* 连接为租户（Tenants）表生成代码。



然后，我们在 *TenantRow* 定义检索脚本，并把 *InstanceName* 属性设置为 *Tenant*：

```
namespace MultiTenancy.Administration.Entities
{
    //...
    [ConnectionKey("Default"), DisplayName("Tenants"),
     InstanceName("Tenant"), TwoLevelCached]
    [LookupScript("Administration.Tenant")]
    public sealed class TenantRow : Row, IIdRow, INameRow
    {
        [DisplayName("Tenant Id"), Identity]
        public Int32? TenantId
        {
            get { return Fields.TenantId[this]; }
            set { Fields.TenantId[this] = value; }
        }

        //...
    }
}
```

让我们定义一个只有 *admin* 用户有的 Administration:Tenants 权限：

```
namespace MultiTenancy.Administration
{
    public class PermissionKeys
    {
        public const string Security = "Administration:Security";
        ;
        public const string Translation = "Administration:Translation";
        public const string Tenants = "Administration:Tenants";
    }
}
```

并把它设置到 TenantRow：

```
[ConnectionKey("Default"), DisplayName("Tenants"), InstanceName(  
    "Tenant"), TwoLevelCached]  
[ReadPermission(PermissionKeys.Tenants)]  
[ModifyPermission(PermissionKeys.Tenants)]  
[LookupScript("Administration.Tenant")]  
public sealed class TenantRow : Row, IIdRow, INameRow  
{
```

## 在用户对话框中选择租户

我们在 *Users* 表中添加一个 *TenantId* 字段，但是没有在 *UserRow* 中定义，并且也不能在用户对话框中看到该字段。

该字段只能被 *admin* 用户查看和编辑。即使是授予了访问管理租户权限的其他用户也不能查看或者修改这些信息。

首先把 *TenantId* 字段添加到 *UserRow.cs*：

```
namespace MultiTenancy.Administration.Entities
{
    //...
    public sealed class UserRow : LoggingRow, IIdRow, INameRow
    {
        //...
        [DisplayName("Last Directory Update"), Insertable(false)
        , Updatable(false)]
        public DateTime? LastDirectoryUpdate
        {
            get { return Fields.LastDirectoryUpdate[this]; }
            set { Fields.LastDirectoryUpdate[this] = value; }
        }

        [DisplayName("Tenant"), ForeignKey("Tenants", "TenantId")
        , LeftJoin("tnt")]
        [LookupEditor(typeof(TenantRow))]
        public Int32? TenantId
        {
            get { return Fields.TenantId[this]; }
            set { Fields.TenantId[this] = value; }
        }

        [DisplayName("Tenant"), Expression("tnt.TenantName")]
        public String TenantName
        {
            get { return Fields.TenantName[this]; }
            set { Fields.TenantName[this] = value; }
        }
}
```

```
}

//...

public class RowFields : LoggingRowFields
{
    //...
    public readonly DateTimeField LastDirectoryUpdate;
    public readonly Int32Field TenantId;
    public readonly StringField TenantName;
    //...

}

}

}
```

要编辑该字段，我们需要在 *UserForm.cs* 添加它：

```
namespace MultiTenancy.Administration.Forms
{
    using Serenity;
    using Serenity.ComponentModel;
    using System;
    using System.ComponentModel;

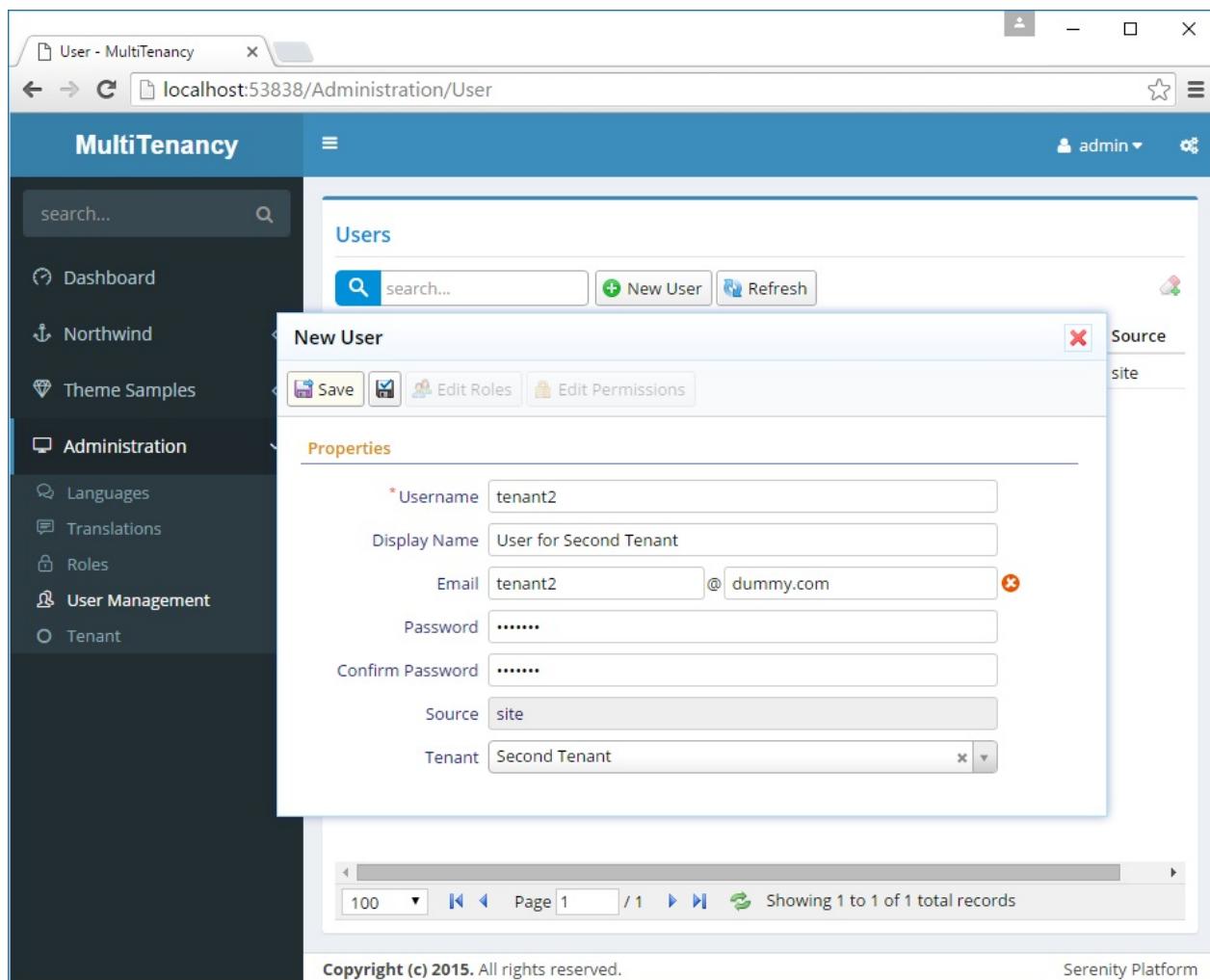
    [FormScript("Administration.User")]
    [BasedOnRow(typeof(Entities.UserRow))]
    public class UserForm
    {
        public String Username { get; set; }
        public String DisplayName { get; set; }
        [EmailEditor]
        public String Email { get; set; }
        [PasswordEditor]
        public String Password { get; set; }
        [PasswordEditor, OneWay]
        public String PasswordConfirm { get; set; }
        [OneWay]
        public string Source { get; set; }
        public Int32? TenantId { get; set; }
    }
}
```

同样需要增加用户对话框的大小，在 `site.administration.less` 中设置：

```
.s-UserDialog {
    > .size { .widthAndMin(650px); }
    .dialog-styles(@h: auto, @l: 150px, @e: 400px);
    .categories { height: 300px; }
}
```

现在，打开 *User Management* 页面并创建一个属于第二个租客的用户 *tenant2*。

在用户对话框中选择租户



创建此用户之后，编辑其权限并授予他用户、角色管理和权限，因为这将是我们的第二个租客的管理用户。

## 使用 **Tenant2** 登录

退出并使用 *tenant2* 身份登录系统。

当你打开 *User Management* 页面，你将看到该用户除了可以查看和编辑自己的 *tenant2* 用户，还可以查看和编辑 *admin* 用户。

在用户对话框中选择租户

The screenshot shows a web application interface for managing users across multiple tenants. The top navigation bar indicates the current tenant is 'tenant2'. The main menu on the left includes 'Dashboard', 'Theme Samples', 'Administration' (selected), and 'User Management'. The 'Administration' menu has sub-options like 'Roles' and 'User Management'. The 'User Management' section shows a list of users:

ID	Username	Display Name	Email	Source
1	admin	admin	admin@dummy.com	site
2	tenant2	User for Second Tena...	tenant2@dummy.com	site

A modal dialog titled 'Edit User (admin)' is open for the user with ID 1 (admin). The dialog contains fields for 'Username' (admin), 'Display Name' (admin), 'Email' (admin@dummy.com), 'Password' (empty), 'Confirm Password' (empty), 'Source' (site), and 'Tenant' (Primary Tenant). Action buttons include 'Update', 'Delete', 'Edit Roles', and 'Edit Permissions'.

这并不是我们所希望的。

让我们阻止他看到其他租户的用户。

# 使用 TenantId 筛选用户

我们首先需要在 `UserDefinition` 加载并缓存租户用户。

打开 `Multitenancy.Web/ Modules/ Administration/ User/ Authentication` 下的 `UserDefinition.cs`，并添加 `TenantId` 属性。

```
namespace MultiTenancy.Administration
{
    using Serenity;
    using System;

    [Serializable]
    public class UserDefinition : IUserDefinition
    {
        public string Id { get { return UserId.Invariant(); } }

        public string DisplayName { get; set; }
        public string Email { get; set; }
        public short IsActive { get; set; }
        public int UserId { get; set; }
        public string Username { get; set; }
        public string PasswordHash { get; set; }
        public string PasswordSalt { get; set; }
        public string Source { get; set; }
        public DateTime? UpdateDate { get; set; }
        public DateTime? LastDirectoryUpdate { get; set; }
        public int TenantId { get; set; }
    }
}
```

当你通过 `Authorization.UserDefinition` 请求当前用户时，该类被返回。

我们也在类被加载的地方修改代码。在同一文件夹编辑 `UserRetrieveService.cs`，并修改 `GetFirst`：

```

private UserDefinition GetFirst(IDbConnection connection, BaseCriteria criteria)
{
    var user = connection.TrySingle<Entities.UserRow>(criteria);
    if (user != null)
        return new UserDefinition
    {
        UserId = user.UserId.Value,
        Username = user.Username,
        Email = user.Email,
        DisplayName = user.DisplayName,
        IsActive = user.IsActive.Value,
        Source = user.Source,
        PasswordHash = user.PasswordHash,
        PasswordSalt = user.PasswordSalt,
        UpdateDate = user.UpdateDate,
        LastDirectoryUpdate = user.LastDirectoryUpdate,
        TenantId = user.TenantId.Value
    };
}

return null;
}

```

现在，是时候使用 *TenantId* 过滤并列出用户。打开 *UserRepository.cs*，定位到 *MyListHandler* 类做如下修改：

```

private class MyListHandler : ListRequestHandler<MyRow>
{
    protected override void ApplyFilters(SqlQuery query)
    {
        base.ApplyFilters(query);

        var user = (UserDefinition)Authorization.UserDefinition;
        if (!Authorization.HasPermission(PermissionKeys.Tenants))
        {
            query.Where(fld.TenantId == user.TenantId);
        }
    }
}

```

在这里，我们先获得当前登录用户的缓存用户定义。

我们检查该用户是否有租户管理权限，该权限只被授予给 *admin*。如果没有，我们使用 *TenantId* 过滤列表记录。

# 从用户窗体移除租户下拉列表

在你重新生成并启动项目后，现在的用户页面是这样的：

ID	Username	Display Name	Email	Source
2	tenant2	User for Second Tena...	tenant2@dummy.com	site

是的，再也不能看到 `admin` 用户，但是有些事不对劲。当你点击 `tenant2` 时，没有任何响应，却得到一个错误 “Can't load script data: `Lookup.Administration.Tenant`”。

该错误与我们在仓储层的最近一次过滤没有关系。它不能加载检索脚本，因为当前用户没有 `Tenants` 表的权限。但是最后怎么就看到该记录了？

可以看到该记录是由于我们之前是以 `admin` 身份登录并打开用户的编辑对话框，此时我们已经加载了这个检索脚本，并且浏览器对它进行缓存，因此当我们使用 `tenant2` 登录并打开编辑对话框时，它从浏览器缓存加载租户。

但这一次，因为我们重新生成项目，浏览器尝试从服务器加载它，由于 `tenant2` 不具有此权限，所以我们有了这个错误。It's ok，我们不想让他有此权限，但如何避免出现这个错误呢？

我们需要从用户窗体删除 `Tenant` 字段。但是 `admin` 用户需要这个字段，所以，我们不能简单地从 `UserForm.cs` 删除它。因此，我们需要有条件地做这事。

转换所有 T4 文件，然后打开 `MultiTutorial.Web/Modules/Administration/User/UserDialog.ts` 并重写 `getPropertyItems` 方法：

```
namespace Serene.Administration {

    @Serenity.Decorators.registerClass()
    export class UserDialog extends Serenity.EntityDialog<UserRow, any> {
        // ...

        protected getPropertyItems() {
            let items = super.getPropertyItems();
            if (!Authorization.hasPermission("Administration:Tenants"))
                items = items.filter(x => x.name != UserRow.Fields.TenantId);
            return items;
        }
    }
}
```

如果 Serenity < 2.0：转换所有 T4 文件，然后打开 *UserDialog.cs* 并重写 *GetPropertyItems* 方法：

```
namespace MultiTenancy.Administration
{
    using jQueryApi;
    using Serenity;
    using System.Collections.Generic;
    using System.Linq;

    //...
    public class UserDialog : EntityDialog<UserRow>
    {
        //...
        protected override List<PropertyItem> GetPropertyItems()
        {
            var items = base.GetPropertyItems();

            if (!Authorization.HasPermission("Administration:
Tenants"))
                items = items.Where(x =>
                    x.Name != UserRow.Fields.TenantId).ToList();

            return items;
        }
    }
}
```

*GetPropertyItems* 是对话框从服务器端表单定义获取表单字段的方法。从服务器端定义的 *UserForm* 中读取该字段。

如果没有租户的管理权限，我们在客户端的表单定义中移除 *TenantId* 字段。

只是在对话框实例中删除 *TenantId* 字段，实际上，并没有修改表单定义。

现在 tenant2 可以编辑自己的用户。

有些用户报告说，这也将删除 admin 用户的租户选择。请确保你 MultiTenancy.Script 项目中 Authorization.cs 文件的 HasPermission 方法是这样的：

```
public static bool HasPermission(string permissionKey)
{
    return
        UserDefinition.Username == "admin" ||
        UserDefinition.Permissions[permissionKey];
}
```

## 在服务端对租户选择进行安全检测

当使用 *tenant2* 身份登录并打开它的编辑窗体，没有显示选择 *Tenant* 下拉列表，因此就不能改变它的 *Tenant* 吗？

错！

如果是一个普通的用户，就不能改变 *Tenant*。但如果有一些 Serenity 及服务器工作原理的知识，就能修改 *Tenant*。

当你正在使用 web 时，你得更认真地对待安全。

非常容易就在 web 应用程序中创建安全漏洞，除非你在客户端和服务器端都进行验证处理。

我们来演示一下。当使用 *tenant2* 身份登录时，打开 Chrome 控制台。

复制下面的代码，并把它粘贴到控制台：

```
Q.serviceCall({
    service: 'Administration/User/Update',
    request: {
        EntityId: 2,
        Entity: {
            UserId: 2,
            TenantId: 1
        }
    }
});
```

现在刷新用户管理页面，你将看到 *tenant2* 现在可以看到 admin 用户。

我们使用 javascript 调用 *User Update* 服务，并把 *tenant2* 用户的 *TenantId* 修改为 1 (主租客)。

让我们把它还原回 第二承租人(2)，然后我们将修复这个安全漏洞：

```
Q.serviceCall({
    service: 'Administration/User/Update',
    request: {
        EntityId: 2,
        Entity: {
            UserId: 2,
            TenantId: 2
        }
    }
});
```

打开 *UserRepository.cs*，定位到 *MySaveHandler* 类，并修改它的 *GetEditableFields* 方法：

```
protected override void GetEditableFields(HashSet<Field> editable)
{
    base.GetEditableFields(editable);

    if (!Authorization.HasPermission(Administration.PermissionKeys.Security))
    {
        editable.Remove(fld.Source);
        editable.Remove(fld.IsActive);
    }

    if (!Authorization.HasPermission(Administration.PermissionKeys.Tenants))
    {
        editable.Remove(fld.TenantId);
    }
}
```

生成你的项目，然后尝试再次在控制台输入：

```
Q.serviceCall({
  service: 'Administration/User/Update',
  request: {
    EntityId: 2,
    Entity: {
      UserId: 2,
      TenantId: 1
    }
  }
});
```

你将得到这个错误：

```
Tenant field is read only!
```

SaveRequestHandler 调用 `GetEditableField` 方法来决定哪些字段是可编辑的，因此由用户更新。默认情况下，这些字段通过检索行（row）属性的 `Updatable` 和 `Insertable` 特性决定。

除非特别指定，所有字段是可插入和可更新的。

如果我们没有租户管理权限，我们从自动确定可编辑字段列表中删除 `TenantId`。

## 为新用户设置 **TenantId**

当我们使用 `Tenant2` 登录系统，并尝试创建一个新用户 `User2`。

你不会收到任何错误，但令人惊讶的是，你不能在列表中看到新创建的用户，`User2` 发生了什么？

由于我们在迁移类中，把 `TenantId` 的值设置为 1，现在 `User2` 的 `TenantId` 也是 1，并且它也是 主租客。

我们需要在已登录的用户中把新用户的 `TenantId` 设置为同样的值。

修改 `UserRepository` 的 `SetInternalFields` 方法：

```
protected override void SetInternalFields()
{
    base.SetInternalFields();

    if (IsCreate)
    {
        Row.Source = "site";
        Row.IsActive = Row.IsActive ?? 1;
        if (!Authorization.HasPermission(Administration.PermissionKeys.Tenants) ||
            Row.TenantId == null)
        {
            Row.TenantId = ((UserDefinition)Authorization.UserDefinition)
                .TenantId;
        }
    }

    if (IsCreate || !Row.Password.IsEmptyOrNull())
    {
        string salt = null;
        Row.PasswordHash = GenerateHash(password, ref salt);
        Row.PasswordSalt = salt;
    }
}
```

除非有租户管理权限，否则我们在这里把该 *TenantId* 设置为与当前用户相同的值。

现在尝试创建一个新用户 *User2b*，这次你将在列表中看到新建的用户。

## 防止编辑其他租户的用户

记住，使用 `tenant2` 可以在一些服务调用中更新自己的 `TenantId`，我们需要在服务器端对它进行安全检测。

与此类似，即使默认情况下，不能看到其他租户的用户，但实际上可以检索和更新他们。

又到了黑客时间。

打开 Chrome 的控制台，并输入：

```
new MultiTenancy.Administration.UserDialog().loadByIdAndOpenDialog(1)
```

什么？可以打开 Admin 的用户对话框并更新内容！

`MultiTenancy.Administration.UserDialog` 是当你在用户管理页面单击用户名时所打开的对话框类。

我们创建一个新的对话框实例，并且使用 ID 加载用户实体。Admin 用户的 ID 是 1。

因此，对话框调用 `UserRepository` 的 `Retrieve` 服务加载 ID 为 1 的实体。

记住，我们在 `UserRepository` 的 `List` 方法做过滤。所以，服务也不知道是否应该返回另一租户的记录。

此时，应该在 `UserRepository` 对 `retrieve` 服务做安全检测。

```
private class MyRetrieveHandler : RetrieveRequestHandler<MyRow>
{
    protected override void PrepareQuery(SqlQuery query)
    {
        base.PrepareQuery(query);

        var user = (UserDefinition)Authorization.UserDefinition;
        if (!Authorization.HasPermission(PermissionKeys.Tenants))
    )
        query.Where(fld.TenantId == user.TenantId);
    }
}
```

我们之前在 `MyListHandler` 中做了同样的修改。

如果你现在尝试运行同样的 Javascript 代码，你将得到一个错误：

```
Record not found. It might be deleted or you don't have required
permissions!
```

但是，我们依然可以手工调用 `Update` 服务更新记录。所以，也需要在 `MySaveHandler` 中做安全检测。

把 `ValidateRequest` 方法修改为：

```
protected override void ValidateRequest()
{
    base.ValidateRequest();

    if (IsUpdate)
    {
        var user = (UserDefinition)Authorization.UserDefinition;
        if (old.TenantId != user.TenantId)
            Authorization.ValidatePermission(PermissionKeys.Tena
nts);

        // ...
    }
}
```

我们在这里检测是否是更新，如果要更新记录的 `TenantId`(`Old.TenantId`) 与当前登录用户的 `TenantId` 不相等时，我们调用 `Authorization.ValidatePermission` 方法确保该用户有租户的管理权限，如果没有该权限，将抛出错误。

```
Authorization has been denied for this request!
```

## 防止删除其他租户的用户

在 `UserRepository` 中有删除和更新处理，并且他们有类似的安全漏洞。

使用类似的方法，我们也需要对它们进行安全检测：

```
private class MyDeleteHandler : DeleteRequestHandler<MyRow>
{
    protected override void ValidateRequest()
    {
        base.ValidateRequest();

        var user = (UserDefinition)Authorization.UserDefinition;
        if (Row.TenantId != user.TenantId)
            Authorization.ValidatePermission(PermissionKeys.Tenants);
    }
}

private class MyUndeleteHandler : UndeleteRequestHandler<MyRow>
{
    protected override void ValidateRequest()
    {
        base.ValidateRequest();

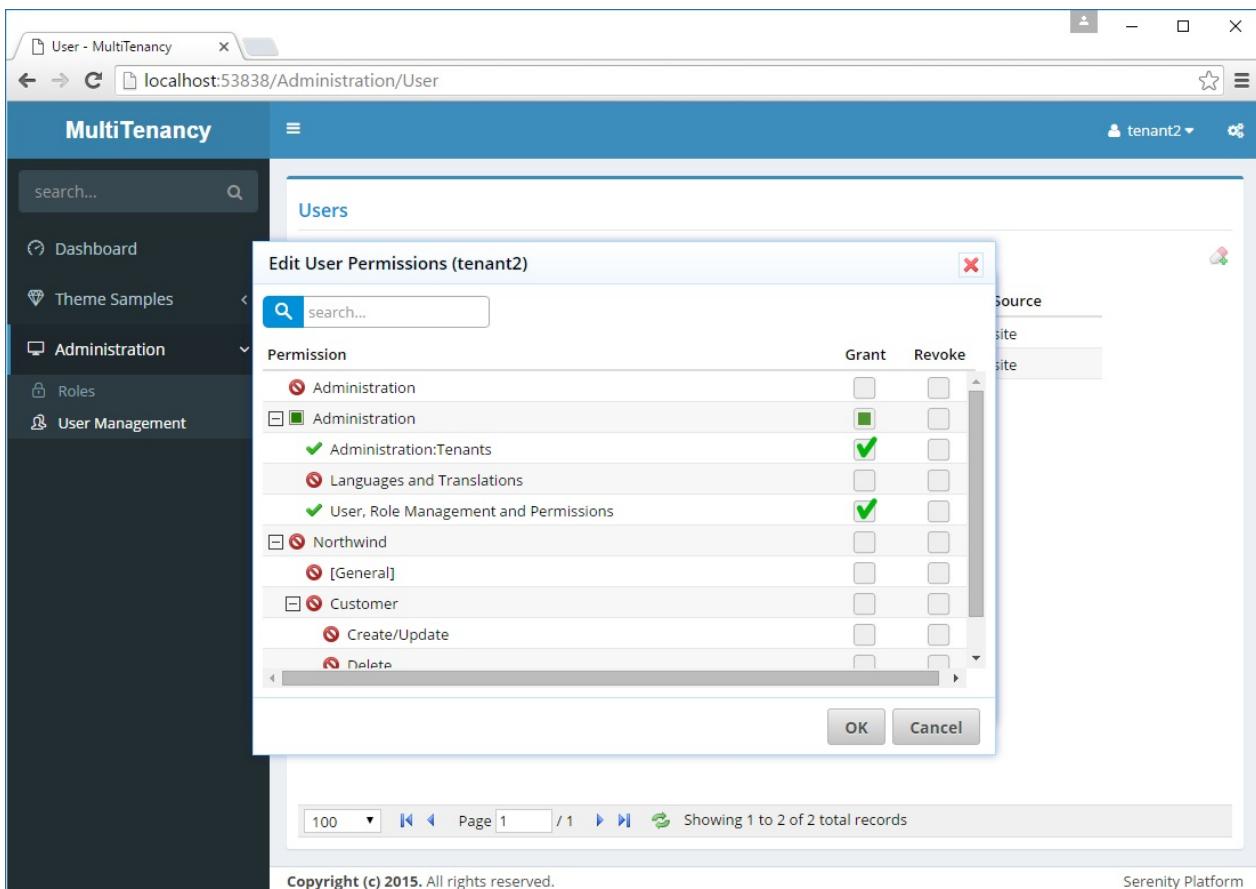
        var user = (UserDefinition)Authorization.UserDefinition;
        if (Row.TenantId != user.TenantId)
            Authorization.ValidatePermission(PermissionKeys.Tenants);
    }
}
```

防止编辑其他租户的用户

---

# 隐藏租户管理权限

我们现在有点问题：因为用户 *tenant2* 有 *Administration:Security* 权限，所以他可以访问用户和角色权限对话框。因此，他可以使用权限 UI 给自己授予 *Administration:Tenants* 权限。



Serenity 扫描程序集的特性，比如 *ReadPermission*、*WritePermission*、*PageAuthorize*、*ServiceAuthorize* 等，并在编辑权限对话框列出这些权限。

我们应该先从默认值列表中删除它。

在 *UserPermissionRepository.cs* 找到 *ListPermissionKeys* 方法：

```

public ListResponse<string> ListPermissionKeys()
{
    return LocalCache.Get("Administration:PermissionKeys", TimeSpan.Zero, () =>
    {
        //...

        result.Remove(Administration.PermissionKeys.Tenants);
        result.Remove("*");
        result.Remove("?");

        //...
    });
}

```

现在，该权限不会在 编辑用户权限 或 编辑角色权限 对话框中被列出来。

但是，黑客仍然可以通过 *UserPermissionRepository.Update* 或 *RolePermissionRepository.Update* 方法给自己授权。

我们应该添加一些检查来阻止该行为：

```

public class UserPermissionRepository
{
    public SaveResponse Update(IUnitOfWork uow,
        UserPermissionUpdateRequest request)
    {
        //...

        var newList = new Dictionary<string, bool>(
            StringComparer.OrdinalIgnoreCase);
        foreach (var p in request.Permissions)
            newList[p.PermissionKey] = p.Grant ?? false;

        var allowedKeys = ListPermissionKeys()
            .Entities.ToDictionary(x => x);
        if (newList.Keys.Any(x => !allowedKeys.ContainsKey(x)))
            throw new AccessViolationException();
        //...
    }
}

```

```
public class RolePermissionRepository
{
    public SaveResponse Update(IUnitOfWork uow,
        RolePermissionUpdateRequest request)
    {
        //...
        var newList = new HashSet<string>(
            request.Permissions.ToList(),
            StringComparer.OrdinalIgnoreCase);

        var allowedKeys = new UserPermissionRepository()
            .ListPermissionKeys()
            .Entities.ToDictionary(x => x);
        if (newList.Any(x => !allowedKeys.ContainsKey(x)))
            throw new AccessViolationException();
        //...
    }
}
```

在这里，我们认为任何试图授权不在权限对话框列表的新访问许可键都是一个黑客行为。

实际上，即使在非多租户系统中也应该默认做这个检查，但是通常我们信任管理的用户。在这里，管理员只能管理他们自己的租户，所以我们必须包含该检查。

# 多租户角色

目前为止，我们已经让用户页面在多租户风格下工作。为使它工作，我们看起来并没有做太多的变化。但请记住，我们正在对一个原来不是多租户的系统作修改。

让我们在 `Roles` 表应用类似的原则。

再一次，一个租户的用户在不能查看或修改其他租户的角色，每个租户的用户是相互独立工作的。

我们先在 `RoleRow.cs` 添加 `TenantId` 属性：

```
namespace MultiTenancy.Administration.Entities
{
    //...
    public sealed class RoleRow : Row, IIdRow, INameRow
    {
        [Insertable(false), Updatable(false)]
        public Int32? TenantId
        {
            get { return Fields.TenantId[this]; }
            set { Fields.TenantId[this] = value; }
        }

        //...

        public class RowFields : RowFieldsBase
        {
            //...
            public readonly Int32Field TenantId;
            //...
        }
    }
}
```

然后，我们在 `RoleRepository.cs` 做几处修改：

```
private class MySaveHandler : SaveRequestHandler<MyRow>
```

```

{
    protected override void SetInternalFields()
    {
        base.SetInternalFields();

        if (IsCreate)
            Row.TenantId = ((UserDefinition)Authorization.UserDefinition).TenantId;
    }
}

private class MyDeleteHandler : DeleteRequestHandler<MyRow>
{
    protected override void ValidateRequest()
    {
        base.ValidateRequest();

        var user = (UserDefinition)Authorization.UserDefinition;
        if (Row.TenantId != user.TenantId)
            Authorization.ValidatePermission(PermissionKeys.Tenants);
    }
}

private class MyRetrieveHandler : RetrieveRequestHandler<MyRow>
{
    protected override void PrepareQuery(SqlQuery query)
    {
        base.PrepareQuery(query);

        var user = (UserDefinition)Authorization.UserDefinition;
        if (!Authorization.HasPermission(PermissionKeys.Tenants))
            query.Where(fld.TenantId == user.TenantId);
    }
}

private class MyListHandler : ListRequestHandler<MyRow>
{
    protected override void ApplyFilters(SqlQuery query)
}

```

```
{  
    base.ApplyFilters(query);  
  
    var user = (UserDefinition)Authorization.UserDefinition;  
    if (!Authorization.HasPermission(PermissionKeys.Tenants)  
)  
        query.Where(fld.TenantId == user.TenantId);  
}  
}
```

# 使用 **Serenity** 服务行为

如果想把多租户系统扩展到 Northwind 数据库中的其他表，我们会重复角色所做的相同步骤。虽然看起来没那么难，但是有太多的手工工作。

Serenity 提供服务行为系统，它可以允许我们拦截添加、更新、检索、列表、删除的操作处理并向其添加用户自定义代码。

在这些处理中有一些操作（如像获取日志、唯一约束验证等）已经使用服务行为实现了。

行为（Behaviors）可能被所有的行（rows）激活，或被基于某些规则（如特定的特性或接口）的行激活。例如，含 [CaptureLog] 特性的行激活

CaptureLogBehavior。

我们首先定义一个将触发新行为的接口 *IMultiTenantRow*。把此类放在 *TenantRow.cs* 旁边的 *IMultiTenantRow.cs* 中：

```
using Serenity.Data;

namespace MultiTenancy
{
    public interface IMultiTenantRow
    {
        Int32Field TenantIdField { get; }
    }
}
```

然后在旁边的 *MultiTenantBehavior.cs* 文件添加行为：

```
using MultiTenancy.Administration;
using Serenity;
using Serenity.Data;
using Serenity.Services;

namespace MultiTenancy
{
```

```
public class MultiTenantBehavior : IImplicitBehavior,
    ISaveBehavior, IDeleteBehavior,
    IListBehavior, IRetrieveBehavior
{
    private Int32Field fldTenantId;

    public bool ActivateFor(Row row)
    {
        var mt = row as IMultiTenantRow;
        if (mt == null)
            return false;

        fldTenantId = mt.TenantIdField;
        return true;
    }

    public void OnPrepareQuery(IRetrieveRequestHandler handler,
        SqlQuery query)
    {
        var user = (UserDefinition)Authorization.UserDefinition;
        if (!Authorization.HasPermission(PermissionKeys.Tenants))
            query.Where(fldTenantId == user.TenantId);
    }

    public void OnPrepareQuery(IListRequestHandler handler,
        SqlQuery query)
    {
        var user = (UserDefinition)Authorization.UserDefinition;
        if (!Authorization.HasPermission(PermissionKeys.Tenants))
            query.Where(fldTenantId == user.TenantId);
    }

    public void OnSetInternalFields(ISaveRequestHandler handler)
    {
```

```
        if (handler.IsCreate)
            fldTenantId[handler.Row] =
                ((UserDefinition)Authorization
                    .UserDefinition).TenantId;
    }

    public void OnValidateRequest(IDeleteRequestHandler handler)
    {
        var user = (UserDefinition)Authorization.UserDefinition;
        if (fldTenantId[handler.Row] != user.TenantId)
            Authorization.ValidatePermission(
                PermissionKeys.Tenants);
    }

    public void OnAfterDelete(IDeleteRequestHandler handler)
    { }

    public void OnAfterExecuteQuery(IRetrieveRequestHandler handler) { }

    public void OnAfterExecuteQuery(IListRequestHandler handler) { }

    public void OnAfterSave(ISaveRequestHandler handler) { }

    public void OnApplyFilters(IListRequestHandler handler,
        SqlDataReader query) { }

    public void OnAudit(IDeleteRequestHandler handler) { }

    public void OnAudit(ISaveRequestHandler handler) { }

    public void OnBeforeDelete(IDeleteRequestHandler handler)
    { }

    public void OnBeforeExecuteQuery(IRetrieveRequestHandler handler) { }

    public void OnBeforeExecuteQuery(IListRequestHandler handler) { }

    public void OnBeforeSave(ISaveRequestHandler handler) { }

    public void OnPrepareQuery(IDeleteRequestHandler handler,
        SqlDataReader query) { }

    public void OnPrepareQuery(ISaveRequestHandler handler,
        SqlDataReader query) { }

    public void OnReturn(IDeleteRequestHandler handler) { }
```

```
public void OnReturn(IRetrieveRequestHandler handler) {  
}  
public void OnReturn(IListRequestHandler handler) { }  
public void OnReturn(ISaveRequestHandler handler) { }  
public void OnValidateRequest(IRetrieveRequestHandler han  
dler) { }  
public void OnValidateRequest(IListRequestHandler handle  
r) { }  
public void OnValidateRequest(ISaveRequestHandler handle  
r) { }  
}  
}
```

行为类实现 `IImplicitBehavior` 接口来决定是否应该被指定的行类型（row type）激活。

它是通过实现 `ActivateFor` 方法做到的，该方法由请求处理（request handlers）调用。

在该方法中，我们检查行类型（row type）是否实现 `IMultiTenantRow` 接口，如果没有，则返回 `false`。

然后，我们得到一个 `TenantIdField` 的私有引用，以便之后在其他方法中使用。

`ActivateFor` 在每个处理类型（handler type）和行（row）中只被调用一次。如果该方法返回 `true`，行为实例出于性能考虑而被缓存，并且被该行（row）和处理类型（handler type）重用。

因此，由于每个实例都被所有的请求共享，所以你在其他方法中所写的代码必须是线程安全的。

一个行为通过实现 `IRetrieveBehavior`, `IListBehavior`, `ISaveBehavior`, 或 `IDeleteBehavior` 接口，可以拦截一个或多个检索、列表、保存、删除处理。

在这里，我们需要拦截所有这些服务调用，因此我们实现所有的接口。

我们只实现相关的方法，其他的方法保留为空。

我们这里实现的方法，对应于上一章节在 `RoleRepository.cs` 重写的方法。它们所包含的代码几乎是相同的，但我们这里需要更加通用，因为该行为将为所有实现 `IMultiTenantRow` 接口的行类型工作。

## 使用行为重新实现 **RoleRepository**

现在还原我们在 *RoleRepository.cs* 做的所有修改：

```
private class MySaveHandler : SaveRequestHandler<MyRow> { }
private class MyDeleteHandler : DeleteRequestHandler<MyRow> { }
private class MyRetrieveHandler : RetrieveRequestHandler<MyRow>
{ }
private class MyListHandler : ListRequestHandler<MyRow> { }
```

并且在 *RoleRow* 添加 *IMultiTenantRow* 接口：

```
namespace MultiTenancy.Administration.Entities
{
    //...
    public sealed class RoleRow : Row, IIdRow, INameRow, IMultiT
enantRow
    {
        //...
        public Int32Field TenantIdField
        {
            get { return Fields.TenantId; }
        }
        //...
    }
}
```

使用更少的代码得到相同的结果。声明式编程几乎总是更好的选择。

# 扩展多租户行为到 Northwind

由于我们现在有一个行为处理仓储的详细信息，我们只需在行（rows）实现 *IMultiTenantRow* 接口并添加 *TenantId* 属性。

从 *SupplierRow.cs* 开始：

```
namespace MultiTenancy.Northwind.Entities
{
    //...
    public sealed class SupplierRow : Row,
        IIdRow, INameRow, IMultiTenantRow
    {
        //...
        [Insertable(false), Updatable(false)]
        public Int32? TenantId
        {
            get { return Fields.TenantId[this]; }
            set { Fields.TenantId[this] = value; }
        }

        public Int32Field TenantIdField
        {
            get { return Fields.TenantId; }
        }

        //...

        public class RowFields : RowFieldsBase
        {
            //...
            public readonly Int32Field TenantId;
        }
    }
}
```

当你在 *SupplierRow* 做这些更改并生成后，你将看到 *tenant2* 不能在供应商页面看到其他租户的供应商。

现在，在

*EmployeeRow*、*CategoryRow*、*CustomerRow*、*ShipperRow*、*OrderRow*、*ProductRow*、*RegionRow* 和 *TerritoryRow* 重复这些修改。

# 处理检索脚本 (Lookup Scripts)

如果我们现在打开 *Suppliers* 页面，我们可以看到 *tenant2* 只能查看属于自己的供应商。但是在网格列表的右上角的 country 下拉列表中，所有的国家选项都被列出来了。

ID	Company Name	Contact Name	Phone	Country
30	Test			France

Country dropdown options:

- Australia
- Brazil
- Canada
- Denmark
- Finland
- France
- Germany
- Italy

这个数据是通过动态脚本提供给脚本端。它不会在我们最近处理的服务列表中加载该数据。

提供给该下拉列表的检索脚本在 *SupplierCountryLookup.cs* 中定义：

```

namespace MultiTenancy.Northwind.Scripts
{
    using Serenity.ComponentModel;
    using Serenity.Data;
    using Serenity.Web;

    [LookupScript("Northwind.SupplierCountry")]
    public class SupplierCountryLookup :
        RowLookupScript<Entities.SupplierRow>
    {
        public SupplierCountryLookup()
        {
            IdField = TextField = "Country";
        }

        protected override void PrepareQuery(SqlQuery query)
        {
            var fld = Entities.SupplierRow.Fields;
            query.Distinct(true)
                .Select(fld.Country)
                .Where(
                    new Criteria(fld.Country) != "" &
                    new Criteria(fld.Country).IsNotNull());
        }

        protected override void ApplyOrder(SqlQuery query)
        {
        }
    }
}

```

因为实际上 Northwind 数据库并没有 country 表，所以我们不能在行 (row) 类上使用简单的 [LookupScript] 特性。我们从供应商表的现有记录中收集不同的国家名称。

我们应该通过当前的租户来筛选查询。

但是这个检索类继承自基类 *RowLookupScript*。让我们创建一个新的基类，为稍后再处理其他检索脚本准备。

```
namespace MultiTenancy.Northwind.Scripts
{
    using Administration;
    using Serenity;
    using Serenity.Data;
    using Serenity.Web;
    using System;

    public abstract class MultiTenantRowLookupScript<TRow> :
        RowLookupScript<TRow>
        where TRow : Row, IMultiTenantRow, new()
    {
        public MultiTenantRowLookupScript()
        {
            Expiration = TimeSpan.FromDays(-1);
        }

        protected override void PrepareQuery(SqlQuery query)
        {
            base.PrepareQuery(query);
            AddTenantFilter(query);
        }

        protected void AddTenantFilter(SqlQuery query)
        {
            var r = new TRow();
            query.Where(r.TenantIdField ==
                ((UserDefinition)Authorization.UserDefinition).T
enantId);
        }

        public override string GetScript()
        {
            return TwoLevelCache.GetLocalStoreOnly("MultiTenantL
ookup:" +
                this.ScriptName + ":" +
                ((UserDefinition)Authorization.UserDefinitio
n).TenantId,
                TimeSpan.FromHours(1),
                new TRow().GetFields().GenerationKey, () =>
```

```
    {
        return base.GetScript();
    });
}
}
```

这将是我们多租户检索脚本的基类。

我们首先将过期时间设置设为一个负的时间间隔来禁用缓存。为什么要这样做呢？因为动态脚本使用键值（**keys**）管理检索脚本的缓存。但我们会有很多个基于 **TenantId** 值的检索脚本的版本。

我们会在动态脚本管理层面关闭缓存并在 **GetScript** 方法中自己处理缓存。在 **GetScript** 方法中，我们使用 **TwoLevelCache.GetLocalStoreOnly** 调用基方法生成我们的检索脚本，并缓存其包含 **TenantId** 缓存键的结果。

更多关于 **TwoLevelCache** 类的信息，请查看相关章节。

通过重写 **PrepareQuery** 方法，我们添加一个使用当前 **TenantId** 的过滤器，就像我们在列表服务处理中做的一样。

现在是时候使用新基类重写 **SupplierCountryLookup**：

```
namespace MultiTenancy.Northwind.Scripts
{
    using Serenity.ComponentModel;
    using Serenity.Data;
    using Serenity.Web;

    [LookupScript("Northwind.SupplierCountry")]
    public class SupplierCountryLookup :
        MultiTenantRowLookupScript<Entities.SupplierRow>
    {
        public SupplierCountryLookup()
        {
            IdField = TextField = "Country";
        }

        protected override void PrepareQuery(SqlQuery query)
        {
            var fld = Entities.SupplierRow.Fields;
            query.Distinct(true)
                .Select(fld.Country)
                .Where(
                    new Criteria(fld.Country) != "" &
                    new Criteria(fld.Country).IsNotNull());
            AddTenantFilter(query);
        }

        protected override void ApplyOrder(SqlQuery query)
        {
        }
    }
}
```

因为我们没有在这里调用基类的 *PrepareQuery* 方法（因此它不会由基类调用），所以我们就手工调用 *AddTenantFilter* 方法。

如果有 *Northwind.DynamicScripts.cs* 文件，请先删除它。

在 *CustomerCountryLookup*、*CustomerCityLookup*、*OrderShipCityLookup*、*OrderShipCountryLookup* 有几个非常相似的检索脚本。我将对它们做类似的修改：把基类改为 *MultiTenantRowLookupScript* 并在 *PrepareQuery* 方法中调用 *AddTenantFilter*。

我们现在还有一个问题需要解决：如果你打开 *Orders* 页面，你将看到 *Ship Via* 和 *Employee* 过滤下拉列表一直列出其他租户的记录。这是因为我们使用含 *[LookupScript]* 特性的行定义检索脚本。

让我们首先来修复雇员检索 (employee lookup)，从 *EmployeeRow* 中删除 *[LookupScript]* 特性。

```
[ConnectionKey("Northwind"), DisplayName("Employees"), InstanceName("Employee"), TwoLevelCached]
[ReadPermission(Northwind.PermissionKeys.General)]
[ModifyPermission(Northwind.PermissionKeys.General)]
public sealed class EmployeeRow : Row, IIdRow, INameRow, IMultiTenantRow
{
    //...
```

并在 *EmployeeRow.cs* 旁边的 *EmployeeLookup* 文件添加一个新的检索：

```
namespace MultiTenancy.Northwind.Scripts
{
    using Entities;
    using Serenity.ComponentModel;
    using Serenity.Web;

    [LookupScript("Northwind.Employee")]
    public class EmployeeLookup :
        MultiTenantRowLookupScript<EmployeeRow>
    {
    }
}
```

由于基类会帮我们处理所有的事情，我们没有重写任何东西。默认情况下，行的 *LookupScript* 特性使用 *RowLookupScript* 作为基类定义一个新的自动检索脚本类。

由于没有办法重写每行的基类，我们显式定义检索脚本类，并使用 *MultiTenantRowLookupScript* 作为基类。

现在如果生成并运行应用程序，你将得到一个错误：

```
'MultiTenancy.Northwind.Entities.EmployeeRow' type doesn't have  
a  
[LookupScript] attribute, so it can't be used with a LookupEditor  
r!
```

这是由于我们的行类前面没有 *[LookupScript]* 特性，但是在一些像表单的地方，我们需要使用 *[LookupEditor("Northwind.Employee")]*。

打开 *OrderRow.cs*，你将看到在 *EmployeeID* 属性上面有这个特性。把它修改为 *[LookupEditor("Northwind.Employee")]*。

我们在 *ShipperRow* 做类似的事情。删除 *LookupScript* 特性并定义下面的类：

```
namespace MultiTenancy.Northwind.Scripts  
{  
    using Entities;  
    using Serenity.ComponentModel;  
    using Serenity.Web;  
  
    [LookupScript("Northwind.Shipper")]  
    public class ShipperLookup :  
        MultiTenantRowLookupScript<ShipperRow>  
    {  
    }  
}
```

并且在 *OrderRow* 的 *ShipVia* 属性上面你将找到另一个相似的 *LookupEditor* 特性。把它修改为 *[LookupEditor("Northwind.Shipper")]*。

在 *ProductRow* 重复同样的步骤。

```
namespace MultiTenancy.Northwind.Scripts
{
    using Entities;
    using Serenity.ComponentModel;
    using Serenity.Web;

    [LookupScript("Northwind.Product")]
    public class ProductLookup : MultiTenantRowLookupScript<ProductRow>
    {
    }
}
```

在 *OrderDetailRow* 的 *ProductID* 属性上面你将找到另一个相似的 *LookupEditor* 特性。把它修改为 `[LookupEditor("Northwind.Product")]`。

*Supplier*、*Category*、*Region*、*Territory* 是下一个需要做类似处理的类。请查看 Serenity 教程的 [github](#) 仓库提交日志。

现在 Northwind 支持多租户。

可能有一些被我忽略了的小问题，如果发现任何问题，请在 Serenity 的 Github 仓库中的提交报告。

如果大家对多租户应用程序有足够的兴趣，该功能可能会被集成到 Serenity。

## 会议管理（进行中.....）

在这个教程中，我们打算开发一个会议管理系统，来帮助跟踪公司会议。

要招开公司会议首先得计划会议地点、时间、议程和参会人员，然后向这些参会人员发送电子邮件邀请。

应用程序也会存储会议决定，并以电子邮件的方式向参会人员发送包含这些决定的会议报告。

可在如下链接找到该教程的源代码：

<https://github.com/volkanceylan/MeetingManagement>

### 创建项目

首先利用 **Serene** 模板创建一个新项目，并命名为 **MeetingManagement**。

### 删除 **Northwind**

使用疑难解惑指南中的方法删除 **Northwind**。

# 创建检索表

让我们从创建系统所要用到的检索表开始。

这是所需的表：

- 会议类型表（董事会会议、每周分析、SCRUM 会议、年会等）；
- 地点表（举行会议的地址、房间号码等）；
- 议程类型表（会议的主题，一个可能有多个主题）；
- 单位表（此次会议的组织单位）；
- 联系人表（参会人员、记者、管理层等）。

我们使用 *met* 作为数据库表的 schema。

在 *Modules/Common/Migrations/DefaultDB* 创建一个名为 *DefaultDB\_20160709\_232400\_MeetingLookups* 的迁移类：

```
using FluentMigrator;

namespace MeetingManagement.Migrations.DefaultDB
{
    [Migration(20160709232400)]
    public class DefaultDB_20160709_232400_MeetingLookups
        : AutoReversingMigration
    {
        public override void Up()
        {
            Create.Schema("met");

            Create.Table("AgendaTypes").InSchema("met")
                .WithColumn("AgendaTypeId").AsInt32()
                    .Identity().PrimaryKey().NotNullable()
                .WithColumn("Name").AsString(100).NotNullable();

            Create.Table("Contacts").InSchema("met")
                .WithColumn("ContactId").AsInt32()
                    .Identity().PrimaryKey().NotNullable()
                .WithColumn("Title").AsString(30).Nullable();
        }
    }
}
```

```

        .WithColumn("FirstName").AsString(50).NotNullable()
    e()
        .WithColumn("LastName").AsString(50).NotNullable()
    () 
        .WithColumn("Email").AsString(100).NotNullable()
    ;
}

Create.Table("Locations").InSchema("met")
    .WithColumn("LocationId").AsInt32()
        .Identity().PrimaryKey().NotNullable()
    .WithColumn("Name").AsString(100).NotNullable()
    .WithColumn("Address").AsString(300).Nullable()
    .WithColumn("Latitude").AsDouble()
    .WithColumn("Longitude").AsDouble();

Create.Table("MeetingTypes").InSchema("met")
    .WithColumn("MeetingTypeId").AsInt32()
        .Identity().PrimaryKey().NotNullable()
    .WithColumn("Name").AsString(100).NotNullable();

Create.Table("Units").InSchema("met")
    .WithColumn("UnitId").AsInt32()
        .Identity().PrimaryKey().NotNullable()
    .WithColumn("Name").AsString(100).NotNullable();
}

}
}
}

```

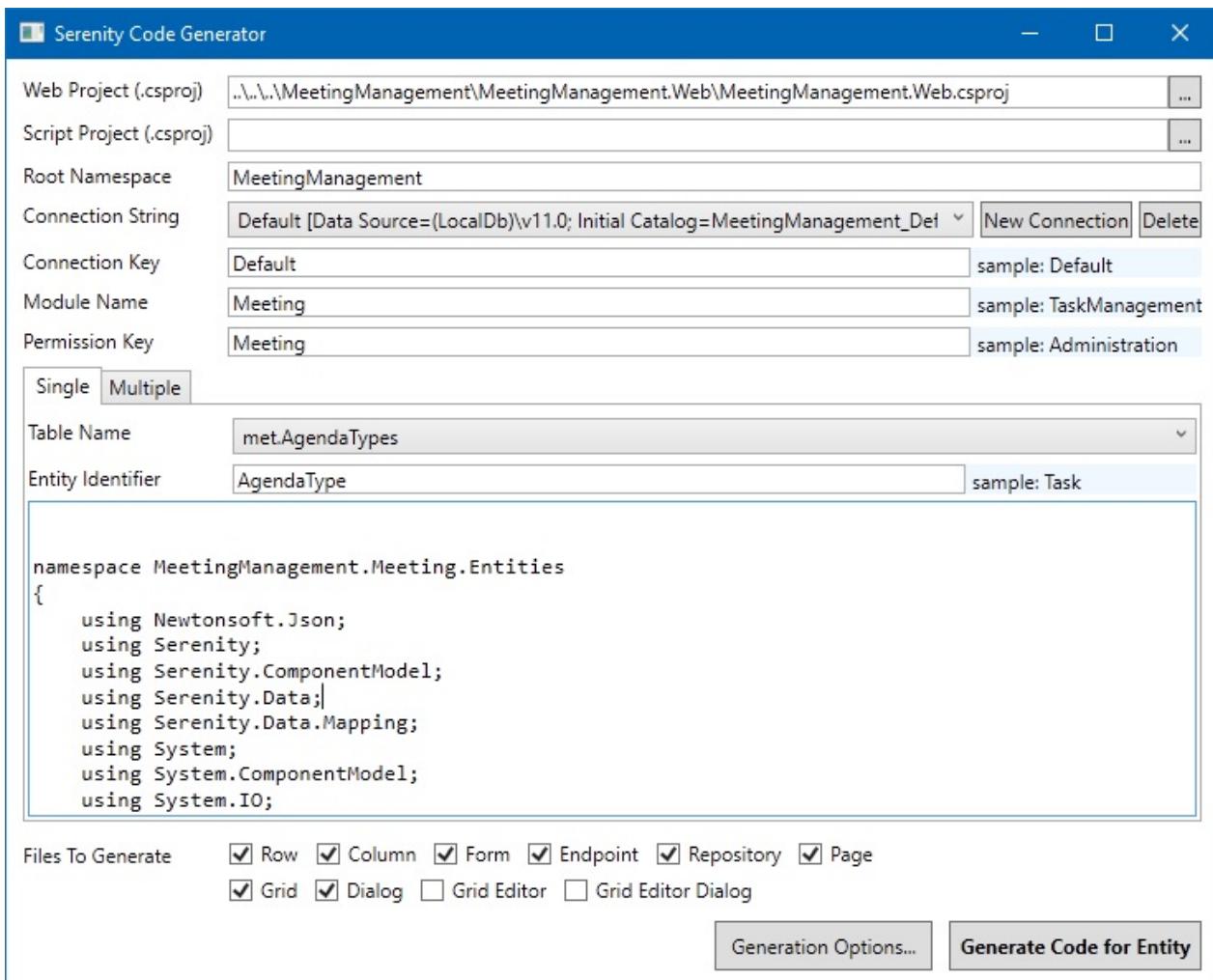
## 为检索表生成代码

我们的模块名称为 *Meetings*。在生成代码时，我们应该使用非复数的实体标识符：

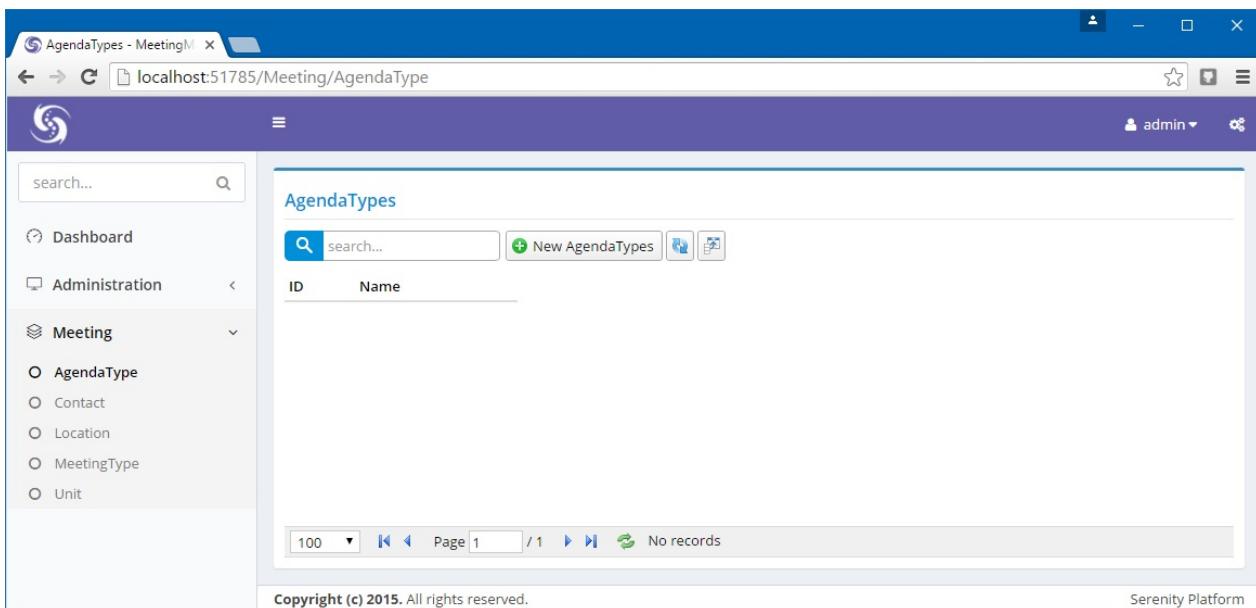
- AgendaTypes => AgendaType
- Contacts => Contact
- Locations => Location
- MeetingTypes => MeetingType
- Units => Unit

为这 5 张表使用上面给出的实体标识符生成代码：

## 创建检索表



对这些表生成的界面并不是很美观，需要做一些美化。



把导航连接移到 **NavigationItems.cs**

打开

*AgendaTypePage.cs*、*ContactPage.cs*、*LocationPage.cs*、*MeetingTypePage.cs* 和 *UnitPage.cs* 文件，并把文件顶部的导航连接移至 *NavigationItems.cs*：

```
using Serenity.Navigation;
using Administration = MeetingManagement.Administration.Pages;
using Meeting = MeetingManagement.Meeting.Pages;

[assembly: NavigationLink(1000, "Dashboard",
    url: "~/", permission: "", icon: "icon-speedometer")]

[assembly: NavigationMenu(2000, "Meeting")]
[assembly: NavigationLink(2500, "Meeting/Agenda Types",
    typeof(Meeting.AgendaTypeController))]
[assembly: NavigationLink(2600, "Meeting/Contacts",
    typeof(Meeting.ContactController))]
[assembly: NavigationLink(2700, "Meeting/Locations",
    typeof(Meeting.LocationController))]
[assembly: NavigationLink(2800, "Meeting/Meeting Types",
    typeof(Meeting.MeetingTypeController))]
[assembly: NavigationLink(2900, "Meeting/Units",
    typeof(Meeting.UnitController))]
```

## 为检索表设置 **DisplayName** 和 **InstanceName** 特性

打开

*AgendaTypeRow.cs*、*ContactRow.cs*、*LocationRow.cs*、*MeetingTypeRow.cs* 和 *UnitRow.cs* 文件，并像下面那样修改 *DisplayName* 和 *InstanceName* 特性：

- *AgendaTypeRow* => "Agenda Types", "Agenda Type"
- *ContactRow* => "Contacts", "Contact"
- *LocationRow* => "Locations", "Location"
- *MeetingTypeRow* => "Meeting Types", "Meeting Type"
- *UnitRow* => "Units", "Unit"

```
[ConnectionKey("Default"), TwoLevelCached,  
 DisplayName("Agenda Types"), InstanceName("Agenda Type")]  
[ReadPermission("Meeting")]  
[ModifyPermission("Meeting")]  
public sealed class AgendaTypeRow : Row, IIdRow, INameRow  
{
```

## 疑难解惑指南

# 如何删除 Serene 的 Northwind 及其他示例？

当你以 Northwind 为样本开发完自己的项目后，你会想从项目中移除 Northwind 和其他示例模块。

这里将介绍如何移除它们。

假设你的解决方案名称是 *MyProject*，因此在你的解决方案中有 *MyProject.Web* 项目。

在 Visual Studio 中执行如下步骤：

## 移除项目文件

- 移除 *MyProject.Web/Modules/AdminLTE* 文件夹。这将删除所有主题示例相关的服务器端代码。
- 移除 *MyProject.Web/Modules/BasicSamples* 文件夹。这将删除所有基本示例的服务器端代码。
- 移除 *MyProject.Web/Modules/Northwind* 文件夹。这将删除所有 Northwind 相关的服务器端代码。

## 移除导航项目

- 打开 *MyProject.Web/Modules/Common/Navigation/NavigationItems.cs*，删除所有含 Northwind、Basic Samples 和 Theme Samples 的行，并删除下面两行代码：

```
using Northwind = MovieTutorial.Northwind.Pages;
using Basic = MovieTutorial.BasicSamples.Pages;
```

## 移除迁移脚本

删除 *MyProject.Web/Modules/Common/Migrations/NorthwindDB/* 文件夹下的所有文件。

从 *MyProject.Web/App\_Start/SiteInitialization.Migrations.cs* 的下面行中删除 "Northwind"。

```
private static string[] databaseKeys = new[] { "Default", "Northwind" };
```

同样，从 *web.config* 删除 *Northwind* 连接字符串。

```
<add name="Northwind" connectionString="Data Source=(LocalDb)\v11.0;
    Initial Catalog=MovieTutorial_Northwind_v1;
    Integrated Security=True"
    providerName="System.Data.SqlClient" />
```

## 移除 LESS 记录

- 打开 *MyProject.Web/Content/site/site.less* 文件，删除下面的代码：

```
@import "site.basicsamples.less";
@import "site.northwind.less";
```

- 删除 *MyProject.Web/Content/site/site.basicsamples.less* 文件。
- 删除 *MyProject.Web/Content/site/site.northwind.less* 文件。

## 移除本地化文本

- 打开 *MyProject.Web/Modules/Texts.cs* 文件，并删除下面的代码：

```
public static LocalText NorthwindPhone = "...";
public static LocalText NorthwindPhoneMultiple = "...";
```

- 删除文件夹 *MyProject.Web/Scripts/site/texts/northwind*。
- 删除文件夹 *MyProject.Web/Scripts/site/texts/samples*。

## 移除 **Northwind / Samples** 生成的代码

- 展开 *MyProject.Web/Modules/Common/Imports/ServerTypings/ServerTypings.tt*。删除以 *Northwind* 或 *BasicSamples* 开头的文件。

## 移除控制面板（**Dashboard**）中的 **Northwind** 数字

打开 *DashboardPage.cs*，删除 using 行：

```
using Northwind;
using Northwind.Entities;
```

由于控制面板从 *Northwind* 表获取数据，你应该把 *Index()* 操作修改为：

```
[Authorize,HttpGet,Route("~/")]
public ActionResult Index()
{
    var cachedModel = new DashboardPageModel()
    {

    }

    return View(MVC.Views.Common.Dashboard.DashboardIndex, cache
dModel);
}
```

你应该替换该模块为你网站的具体内容，并相应地修改 *DashboardIndex*。

打开 *DashboardIndex.cshtml*，清除包含 "Northwind" 的 *href* 属性内容，如：

```
<a href "~/Northwind/Order?shippingState=1"></a>
<a href=""></a>
```

## 生成项目并运行 **T4 (.tt)** 模板

- 现在重新生成你的解决方案；

- 在执行下一步骤前，请确保成功生成；
- 点击生成菜单，然后点击转换所有模板；
- 再次重新生成解决方案。
- 在整个解决方案中搜索 *Northwind*、*Basic Samples* 和 *Theme Samples*，应该找不到任何结果。
- 运行项目，现在已经没有 *Northwind* 和 *Sample* 菜单。

## 移除 **Northwind** 表

*Northwind* 的表都在一个单独的数据库中，因此，删除该数据库即可。

## 如何更新 **Serenity** 的 **NuGet** 程序包？

Serene 模板包含下面的 Serenity NuGet 程序包引用：

```
Serenity.Core  
Serenity.Data  
Serenity.Data.Entity  
Serenity.Services  
Serenity.Web  
Serenity.CodeGenerator
```

要更新到最新版本的 Serenity 程序包，打开程序包管理器控制台（点击 视图 -> 其他窗口 -> 程序包管理器控制台）。

并输入：

```
Update-Package Serenity.Web  
Update-Package Serenity.CodeGenerator
```

更新这些包的同时也将更新其他的程序包（由于依赖引用关系）。

# 如何升级到 **Serenity 2.0** 并启用 **TypeScript** ?

Serenity 从 2.0 版本开始支持 TypeScript。

这是一个为使用旧版本 **Serene** 模板且想使用 TypeScript 功能的用户提供的迁移指南。

如果你不需要 TypeScript，只需要更新你的 Serenity 程序包，它应该能正常工作。

即使你不需要 TypeScript，还是建议你执行此处列出的步骤来更新你的项目，因为 **Serene** 已经发生了许多变化以支持 TypeScript，升级可能会帮助你避免将来遇到问题。

## 我应该切换到 **TypeScript** 吗？

Serenity 已经稳定支持使用 TypeScript 并强烈推荐你也使用它编写代码。因为 TypeScript 有更强的后盾（微软和大量用户），TypeScript 将是 Serenity 应用程序的未来。

此外 TypeScript 感觉像原生 Javascript 拥有特赞的智能感知、重构和编译时类型检查。

自 Serenity 一开始，我们一直使用 Saltaralle，但由于它于去年 6 月（2015 年）被 Bridge.NET 收购，此后没有得到任何更新，它的未来有点模糊。

使用 Saltaralle 写的旧代码将继续工作，Serenity 尽可能地向前兼容。

如果 Bridge.NET v2.0（下一个 Saltaralle）出来后，我们可能尝试切换回去，除非它涉及的改变大多而不能处理。

## 迁移你的 **Serene** 应用程序到 **v2.0**

### 请检查你的解决方案正确生成

首先确保你的解决方案能成功生成。

如果可能的话，以 ZIP 备份解决方案，因为我们执行的一些步骤可能难以还原。

## 安装 TypeScript

从

<https://www.typescriptlang.org/#download-links>

为你的 Visual Studio 版本安装 TypeScript 1.8+ 。

## 更新 NuGet 程序包

像平常那样更新 Serenity 2.0 的程序包：

```
Update-Package Serenity.Web  
Update-Package Serenity.CodeGenerator  
Update-Package Serenity.Script
```

当更新 Serenity.Web 时，VS 可能会显示 "你的项目已经配置为支持 TypeScript" 的对话框，点击确定即可。

## 确保更新程序包没有导致问题

再次重新生成你的解决方案并运行它。打开一些页面、对话框等，以确保在 2.0 程序包下正确地工作。

## 在 Web 项目中配置 TypeScript

卸载 MyProject.Web 并编辑它：

在 TypeScriptToolsVersion 后面添加下面的配置：

```
// ...
<TypeScriptToolsVersion>1.8</TypeScriptToolsVersion>
<TypeScriptCompileBlocked>True</TypeScriptCompileBlocked>
</PropertyGroup>
<PropertyGroup>
    <TypeScriptCharset>utf-8</TypeScriptCharset>
    <TypeScriptEmitBOM>True</TypeScriptEmitBOM>
    <TypeScriptGeneratesDeclarations>False</TypeScriptGeneratesD
eclarations>
    <TypeScriptExperimentalDecorators>True</TypeScriptExperiment
alDecorators>
    <TypeScriptOutFile>Scripts\site\Serene.Web.js</TypeScriptOut
File>
    <TypeScriptCompileOnSaveEnabled>False</TypeScriptCompileOnSa
veEnabled>
</PropertyGroup>
```

用项目名称替代 **Serene.Web.js** 。

在同一文件的末尾，你将看到像下面的内容：

```
<Import Project="...Microsoft.CSharp.targets" />
<Import Project="...Microsoft.WebApplication.targets" />
<Import Project="...Microsoft.TypeScript.targets" />
```

确保含 **TypeScript.targets** 的行在所有其他目标 (**targets**) 下面。如果不在该位置，则把它移到 **WebAplication.targets** 下面。若把它放在 **Microsoft.WebApplication.targets** 之前，VS 将不能正常工作。

同样地，在文件底部，你可以找到 编译网站 **Less (CompileSiteLess)** 的步骤，在这里添加 **TSC** :

```
<Target Name="CompileSiteLess" AfterTargets="AfterBuild">
  <Exec Command="$(ProjectDir)tools\node\lessc.cmd";
        &quot;$(ProjectDir)Content\site\site.less&quot; &gt;
        &quot;$(ProjectDir)Content\site\site.css&quot;">
  </Exec>
  <Exec Command="$(TscToolPath)\$(TypeScriptToolsVersion"
        $(TscToolExe)&quot; -project &quot;
        $(ProjectDir)tsconfig.json&quot;" ContinueOnError="true"
  />
</Target>
```

保存修改，重新加载项目并继续下面的步骤。

## 添加 **tsconfig.json** 文件

在你的 Web 项目的根目录下（web.conifg 和 Global.asax 文件所在的位置）添加 **tsconfig.json** 文件，其内容如下：

```
{
  "compileOnSave": true,
  "compilerOptions": {
    "preserveConstEnums": true,
    "experimentalDecorators": true,
    "declaration": true,
    "emitBOM": true,
    "sourceMap": true,
    "target": "ES5",
    "outFile": "Scripts/site/Serene.Web.js"
  },
  "exclude": [
    "Scripts/site/Serene.Web.d.ts"
  ]
}
```

用项目名称替代 \***Serene.Web** 。

## 添加一个测试的 **TypeScript** 文件

在 YourProject.Web/scripts/site/ 中添加一个文件 **dummy.ts**。打开该文件并输入如下内容：

```
namespace MyProject {  
    export class Dummy {  
    }  
}
```

当你保存该文件后，应该有一个含如下内容的 **MyProject.Web.js** 文件。如果你没有看到该文件，点击 显示所有文件 并刷新文件夹。

```
var MyProject;  
(function (MyProject) {  
    var Dummy = (function () {  
        function Dummy() {  
        }  
        return Dummy;  
    }());  
    MyProject.Dummy = Dummy;  
})(MyProject || (MyProject = {}));  
//# sourceMappingURL=SereneUpgrading.Web.js.map
```

右键并包含该文件到项目。现在你可以删除 **dummy.ts**。

如果你使用 VS2015 之前的版本，可能保存后不能编译 TS 文件，它们将在生成项目时被编译。

## 在 **\_LayoutHead.cshtml** 包含 **MyProject.Web.js** 文件

编辑 **MyProject.Web/Views/Shared/\_LayoutHead.cshtml**，并在 **MyProject.Script.js** 之后包含 **MyProject.Web.js** 引用。

```
// ...
@Html.Script("~/Scripts/Site/MyProject.Script.js")
@Html.Script("~/Scripts/Site/MyProject.Web.js")
// ...
```

你项目的 TypeScript 已经配置好了。

## 修改 T4 模板的位置

Serene v2.0 已经合并了一些 .TT 模板并为 TypeScript 创建了一个新的代码生成器。

请确保你的项目成功生成，并且在执行生成时，不要取消生成，否则你将得到残缺的项目。

定位到 YourProject.Web\Modules\Common\Imports\  
MultipleOutputHelper.ttinclude 文件。

在同一文件夹中复制一份该文件，并重命名为 *CodeGenerationHelpers.ttinclude*。

复制下面连接中最新 *CodeGenerationHelpers.ttinclude* 代码，并把它粘贴到新建的文件中：

<https://raw.githubusercontent.com/volkanceylan/Serene/master/Serene/Serene.Web/Modules/Common/Imports/CodeGenerationHelpers.ttinclude>

在该文件中 查找 **Serene** 并替换为 **YourProjectName**，该文件中不应该含任何 **Serene** 。

| 你也可以用最新版本的模板创建 Serene 项目来获得这些文件。

## ClientTypes.tt

创建文件夹 *YourProject.Web\Modules\Common\Imports\ClientTypes* 并把 *ScriptEditorTypes.tt* 移到该文件夹，然后把 *ScriptEditorTypes.tt* 重命名为 *ClientTypes.tt*。

从下面连接复制粘贴得到 *ClientTypes.tt* 内容：

<https://raw.githubusercontent.com/volkanceylan/Serene/master/Serene/Serene.Web/Modules/Common/Imports/ClientTypes/ClientTypes.tt>

在该文件中 查找 **Serene** 并替换为 **YourProjectName**

## ServerTypings.tt

创建文件夹 *YourProject.Web\Modules\Common\Imports\ServerTypings* 并把 *ScriptFormatterTypes.tt* 移到该文件夹，然后把 *ScriptFormatterTypes.tt* 重命名为 *ServerTypings.tt*。

从下面连接复制粘贴得到 *ServerTypings.tt* 内容：

<https://raw.githubusercontent.com/volkanceylan/Serene/master/Serene/Serene.Web/Modules/Common/Imports/ServerTypings/ServerTypings.tt>

在该文件中 查找 **Serene** 并替换为 **YourProjectName**

## 生成代码

当保存打开的 *ClientTypes.tt* 和 *ServerTypings.tt* 文件时，模板文件将生成代码。

保存并重新生成项目。

## 修改 T4 模板的 **FormContexts** 和 **ServiceContracts** 的位置

这两个模板已经合并成一个。

我们将在 Web 项目重复类似的步骤。

定位到 *YourProject.Script\Imports\ MultipleOutputHelper.ttinclude* 文件。

在同一文件夹中复制一份该文件，并重命名为 *CodeGenerationHelpers.ttinclude*。

复制下面连接（连接已经不一样了）中最新 *CodeGenerationHelpers.ttinclude* 代码，并把它粘贴到新建的文件中：

<https://raw.githubusercontent.com/volkanceylan/Serene/b900c67b4c820284379b9c613b16379bb8c530f3/Serene/Serene.Script/Imports/CodeGenHelpers.ttinclude>

在该文件中 查找 **Serene** 并替换为 **YourProjectName**。

## ServiceContracts.tt

把文件夹 *YourProject.Script\Imports\ServiceContracts* 重命名为 **ServerImports**，并把 *ServiceContracts.tt* 重命名为 *ServerImports.tt*。

从下面连接复制粘贴得到 *ServerImports.tt* 内容：

<https://raw.githubusercontent.com/volkanceylan/Serene/b900c67b4c820284379b9c613b16379bb8c530f3/Serene/Serene.Script/Imports/ServerImports/ServerImports.tt>

在该文件中 查找 **Serene** 并替换为 **YourProjectName**。

删除 *FormContexts* 文件夹及其 *FormContext.tt* 文件。Delete folder *FormContexts* with file *FormContext.tt* in it.

保存 **ServerImports.tt** 并等待其生成代码。这可能得花点时间，由于包含一些 Saltaralle 代码导致其生成速度慢下来。

重新生成解决方案并确保没有生成错误。

恭喜！你的项目已经为 **TypeScript** 和其他功能做好准备了

## 这些新的 **.tt** 文件有什么作用？

- **ServerTypings.tt**: 为 TypeScript 生成代码，包括 行 (Row)、表单 (Form)、列 (Column)、从服务器 (Web) 代码导入服务声明引用。此外，如果有 *YourProject.Script* 文件，则将导入该类的引用。
- **ServerTypes.tt**: 为 Saltaralle 生成代码，包括 行 (Row)、表单 (Form)、列 (Column)、从服务器 (Web) 代码导入服务声明引用。这里还没有从 TypeScript 导入类引用，因此如果你想在 Saltaralle 代码中使用一些 TypeScript 类，需要手工导入 (*import*) 类引用。

- **ClientTypes.tt**: 为 Web 项目生成代码，包含从 TypeScript 和 Saltaralle 导入的编辑器和格式化器。

## 怎样生成 **TypeScript** 网格/对话框代码

Serenity 代码生成器 (Sergen) 现在有一个选项，只需勾选 **Generate Grid/Dialog Code in TypeScript (instead of Saltaralle)** 就可在 YourProject.Web/Modules/YourEntity 目录下生成 YourDialog.ts 和 YourGrid.ts 文件，以替代 YourProject.Script 项目的 YourGrid.cs 和 YourDialog.cs 。

请不要使用 **Sergen** 为已存在的 **Saltaralle** 对话框或网格生成代码。否则，你将有两个 **YourGrid** 和 **YourDialog** 类，很可能导致始料不及的错误。

## 如何使用其他类型的数据库？

Serenity 有方言系统（dialect system）以支持在非 Sql Server 的其他数据库中工作。

方言系统（Dialect system）目前还是一项在不断完善的实验性功能。虽然它也能工作，但如果有任何问题，你可以在 Serenity 的 GitHub 向我们反馈。

如果你需要支持多种数据库类型，只需在 `web.config` 修改连接字符串，并在表达式中小心使用数据库特定的功能及避免使用保留的关键字。

## PostgreSQL

### 注册 Npgsql 提供者

PostgreSQL 有一个名为 Npgsql 的 .NET 提供者。你首先需要在 `MyProject.Web` 中安装该提供者：

```
Install-Package Npgsql -Project MyProject.Web
```

如果你之前没有在 `GAC/machine.config` 安装该提供者，或者不想在此安装，你需要在 `web.config` 文件配置：

```
<configuration>
// ...
<system.data>
<DbProviderFactories>
<remove invariant="Npgsql"/>
<add name="Npgsql Data Provider"
      invariant="Npgsql"
      description=".Net Data Provider for PostgreSQL"
      type="Npgsql.NpgsqlFactory, Npgsql, Culture=neutral,
            PublicKeyToken=5d8b90d52f46fda7"
      support="FF" />
</DbProviderFactories>
</system.data>
// ...
```

## 设置连接字符串

下一步是把连接字符串修改为 Postgres 数据库的配置：

确保你使用自己的服务器信息替换连接字符串的对应参数值。

```
<connectionStrings>
<add name="Default" connectionString="
    Server=127.0.0.1;Database=serene_default_v1;
    User Id=postgres;Password=yourpassword;" 
    providerName="Npgsql" />

<add name="Northwind" connectionString="
    Server=127.0.0.1;Database=serene_northwind_v1;
    User Id=postgres;Password=yourpassword;" 
    providerName="Npgsql" />
</connectionStrings>
```

由于 Postgres 总会将字母转换为小写，所以请使用小写的数据库名称，如 `serene_default_v1`。

提供者名称必须是 `Npgsql`，以使 Serenity 自动检测方言。

## 注意标识符的大小写

PostgreSQL 的标识符区分大小写。

FluentMigrator 自动为所有标识符添加引号，所以数据库中的表和列名称将被添加引号并区分大小写。当试图检索不带引号的表或列标识符时，可能会导致问题。

一种做法是在迁移中始终使用小写字母标识符，但这种命名方案在其它数据库类型中不是很好，因此我们不喜欢这种方式。

为了防止 Postgres 的这类问题，Serenity 有自动添加引号功能，以解决 Postgres/FluentMigrator 的兼容问题，但需要在应用程序的 `SiteInitialization.cs` 启动方法中启用：

```
public static void ApplicationStart()
{
    try
    {
        SqlSettings.AutoQuotedIdentifiers = true;
        Serenity.Web.CommonInitialization.Run();
    }
}
```

确保在 `CommonInitialization.Run` 运行之前设置 `AutoQuotedIdentifiers`。

该设置自动为实体的列名称添加引号，但不能应用在手工编写的表达式中（如，表达式特性）。

如果你想支持多数据库类型，在表达式中对标识符使用方括号 `[]`。Serenity 在运行查询之前，将自动把方括号转为数据库具体的引用类型（`quote type`）。

你可能还希望在表达式中使用双引号，但它不能与 MySQL 数据库兼容。

## 设置默认方言

这一步骤是可选的。

Serenity 通过检索连接字符串的 `providerName` 自动决定使用的方言。

它甚至可以在同一时间与多个数据库类型工作。

例如，`Northwind` 使用 `Sql Server`，而 `Default` 使用 `PostgreSQL`。

但是，如果你打算每个站点只使用一种数据库类型，可以在 `SiteInitialization` 注册默认使用的数据库类型。

```
public static void ApplicationStart()
{
    try
    {
        SqlSettings.DefaultDialect = PostgresDialect.Instance;
        SqlSettings.AutoQuotedIdentifiers = true;
        Serenity.Web.CommonInitialization.Run();
    }
}
```

当连接方言/实体等不能自动确定时，使用默认方言。

该设置不会覆盖自动检测，它只是被用作回退。

## 启动应用程序

现在启动你的应用程序，如果之前没有手工创建数据库，它将自动创建数据库。

## 配置代码生成器

`Sergen` 没有 PostgreSQL 提供者的引用，因此如果若想使用它生成代码，你必须向其注册该提供者。

`Sergen.exe` 是一个 `exe` 文件，因此你不能为它添加 NuGet 引用。我们需要在应用程序配置文件中注册该提供者。

也可以在 GAC/machine.config 注册提供者并完全跳过此步骤。

定位到 `Sergen.exe`，该文件在 `packages/Serenity.CodeGenerator.1.8.6/tools` 目录下，并创建文件 `Sergen.exe.config`，然后向其添加如下内容：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.data>
    <DbProviderFactories>
      <remove invariant="Npgsql"/>
      <add name="Npgsql Data Provider"
           invariant="Npgsql"
           description=".Net Data Provider for PostgreSQL"
           type="Npgsql.NpgsqlFactory, Npgsql, Culture=neutral,
                 PublicKeyToken=5d8b90d52f46fda7"
           support="FF" />
    </DbProviderFactories>
  </system.data>
  <appSettings>
    <add key="LoadProviderDLLs" value="Npgsql.dll"/>
  </appSettings>
</configuration>
```

把 Npgsql.dll 拷贝到与 Sergen.exe 同一文件夹。现在 Sergen 将可以为 Postgres 表生成代码。

如果想能够使用多个数据库，你可能想删除 [public]. 前缀，该前缀是生成的行 (rows) 中 tablename/column 表达式的默认 schema。

## MySql

Serene 1.8.13+ 支持 MySql

### 注册 MySql 提供者

MySQL 有一个名为 MySql.Data 的 .NET 提供者。你首先需要在 MyProject.Web 中安装该提供者：

Install-Package MySql.Data -Project MyProject.Web

如果你之前没有在 GAC/machine.config 安装该提供者，或者不想在此安装，你需要在 web.config 文件配置（MySql.Data NuGet 程序包在安装时已经为我们添加了该配置）：

```
<configuration>
// ...
<system.data>
<DbProviderFactories>
<remove invariant=" MySql.Data.MySqlClient" />
<add name="MySQL Data Provider"
      invariant=" MySql.Data.MySqlClient"
      description=".Net Framework Data Provider for MySQL"
      type=" MySql.Data.MySqlClient.MySqlClientFactory,
            MySql.Data, Culture=neutral,
            PublicKeyToken=c5687fc88969c44d" />
</DbProviderFactories>
</system.data>
// ...
```

## 设置连接字符串

下一步是把连接字符串修改为 MySQL 数据库的配置：

确保你使用自己的服务器信息替换连接字符串的对应参数值。

```
<connectionStrings>
<add name="Default" connectionString="
      Server=localhost; Port=3306; Database=Serene_Default
      _v1;
      Uid=root; Pwd=yourpass"
      providerName=" MySql.Data.MySqlClient" />

<add name="Northwind" connectionString="
      Server=localhost; Port=3306; Database=Serene_Northwi
      nd_v1;
      Uid=root; Pwd=yourpass"
      providerName=" MySql.Data.MySqlClient" />
</connectionStrings>
```

提供者名称必须是 `MySql.Data.MySqlClient`，以使 Serenity 自动检测方言。注意，上述重写默认方言。

MySQL 使用小写的数据库（schema）和表名称，但没有 Postgres 的区分大小写问题。

## 配置代码生成器

Sergen 没有 MySQL 提供者的引用，因此如果你想使用它生成代码，你必须向其注册该提供者。

Sergen.exe 是一个 `exe` 文件，因此你不能为它添加 NuGet 引用。我们需要在应用程序配置文件中注册该提供者。

也可以在 `GAC/machine.config` 注册提供者并完全跳过此步骤。

定位到 Sergen.exe，该文件在 `packages/Serenity.CodeGenerator.1.8.6/tools` 目录下，并创建文件 `Sergen.exe.config`，然后向其添加如下内容：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.data>
    <DbProviderFactories>
      <remove invariant="MySql.Data.MySqlClient"/>
      <add name="MySQL Data Provider"
           invariant="MySql.Data.MySqlClient"
           description=".Net Framework Data Provider for MySQL"
           type="MySql.Data.MySqlClient.MySqlClientFactory,
                 MySql.Data, Culture=neutral,
                 PublicKeyToken=c5687fc88969c44d" />
    </DbProviderFactories>
  </system.data>
  <appSettings>
    <add key="LoadProviderDLLs" value="MySql.Data.dll"/>
  </appSettings>
</configuration>
```

把 `MySql.Data.dll` 拷贝到与 Sergen.exe 同一文件夹。现在 Sergen 将可以为 Postgres 表生成代码。

## Oracle

Serene 2.2.2+ 已支持 Oracle 数据库

### 注册 Oracle 提供者

Oracle 有一个名为 `Oracle.ManagedDataAccess` 的托管 .NET 提供者。你首先需要在 `MyProject.Web` 中安装该提供者：

```
Install-Package Oracle.ManagedDataAccess -Project MyProject.Web
```

如果你之前没有在 `GAC/machine.config` 安装该提供者，或者不想在此安装，你需要在 `web.config` 文件配置 (`Oracle.ManagedDataAccess` NuGet 程序包在安装时已经为我们添加了该配置)：

```
<configuration>
    // ...
    <system.data>
        <DbProviderFactories>
            <remove invariant="Oracle.ManagedDataAccess.Client"/>
            <add name="ODP.NET, Managed Driver"
                invariant="Oracle.ManagedDataAccess.Client"
                description="Oracle Data Provider for .NET, Managed D
river"
                type="Oracle.ManagedDataAccess.Client.OracleClientFac
tory,
                Oracle.ManagedDataAccess, Version=4.121.2.0, Cu
lture=neutral,
                PublicKeyToken=89b483f429c47342"/>
        </DbProviderFactories>
    </system.data>
    // ...
```

### 创建数据库

Serene 不能在 Oracle 自动创建数据库 (tablespace)。你需要自己创建它们，或者使用下面的脚本 (我在 XE 运行该脚本)：

```
CREATE TABLESPACE Serene_Default_v1_TABS
    DATAFILE 'Serene_Default_v1_TABS.dat' SIZE 10M AUTOEXTEND ON
;
CREATE TEMPORARY TABLESPACE Serene_Default_v1_TEMP
    TEMPFILE 'Serene_Default_v1_TEMP.dat' SIZE 5M AUTOEXTEND ON;
CREATE USER Serene_Default_v1
    IDENTIFIED BY somepassword
    DEFAULT TABLESPACE Serene_Default_v1_TABS
    TEMPORARY TABLESPACE Serene_Default_v1_TEMP;
GRANT CREATE SESSION TO Serene_Default_v1;
GRANT CREATE TABLE TO Serene_Default_v1;
GRANT CREATE SEQUENCE TO Serene_Default_v1;
GRANT CREATE TRIGGER TO Serene_Default_v1;
GRANT UNLIMITED TABLESPACE TO Serene_Default_v1;

CREATE TABLESPACE Serene_Northwind_v1_TABS
    DATAFILE 'Serene_Northwind_v1_TABS.dat' SIZE 10M AUTOEXTEND
ON;
CREATE TEMPORARY TABLESPACE Serene_Northwind_v1_TEMP
    TEMPFILE 'Serene_Northwind_v1_TEMP.dat' SIZE 5M AUTOEXTEND ON
;
CREATE USER Serene_Northwind_v1
    IDENTIFIED BY somepassword
    DEFAULT TABLESPACE Serene_Northwind_v1_TABS
    TEMPORARY TABLESPACE Serene_Northwind_v1_TEMP;

GRANT CREATE SESSION TO Serene_Northwind_v1;
GRANT CREATE TABLE TO Serene_Northwind_v1;
GRANT CREATE SEQUENCE TO Serene_Northwind_v1;
GRANT CREATE TRIGGER TO Serene_Northwind_v1;
GRANT UNLIMITED TABLESPACE TO Serene_Northwind_v1;
```

## 设置连接字符串

你可能想要为 ORACLE 配置你的数据源。我在这里用 Express Edition (XE) :

```
<configuration>
  <oracle.manageddataaccess.client>
    <version number="*">
      <dataSources>
        <dataSource alias="XE"
          descriptor="">
          (DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)
            (HOST=localhost)(PORT=1521))
            (CONNECT_DATA=(SERVICE_NAME=XE))) ">
      </dataSources>
    </version>
  </oracle.manageddataaccess.client>
</configuration>
```

下一步是把连接字符串修改为 Oracle 数据库的配置：

确保你使用自己的服务器信息替换连接字符串的对应参数值。

```
<connectionStrings>
  <add name="Default" connectionString="
    Data Source=XE;User Id=Serene_Default_v1;Password=somepassword;">
    providerName="Oracle.ManagedDataAccess.Client"/>
  <add name="Northwind" connectionString="
    Data Source=XE;User Id=Serene_Northwind_v1;Password=somepassword;">
    providerName="Oracle.ManagedDataAccess.Client"/>
</connectionStrings>
```

提供者名称必须是 `Oracle.ManagedDataAccess.Client`，以使 Serenity 自动检测方言。注意，上述重写默认方言。

## 配置代码生成器

Sergen 目前还不支持 Oracle，希望尽快支持.....

## 如何使用其他类型的数据库？

---

# 如何使用 Active Directory 或 LDAP 进行身份验证？

Serene 1.8.12+ 有一些基本的 ActiveDirectory / LDAP 集成示例。

若要启用它们，你必须在 `web.config` 配置为两者之一。

如果要启用 ActiveDirectory，则需要在 `appSetting` 添加 `ActiveDirectory` 键，内容如下：

```
<add key="ActiveDirectory"
      value="{ Domain: 'youractivedirectorydomain' }" />
```

如果该配置在你的 Active Directory 服务器中不能工作，则需要修改 `ActiveDirectoryService` 类。

当 AD 用户第一次尝试登录，Serene 使用配置的域对用户进行身份认证，检索用户详细信息并把具有类型 `目录 (directory)` 的用户插入到 `用户 (users)` 表。

AD 密码使用哈希算法加密，并且缓存用户信息一小时，所以用户可以在一小时内不用密码就可使用缓存凭据登录。

在那之后，尝试从 AD 更新用户信息。如果发生错误，将允许用户使用缓存凭据登录。

这些细节可以在 `AuthenticationService` 类中查看并修改。

若要启用 LDAP 身份认证（使用 OpenLDAP 测试），你需要在 `web.config` 的 `appSetting` 节点添加 `LDAP` 键：

```
<add key="LDAP"
      value="{
        Host: '123.124.125.126',
        Port: 389,
        DistinguishedName: 'dc=yourdomain, dc=com',
        Username: 'cn=someuserthatcanreadldap, ou=groupofthatuser
',
        Password: 'passwordofthatuser'
    }"
/>
```

再次，有许多不同 LDAP 服务器配置，所以如果你的配置不能正常工作，你可能需要修改 *LdapDirectoryService* 类。

## 如何设置连接的数据库方言 (Database Dialect) ?

有时，使用 `providerName` 的自动方言检测可能不会正常工作，或对于某些连接，你想要使用 `SqlServer2000` 或 `SqlServer2005` 方言。

可以设置默认的全局方言，但这不会覆盖自动检测：

```
SqlSettings.DefaultDialect = SqlServer2005Dialect.Instance;
```

由于 "Northwind" 和 "Default" 连接的提供者名称是 "System.Data.SqlClient"，即使你重写了全局方言，Serenity 也将自动把它们的方言设置为 `SqlServer2012`。

但是也可能在基本的连接键 (connection key) 中修改方言：

```
public static partial class SiteInitialization
{
    public static void ApplicationStart()
    {
        try
        {
            SqlConnections.GetConnectionString("Default").Dialect =
                SqlServer2005Dialect.Instance;

            SqlConnections.GetConnectionString("Northwind").Dialect =
                SqlServer2005Dialect.Instance;
        }
    }
}
```

Serenity 1.8.8+ 也支持通过应用程序配置条目设置：

```
<configuration>
  <appSettings>
    <add key="ConnectionSettings" value="{
      Default: {
        Dialect: 'SqlServer2005'
      },
      Northwind: {
        Dialect: 'Postgres'
      }
    }"/>
```

## 在行 (**Rows**) 中提示关于 **CONCAT** 及其他类似信息的警告

Serene 已经支持多种数据库引擎，包括 MySQL、Postgres 等。这些数据库没有像 MsSqlServer 的字符串加号 (+) 运算符。因此，在 Northwind 中，用 CONCAT 函数来替代 + 运算符：

```
[Expression("CONCAT(T0.[FirstName], CONCAT(' ', T0.[LastName]))")]
public String FullName
{
    get { return Fields.FullName[this]; }
    set { Fields.FullName[this] = value; }
}
```

CONCAT 在 Sql Server 2012 之后的版本是可用的。所以如果你要使用旧版本的 SQL server，例如 SQL server 2005 或 SQL server 2008，这些表达式将被替换成：

```
[Expression("T0.[FirstName] + ' ' + T0.[LastName]")]
public String FullName
{
    get { return Fields.FullName[this]; }
    set { Fields.FullName[this] = value; }
}
```

## 如何设置连接的数据库方言（Database Dialect）？

---

## 如何删除网格中的新增（add）按钮？

本节是为 **Saltaralle** 编写。若是 **TypeScript** 版本，请参阅 **Serene** 的 **Basic Samples => Grids => Removing Add Button** 页面

你可以在 *MyGrid.ts* 重写 *getButtons()* 方法。

下面的代码会删除所有按钮（包括刷新）：

```
public class MyGrid : EntityGrid<MyRow>
{
    //...

    protected override List<ToolButton> GetButtons()
    {
        return new List<ToolButton>();
    }
}
```

使用索引（0 - 第一个按钮）删除新增（add）按钮：

```
public class MyGrid : EntityGrid<MyRow>
{
    //...

    protected override List<ToolButton> GetButtons()
    {
        var buttons = base.GetButtons();
        buttons.RemoveAt(0);
        return buttons;
    }
}
```

使用硬编码索引有一种糟糕代码的气味

或者，返回一个新的按钮列表：

```
public class MyGrid : EntityGrid<MyRow>
{
    //...

    protected override List<ToolButton> GetButtons()
    {
        var buttons = new List<ToolButton>();

        buttons.Add(new ToolButton
        {
            Title = "My New Button",
            CssClass = "my-new-button",
            OnClick = delegate {
                // ...
            }
        });

        buttons.Add(NewRefreshButton(noText: true));

        return buttons;
    }
}
```

# 如何使用 **SlickGrid** 格式化器 (Formatter) ?

本节的 **TypeScript** 版本有待更新

使用 SlickGrid 格式化器功能，例如在 SlickGrid 列中显示完成程度的百分比进度条示例：

<http://mleibman.github.io/SlickGrid/examples/example2-formatters.html>

## 包含必须的资源

首先在 `_LayoutHead.cshtml` (`MyProject.Web/Views/Shared/_LayoutHead.cshtml`) 文件中包含这些格式化程序的引用：

```
//...
@Html.Script("~/Scripts/jquery.slimscroll.js")
@Html.Script("~/Scripts/SlickGrid/slick.formatters.js")
@Html.Script("~/Scripts/Site/MovieTutorial.Script.js")
//...
```

同样需要包含下面的 CSS (来自 `example.css`，可以插入到 `site.less`) 。

```
.percent-complete-bar {
    display: inline-block;
    height: 6px;
    -moz-border-radius: 3px;
    -webkit-border-radius: 3px;
}
```

## 定义 **Serenity** 的 **DataGrid** 格式化器

假设我们想在 `StudentCourseGrid` 的 `CourseCompletion` 列使用 `SlickFormatters.PercentCompleteBar` 格式化。

```
public class StudentCourseColumns
{
    //...
    [Width(200)]
    public Decimal CourseCompletion { get; set; }
}
```

要在服务器端引用 SlickGrid 格式化器，需要为 Serenity 网格定义一个格式化器类型。

以在 `MyApplication.Script` 项目的 `StudentCourseGrid.cs` 旁定义一个文件 (`PercentCompleteBarFormatter.cs`) 为例，该文件内容如下：

```
using Serenity;
using System;

namespace MyApplication
{
    public class PercentCompleteBarFormatter : ISlickFormatter
    {
        private SlickColumnFormatter formatter =
            Type.GetType("Slick.Formatters.PercentCompleteBar").
As<SlickColumnFormatter>();

        public string Format(SlickFormatterContext ctx)
        {
            return formatter(ctx.Row, ctx.Cell, ctx.Value, ctx.C
olumn, ctx.Item);
        }
    }
}
```

使用你的根命名空间（解决方案名称）替换 `MyApplication`。

现在你可以在服务端引用它：

```
public class StudentCourseColumns
{
    //...
    [FormatterType("PercentCompleteBar"), Width(200)]
    public Decimal CourseCompletion { get; set; }
}
```

重新生成你的项目，你将看到 CourseCompletion 列有一个像 SlickGrid 示例的百分比栏。

## 获得智能感知和编译时检查

要在 PercentCompleteBarFormatter 服务端获得智能感知（可避免使用魔术字符串），你应该转换 T4 模板（在转换之前请确保成功生成解决方案）。

在此之后，你可以在服务器端像下面这样引用它：

```
public class StudentCourseColumns
{
    //...
    [PercentCompleteBarFormatter, Width(200)]
    public Decimal CourseCompletion { get; set; }
}
```

## 替代方案（不建议）

也可以不用定义 Serenity 格式化器类，例如在 *StudentCourseGrid.cs* 通过重写其 *GetColumns* 方法直接在脚本代码中设置 SlickGrid 列格式化器功能：

```
protected override List<SlickColumn> GetColumns()
{
    var columns = base.GetColumns();
    columns.Single(x => x.Field == "CourseCompletion").Formatter =
        Type.GetType("Slick.Formatters.PercentCompleteBar").
As<SlickColumnFormatter>();
    return columns;
}
```

该方式是不可重用的，但可省去定义格式化器类。

## 如何将内联动作按钮（**Inline Action Buttons**）添加到网格？

本节的 **TypeScript** 版本有待更新

你可以在网格添加一些内联动作图标，用以删除当前行、打开对话框、调用服务等。

让我们在客户网格列表（CustomersGrid）添加一个含删除按钮的新列。

首先从 CustomerGrid 删除 IAsyncInit 接口，否则你得重写 GetColumnsAsync。

```
namespace Serene.Northwind
{
    //...
    public class CustomerGrid : EntityGrid<CustomerRow>
    {
        public CustomerGrid(jQueryObject container)
            : base(container)
        {
        }
        //...
        protected override List<SlickColumn> GetColumns()
        {
            var columns = base.GetColumns();

            columns.Add(new SlickColumn
            {
                Field = "DeleteRow",
                Title = "",
                Format = ctx =>
                {
                    return "<a class='inline-action delete-row' title='delete'></a>";
                },
                Width = 24,
                MinWidth = 24,
                MaxWidth = 24
            });

            return columns;
        }
    }
}
```

现在我们需要给我们的按钮添加样式。在 `site.less` 末尾添加下面的样式：

```
.inline-action {  
    background-repeat: no-repeat;  
    background-position: center center;  
    width: 16px;  
    height: 16px;  
    cursor: pointer;  
    opacity: 0.5;  
    display: inline-block;  
  
    &:hover {  
        opacity: 1;  
    }  
}  
  
.delete-row {  
    background-image: url(../serenity/images/delete2.png);  
}
```

我用基本 CSS 类 `inline-action` 来设置默认按钮的样式。通过这种方式，你可以只改变 `delete-row` 类并设置其 `background-image` 创建不同的动作按钮。

生成并运行项目后，在客户网格列表最右边，我们将有一个删除连接。当你点击该连接，不会有任何响应，因为我们还没有处理该点击的事件。下面我们来添加该点击事件：

```
protected override void OnClick(jQueryEvent e, int row, int cell)
{
    base.OnClick(e, row, cell);

    if (e.IsDefaultPrevented())
        return;

    var item = Items[row];

    if (J(e.Target).HasClass("inline-action"))
    {
        e.PreventDefault();

        if (J(e.Target).HasClass("delete-row"))
        {
            Q.Confirm("Delete record?", () =>
            {
                CustomerService.Delete(new DeleteRequest
                {
                    EntityId = item.ID.Value
                }, onSuccess: response =>
                {
                    Refresh();
                });
            });
        }
    }
}
```

我们首先检查点击事件是否被基本的网格类处理，来防止同样的事件被处理再次。例如，基本的网格类在 customer ID 和 name 处理了编辑连接。

然后，我们获得点击行的实体。

下一步骤是检查单击的 HTML 元素是否有 inline-action 样式，如果有该样式，我们就阻止其默认行为。

然后，我们检查单击元素是否是 `delete-row` 操作，如果是该操作，则弹出确认框给用户确认。 `Q.Confirm` 在第二个参数中指定用户确认操作后所调用的操作。

如果用户确认了删除操作，我们调用 `CustomerService.Delete` 方法删除所选的行，并使用删除成功之后的回调方法刷新网格。

让我们添加另一个操作来编辑客户（你可以用相同的方法打开另一对话框）。

```
protected override List<SlickColumn> GetColumns()
{
    var columns = base.GetColumns();

    columns.Add(new SlickColumn
    {
        Field = "DeleteRow",
        Title = "",
        Format = ctx =>
        {
            return "<a class='inline-action delete-row' title='d
elete'></a>";
        },
        Width = 24,
        MinWidth = 24,
        MaxWidth = 24
    });

    columns.Add(new SlickColumn
    {
        Field = "EditRow",
        Title = "",
        Format = ctx =>
        {
            return "<a class='inline-action edit-row' title='edi
t customer'></a>";
        },
        Width = 24,
        MinWidth = 24,
        MaxWidth = 24
    });

    return columns;
}
```

为其添加 CSS：

```
.delete-row {  
    background-image: url(../serenity/images/delete2.png);  
}  
  
.edit-row {  
    background-image: url(../serenity/images/magnifier.png);  
}
```

添加单击处理事件：

```
protected override void OnClick(jQueryEvent e, int row, int cell)
{
    base.OnClick(e, row, cell);

    if (e.IsDefaultPrevented())
        return;

    var item = Items[row];

    if (J(e.Target).HasClass("inline-action"))
    {
        e.PreventDefault();

        if (J(e.Target).HasClass("delete-row"))
        {
            Q.Confirm("Delete record?", () =>
            {
                CustomerService.Delete(new DeleteRequest
                {
                    EntityId = item.ID.Value
                }, onSuccess: response =>
                {
                    Refresh();
                });
            });
        }
        else if (J(e.Target).HasClass("edit-row"))
        {
            var dlg = new CustomerDialog();
            InitEntityDialog(dlg);
            dlg.LoadByIdAndOpenDialog(item.ID.Value);
        }
    }
}
```

我们首先创建一个新的客户对话框。InitEntityDialog 是 CustomerGrid 的一个方法，它允许附加 ondatachange 事件，因此当对话框的数据改变时，它可以自动刷新。然后，我们调用 LoadByIdAndOpenDialog 加载所选 ID 的客户到对话框，在加

载完成后显示对话框。

# 如何添加一个行选择列？

本节的 **TypeScript** 版本有待更新

若要添加列以选择单行或所有行，可以使用 **GridRowSelectionMixin**。

GridRowSelectionMixin 在 Serenity 1.6.8+ 有效。

示例代码：

```
public class MyGrid : EntityGrid<MyRow>
{
    private GridRowSelectionMixin rowSelection;

    public MyGrid(jQueryObject container)
        : base(container)
    {
        rowSelection = new GridRowSelectionMixin(this);
    }

    protected override List<SlickColumn> GetColumns()
    {
        var columns = base.GetColumns();
        columns.Insert(0, GridRowSelectionMixin.CreateSelectColumn(() => rowSelection));
        return columns;
    }

    protected override List<ToolButton> GetButtons()
    {
        var buttons = base.GetButtons();

        buttons.Add(new ToolButton
        {
            CssClass = "tag-button",
            Title = "Do Something With Selected Rows",
            OnClick = delegate
            {
                var selectedIDs = rowSelection.GetSelectedKeys();
            }
        });
    }
}
```

```
;  
    if (selectedIDs.Count == 0)  
        Q.NotifyWarning("Please select some rows");  
    else  
        Q.NotifySuccess("You have selected " + selectedIDs.Count +  
" row(s) with ID(s): " + string.Join(",  
", selectedIDs));  
    }  
});  
  
return buttons;  
}  
  
}
```

# 如何设置级联编辑器（Cascaded Editors）？

你可能需要多级级联的编辑器，如 国家 => 城市、课程 => 班级 => 科目。

从 Serenity 1.8.2 开始，级联编辑将变得简单。检索编译器（Lookup editors）已经集成了该功能。

在 1.8.2 之前的版本，也可以实现该功能，并且在 Serene 中也有一些示例，但为使它工作，你需要定义一些编辑器类。

假设我们的数据库有三张表：Country、City、District：

- **Country Table:** CountryId, CountryName
- **City Table:** CityId, CityName, CountryId
- **District Table:** DistrictId, DistrictName, CityId

首先，确保你已经使用 Sergen 为这三张表生成代码，并且都含有 [LookupScript] 特性：

```
[LookupScript("MyModule.Country")]
public sealed class CountryRow : Row...
{
    [DisplayName("Country Id"), Identity]
    public Int32? CountryId
    {
        get { return Fields.CountryId[this]; }
        set { Fields.CountryId[this] = value; }
    }

    [DisplayName("Country Name"), Size(50), NotNull, QuickSearch]
    public String CountryName
    {
        get { return Fields.CountryName[this]; }
        set { Fields.CountryName[this] = value; }
    }
}
```

```
[LookupScript("MyModule.City")]
public sealed class CityRow : Row...
{
    [DisplayName("City Id"), Identity]
    public Int32? CityId
    {
        get { return Fields.CityId[this]; }
        set { Fields.CityId[this] = value; }
    }

    [DisplayName("City Name"), Size(50), NotNull, QuickSearch]
    public String CityName
    {
        get { return Fields.CityName[this]; }
        set { Fields.CityName[this] = value; }
    }

    [DisplayName("Country"), ForeignKey("Country", "CountryId"),
     LookupInclude]
    public Int32? CountryId
    {
        get { return Fields.CountryId[this]; }
        set { Fields.CountryId[this] = value; }
    }

}
```

```
[LookupScript("MyModule.District")]
public sealed class DistrictRow : Row...
{
    [DisplayName("District Id"), Identity]
    public Int32? DistrictId
    {
        get { return Fields.DistrictId[this]; }
        set { Fields.DistrictId[this] = value; }
    }

    [DisplayName("District Name"), Size(50), NotNull, QuickSearch]
    public String DistrictName
    {
        get { return Fields.DistrictName[this]; }
        set { Fields.DistrictName[this] = value; }
    }

    [DisplayName("City"), ForeignKey("City", "CityId"), LookupInclude]
    public Int32? CityId
    {
        get { return Fields.CityId[this]; }
        set { Fields.CityId[this] = value; }
    }

}
```

请确保在 DistrictRow 的 CityId 字段和 CityRow 的 CountryId 字段添加 `LookupInclude` 特性。我们在客户端需要它们，否则在默认情况下，它们不包含在检索脚本中。

如果你想在表单（如 CustomerForm）中以级联检索的形式编辑这些字段。你需要把它们设置为：

```
[FormScript("MyModule.Customer")]
[BasedOnRow(typeof(Entities.CustomerRow))]
public class CustomerForm
{
    public String CustomerID { get; set; }
    public String CustomerName { get; set; }

    [LookupEditor(typeof(Entities.CountryRow))]
    public Int32? CountryId { get; set; }

    [LookupEditor(typeof(Entities.CityRow),
        CascadeFrom = "CountryId", CascadeField = "CountryId")]
    public Int32? CityId { get; set; }

    [LookupEditor(typeof(Entities.DistrictRow),
        CascadeFrom = "CityId", CascadeField = "CityId")]
    public Int32? DistrictId { get; set; }
}
```

你应该同样在 `CustomerRow` 中设置这些特性。

在这里，`CascadeFrom` 特性告诉市（city）编辑器，它将绑定（级联）父编辑器的 ID。

当生成这个表单时，`CountryId` 字段将被 ID 为 `CountryId` 的编辑器处理。所以我们将在 `CityId` 检索编辑器的 `CascadeFrom` 特性设置为该 ID。

`CascadeField` 决定城市过滤器所在的字段。因此，当国家编辑器的值改变时，城市编辑器的项目也将被 `CountryId` 过滤，如：

```
this.Items.Where(x => x.CountryId == CountryEditorValue)
```

如果 `CascadeFrom` 和 `CascadeField` 特性是一样的，你只需要指定 `CascadeFrom`。

如果你想在客户网格列表的筛选栏添加这些级联编辑器，则需要在 `CustomerGrid.cs` 的 `CreateToolbarExtensions` 方法中执行此操作：

```
AddEqualityFilter<LookupEditor>("CountryId",
    options: new LookupEditorOptions
    {
        LookupKey = "MyModule.Country"
    });

AddEqualityFilter<LookupEditor>("CityId",
    options: new LookupEditorOptions
    {
        LookupKey = "MyModule.City",
        CascadeFrom = "CountryId",
        CascadeField = "CountryId"
    });

AddEqualityFilter<LookupEditor>("DistrictId",
    options: new LookupEditorOptions
    {
        LookupKey = "MyModule.District",
        CascadeFrom = "CityId",
        CascadeField = "CityId"
    });
}
```

在这里，我假设在 CustomerRow 有 CountryId、CityId 和 DistrictId 字段。

现在你的编辑和过滤都有可用的多级编辑器。

# 如何使用验证码？

要在登录窗体中使用验证码，请按照这些步骤操作：

要求 Serenity 1.8.5+

你可能在其他窗体也使用验证码功能，但这里以登录为例。

首先，你需要到 Recaptcha 注册一个新网站：

<https://www.google.com/recaptcha/admin>

一旦你有了站点密钥（site key）和 安全密钥（secret key），就可在 web.config/appSettings 中配置：

```
<add key="Recaptcha" value="{
    SiteKey: '6LeIxAcTAAAAAJcZVRqyHh71UMIEGNQ_MXjiZKhI',
    SecretKey: '6LeIxAcTAAAAAGG-vFI1TnRWxMZNFUojJ4WifJWe' }" />
```

上面列出的密钥仅供测试，请不要将其应用到实际产品。

编辑 LoginForm.cs，并添加 Recaptcha 属性：

```
public class LoginForm
{
    [Placeholder("default username is 'admin'")]
    public String Username { get; set; }
    [PasswordEditor, Placeholder("default password for admin user
r is 'serenity'"), Required(true)]
    public String Password { get; set; }
    [DisplayName(""), Recaptcha]
    public string Recaptcha { get; set; }
}
```

编辑 LoginRequest.cs，并添加 Recaptcha 属性：

```
public class LoginRequest : ServiceRequest
{
    public string Username { get; set; }
    public string Password { get; set; }
    public string Recaptcha { get; set; }
}
```

编辑 AccountPage.cs 的 Login 方法，在此对验证码进行验证：

```
[HttpPost, JsonFilter]
public Result<ServiceResponse> Login(LoginRequest request)
{
    return this.ExecuteMethod(() =>
    {
        request.CheckNotNull();

        if (string.IsNullOrEmpty(request.Username))
            throw new ArgumentNullException("username");

        var username = request.Username;

        // just add line below
        Serenity.Web.RecaptchaValidation.Validate(request.Recaptcha);

        if (WebSecurityHelper.Authenticate(ref username, request.Password, false))
            return new ServiceResponse();

        throw new ValidationError("AuthenticationError",
            Texts.Validation.AuthenticationError);
    });
}
```

# 如何在 **Serene** 中注册权限？

Serene 在用户和角色权限对话框中显示权限列表，若要在这里显示你自己的权限，需要在你自己的权限中使用下面的特性：

- 继承自 `PermissionAttributeBase` 的特性：
  - `ReadPermission`
  - `ModifyPermission`
  - `InsertPermission`
  - `UpdatePermission`
  - `DeletePermission`
- `Page` 和 `Endpoint` 访问控制特性：
  - `PageAuthorize`
  - `ServiceAuthorize`

这些特性可以在下面类型中使用：

- `XYZRow` (可设置 `Read`, `Write`, `Insert`, `Update`, `Delete` 权限)
- `XYZPage` 及操作方法 (`PageAuthorize`)
- `XYZEndpoint` 及服务操作 (`ServiceAuthorize`)

当你在这些属性中使用访问许可键 (permission key)，Serene 将在应用程序启动时使用反射自动检测它们。

`PermissionKeys` 是 Serene 的一个类。一些用户希望在该类中也能检测到自己写的访问许可键。

但是，`PermissionKeys` 类只是为了智能提示，它被 Serene 忽略。

如果你不使用其中任何的访问许可键，但仍然想在权限对话框中显示它，你可以在程序集中使用 `RegisterPermission` 特性（可在 `YourProject.Web` 任何地方使用）：

```
[assembly: Serenity.ComponentModel.RegisterPermissionKey("MySpec  
ialPermissionKey")]
```

## 组织权限树

若要在树层次结构中创建权限，在你的访问许可键中使用冒号（:）作为分隔符：

- MyModule:SubModule:General
- MyModule:SubModule:Permission1
- MyModule:SubModule:Permission2

这些键会在 **MyModule / SubModule** 类别下显示。因此它们的类别是：

- MyModule:SubModule:

| 类别键(Category keys) 以冒号结束，不要使用以冒号结尾的访问许可键。

请不要使用匹配类别键的访问许可键。如果你使用这样的键，如 **MyModule:SubModule**，它不会在 **MyModule / SubModule** 类别下显示而会在同一级别中显示。

如果你需要在类别中使用一个通用权限 (generic permission)，可以使用 **MyModule:SubModule:General**。

| *General* 没有特殊的含义，若你喜欢，也可以使用 Common，Module，View。

## 处理类别的显示文本

因为类别由访问许可键自动决定，可能没有友好的显示文本。

你需要使用本地化系统为它们添加显示文本。

| 如果你不需要本地化，只需要在 `site.texts.invariant.json` 添加文本。

例如，在 `site.texts.invariant.json` 文件中有这样的键值对：

```
"Permission.Administration": "Administration",
"Permission.Administration:Security": "User, Role Management and Permissions",
"Permission.Administration:Translation": "Languages and Translations",
"Permission.Northwind:Customer": "Customers",
"Permission.Northwind:Customer:View": "View",
"Permission.Northwind:Customer:Delete": "Delete",
"Permission.Northwind:Customer:Modify": "Create/Update",
"Permission.Northwind:General": "[General]"
```

以冒号结尾的键，如 *Administration:* 和 *Customer:* 对应类别显示的文本。

你需要将类别文本添加到固定的语言。如果你想要本地化，也可以添加其他语言。

## 如何在 **Serenity** 中使用第三方插件？

在 Serenity 应用程序使用第三方/自定义插件没有涉及特别的步骤。你可以在 `_LayoutHead.cshtml` 包括第三方脚本和 CSS 并遵照官方文档的步骤。

尤其是如果你使用 TypeScript，更没有涉及特别的步骤。

如果使用 Saltaralle（已经被弃用），则可能必须编写一些导入类（import classes）或动态使用这些类。

但是，如果你希望该组件与 Serenity 编辑对话框更好地工作，你可以尝试把它封装成 Serenity 部件。

在这里，我们将以 Bootstrap 的多选插件作为示例，将其集成到 Serenity，使其类似于 `LookupEditor`。

这是 Bootstrap 多选组件的文档和示例：

<http://davidstutz.github.io/bootstrap-multiselect/>

### 获取脚本和 CSS 文件

首先，我们应该下载组件的脚本和 CSS 文件，并把它们分别放在 `MyProject.Web/scripts/` 和 `MyProject.Web/content` 文件夹下。

该组件有 NuGet 程序包，但是不幸的是它并不是按我们项目标准的形式安装（它不会将文件放到项目文件夹），所以我们要手动下载文件。

下载脚本文件，并把它放到 `MyProject.Web/Scripts` 下：

<https://raw.githubusercontent.com/davidstutz/bootstrap-multiselect/master/dist/js/bootstrap-multiselect.js>

下载 CSS 文件，并把它放到 `MyProject.Web/Content` 下：

<https://raw.githubusercontent.com/davidstutz/bootstrap-multiselect/master/dist/css/bootstrap-multiselect.css>

### 在 `_LayoutHead.cshtml` 包含脚本/样式文件

根据该插件的文档，我们应该包含这些文件：

```
<!-- Include the plugin's CSS and JS: -->
<script type="text/javascript"
    src="js/bootstrap-multiselect.js">
</script>
<link rel="stylesheet" type="text/css"/
    href="css/bootstrap-multiselect.css" />
```

打开 MyProject.Web/Views/Shared 下的 \_LayoutHead.cshtml，并包含这些文件：

```
// ...
@Html.Stylesheet("~/Content/bootstrap-multiselect.css")
@Html.Stylesheet("~/Content/serenity/serenity.css")
@Html.Stylesheet("~/Content/site/site.css")
// ...
@Html.Script("~/Scripts/bootstrap-multiselect.js")
@Html.Script("~/Scripts/Site/Serene.Script.js")
@Html.Script("~/Scripts/Site/Serene.Web.js")
```

## 创建 **BSMultiSelectEditor.ts**

现在，我们需要一个 TypeScript 源文件来放置组件。我们可以把它放在 MyProject.Web/Scripts 或者 MyProject.Web/Modules 目录下面。

我将在 MyProject.Web/Modules/Common/Widgets（你需要先创建 Widgets 文件夹）下创建该文件。

创建 BSMultiSelectEditor.ts 文件：

```
namespace MyProject {
    @Serenity.Decorators.element("<select/>")
    @Serenity.Decorators.registerClass(
        [Serenity.IGetEditValue, Serenity.ISetEditValue])
    export class BSMultiSelectEditor
        extends Serenity.Widget<BSMultiSelectOptions>
        implements Serenity.IGetEditValue, Serenity.ISetEditValue {
        constructor(element: JQuery, opt: BSMultiSelectOptions) {
            super(element, opt);
        }

        public setEditValue(source: any,
            property: Serenity.PropertyItem): void {
        }

        public getEditValue(property: Serenity.PropertyItem,
            target: any): void {
        }
    }

    export interface BSMultiSelectOptions {
        lookupKey: string;
    }
}
```

这里，我们定义了一个新的继承自 `Widget` 的编辑器类型。我们的小部件使用 `BSMultiSelectOptions` 类型选项，且类似于 `LookupEditor`，包含 `lookupKey` 选项。它还实现了 `IGetEditValue` 和 `ISetEditValue` 的 TypeScript 接口（这不同于 C# 接口）。

```
@Serenity.Decorators.element("<select/>")
```

在上面这行代码，我们指定小部件与 `SELECT` 元素一起工作，因为该 bootstrap 多选插件也要求一个 `select` 元素。

```
@Serenity.Decorators.registerClass(  
    [Serenity.IGetEditValue, Serenity.ISetEditValue])
```

上面，我们采用 Saltaralle 类型系统注册 TypeScript 类，并指定小部件实现自定义值的 `getter` 和 `setter` 方法，这些方法对应于 `getEditValue` 和 `setEditValue`。

这里的语法有点简洁，因为我们必须处理 Saltaralle 和 TypeScript 之间的互操作。

我们的 `constructor`、`getEditValue` 和 `setEditValue` 方法还是空的，我们待会将填充它们。

## 使用我们的新编辑器

现在，生成你的项目并转换模板。

打开 `CustomerRow.cs` 并定位到 `Representatives` 属性：

```
[LookupEditor(typeof(EmployeeRow), Multiple = true), ClientSide]  
[LinkingSetRelation(typeof(CustomerRepresentativesRow),  
    "CustomerId", "EmployeeId")]  
public List<Int32> Representatives  
{  
    get { return Fields.Representatives[this]; }  
    set { Fields.Representatives[this] = value; }  
}
```

我们在这里可以看到 `Representatives` 使用 `Multiple` 值为 `true` 的 `LookupEditor`。我们把它替换成新的编辑器：

```
[BSMultiSelectEditor(LookupKey = "Northwind.Employee"), ClientSide]
[LinkingSetRelation(typeof(CustomerRepresentativesRow),
    "CustomerId", "EmployeeId")]
public List<Int32> Representatives
{
    get { return Fields.Representatives[this]; }
    set { Fields.Representatives[this] = value; }
}
```

## 使用检索项目填充编辑器

如果你现在生成项目，并打开客户对话框，你将看到 Customer representatives 字段的选项为为空。

让我们先用数据填充它：

```
export class BSMultiSelectEditor {
    constructor(element: JQuery, opt: BSMultiSelectOptions) {
        super(element, opt);

        let lookup = Q.getLookup(this.options.lookupKey) as Q.Lookup<any>;
        for (let item of lookup.getItems()) {
            let key = item[lookup.getIdField()];
            let text = item[lookup.getTextField()] || '';
            Q.addOption(element, key, text);
        }
    }
}
```

我们首先获得由 *lookupKey* 选项指定检索的对象。

检索（Lookups）有 *idField* 和 *textField* 属性，通常对应的字段由检索行（lookup row）上的 *IIdRow* 和 *INameRow* 接口决定。

我们在 *lookup* 枚举所有的项目并使用 *idField* 和 *textField* 属性确定这些项的 *key* 和 *text* 属性。

现在保存文件，并再次打开客户对话框，这次你将看到选项被填充了。

## Bootstrap Multi Select Typings

根据文档，现在应该在我们的元素中调用 jQuery 的扩展方法 ".multiselect()"。

我会把 SELECT 元素转换为 `<any>` 并在其上调用 `.multiselect`。但是我想使用 TypeScript 写一个定义文件 `.d.ts` 以重用含智能感知的多选。

因此，在 `MyProject.Web/Scripts/typings/bsmultiselect` 文件夹下，创建文件 `bsmultiselect.d.ts`：

```
interface JQuery {
    multiselect(options?: BSMultiSelectOptions | string): JQuery
}

interface BSMultiSelectOptions {
    multiple?: boolean;
    includeSelectAllOption?: boolean;
    selectAllText?: string;
    selectAllValue?: string | number;
}
```

在这里，我扩展了原属于 jQuery 自身且被定义在 `jquery.d.ts` 的 `JQuery` 接口。在 TypeScript 中，你可以用新的方法扩展任何接口、属性等。

我使用插件文档的方法定义 `BSMultiSelectOptions`。该插件其实有更多的选项，但为了演示，我尽量保持简单。

## 在我们的编辑器中创建 Bootstrap 多选

现在，我将回到我们的构造函数并使用多选插件初始化它：

```
export class BSMultiSelectEditor {
    constructor(element: JQuery, opt: BSMultiSelectOptions) {
        super(element, opt);

        element.attr('multiple', 'multiple')

        let lookup = Q.getLookup(this.options.lookupKey) as Q.Lookup<any>;
        for (let item of lookup.getItems()) {
            let key = item[lookup.getIdField()];
            let text = item[lookup.getTextField()] || '';
            Q.addOption(element, key, text);
        }

        element
            .attr('name', this.uniqueName + "[]")
            .multiselect();
    }
}
```

打开客户对话框，你将看到 Representatives 已经有 Bootstrap 多选编辑器。

## 处理 **GetEditValue** 和 **SetEditValue** 方法

如果我们没有处理这些方法，Serenity 不会知道如何读取或设置编辑器的值，因此即使你选择了一些代表（representatives），下一次你再打开对话框，也会得到空的选项。

```
export class BSMultiSelectEditor {  
    //...  
  
    public setEditValue(source: any, property: Serenity.PropertyItem): void {  
        this.element.val(source[property.name] || []).multiselect('refresh');  
    }  
  
    public getEditValue(property: Serenity.PropertyItem, target: any): void {  
        target[property.name] = this.element.val() || [];  
    }  
}
```

`setEditValue` 在编辑器值需要被设置时调用。它接受一个源 (`source`) 对象，该对象通常是对话框中加载的实体。

`Property` 参数是包含有关正在处理字段详细信息的属性项目对象，如 `Representatives` 属性。它的 `name` 字段包含字段名称，如 `Representatives`。

这里，我们在设置所选值后调用 `multiselect('refresh')`，因为多选插件不知道选项什么时候会被更改。

`getEditValue` 正好相反。它会读取编辑值并将其设置到目标实体。

Ok，现在我们的客户编辑器可以很好地工作了。

# 如何启用脚本合并（Script Bundling）？

在 Serene 模板中，\_LayoutHead.cshtml 默认包含了 3MB+ 的 javascript 引用文件。

这可能导致系统出现一些带宽和性能问题，特别是使用移动设备访问基于 Serenity 的站点。

有几种方式可以处理这些问题，如压缩文件以减少脚本大小；合并打包脚本到较少的文件，从而减少请求的数量。

你可能更喜欢使用工具，如 Webpack、Grunt、Gulp、UglifyJS 等，但如果想要更简单有效且更少手工操作的解决方案，Serenity 自带的脚本合并和压缩系统可开箱即用。

请注意，该功能要求 Serenity 2.0.13+。

## ScriptBundles.json

首先，你需要在 `MyProject.Web/scripts/site` 文件夹下添加一个文件 `ScriptBundles.json`。ScriptBundles.json 配置启用合并时，合并脚本将包含这些文件。

在 Serene 模板 2.0.13+ 中默认包含该文件，它看起来像：

除非你想要向包中添加一些自定义的脚本，否则你不需要修改此文件。

```
{  
  "Libs": [  
    "~/Scripts/pace.js",  
    "~/Scripts/rsvp.js",  
    "~/Scripts/jquery-{version}.js",  
    "~/Scripts/jquery-ui-{version}.js",  
    "~/Scripts/jquery-ui-i18n.js",  
    "~/Scripts/jquery.validate.js",  
    "~/Scripts/jquery.blockUI.js",  
    "~/Scripts/jquery.cookie.js",  
    "~/Scripts/jquery.json.js",  
    ...]  
}
```

```
"~/Scripts/jquery.autonumeric.js",
"~/Scripts/jquery.colorbox.js",
"~/Scripts/jquery.dialogextendQ.js",
"~/Scripts/jquery.event.drag.js",
"~/Scripts/jquery.scrollintoview.js",
"~/Scripts/jsrender.js",
"~/Scripts/select2.js",
"~/Scripts/toastr.js",
"~/Scripts/SlickGrid/slick.core.js",
"~/Scripts/SlickGrid/slick.grid.js",
"~/Scripts/SlickGrid/slick.groupitemmetadataprovider.js"

',
"~/Scripts/SlickGrid/Plugins/slick.autotooltips.js",
"~/Scripts/SlickGrid/Plugins/slick.headerbuttons.js",
"~/Scripts/SlickGrid/Plugins/slick.rowselectionmodel.js"

',
"~/Scripts/SlickGrid/Plugins/slick.rowmovemanager.js",
"~/Scripts/bootstrap.js",
"~/Scripts/Saltarelle/mscorlib.js",
"~/Scripts/Saltarelle/linq.js",
"~/Scripts/Serenity/Serenity.CoreLib.js",
"~/Scripts/Serenity/Serenity.Script.UI.js",
"~/Scripts/Serenity/Serenity.Externals.js",
"~/Scripts/Serenity/Serenity.Externals.Slick.js",
"~/Scripts/jquery.cropzoom.js",
"~/Scripts/jquery.fileupload.js",
"~/Scripts/jquery.iframe-transport.js",
"~/Scripts/jquery.slimscroll.js",
"~/Scripts/mousetrap.js",
"~/Scripts/fastclick/fastclick.js"

],
"Site": [
"~/Scripts/adminlte/app.js",
"~/Scripts/Site/Serene.Script.js",
"~/Scripts/Site/Serene.Web.js"
]
}
```

在这里我们定义两个不同的合并配置，**Libs** 和 **Site**，对应于动态脚本文件：

*Bundle.Site.js* 配置为包含这三个 JS 文件（在列出的顺序）：

```
"~/Scripts/adminlte/app.js",
~/Scripts/Site/Serene.Script.js",
~/Scripts/Site/Serene.Web.js"
```

而 *Bundle.Libs.js* 包含所有其他 javascript 文件。

默认情况下，我们这里使用 2 个合并包（bundles），但如果你需要不同的配置，只要注意依赖关系，也可以使用一个、三个或更多的合并包。

在这里，合并包（bundle）内的顺序是非常重要的。必须按在 *\_LayoutHead.cshtml* 出现的顺序包含脚本。

在你添加另外的客户脚本时，请确保把它放在所有依赖脚本后面。

例如，如果在 jquery 前加载 jquery 插件，你将得到错误。

同时，也要确保在两个合并包（bundles）中不能包含同一文件。

## Enabling Bundling

你应该只在生产发布时启用合并（特别是压缩）。否则 Javascript 将会难以调试。

为了启用合并，只需要把 *web.config* 中 *ScriptBundling* 的 *Enabled* 属性改为 *true*：

```
<add key="ScriptBundling" value=
  { Enabled: true, Minimize: false, UseMinJS: false }" />
```

当 *Enabled* 是 *false*（默认情况）时，系统不会做任何事，并且你包含的脚本是未经压缩的。你的页面源代码看起来是这样的：

```
<script src="/Scripts/pace.js?v=..."></script>
<script src="/Scripts/rsvp.js?v=..."></script>
<script src="/Scripts/jquery-2.2.3.js?v=..."></script>
<script src="/Scripts/jquery-ui-1.11.4.js?v=..."></script>
<script src="/Scripts/jquery-ui-i18n.js?v=..."></script>
...
...
...
<script src="/Scripts/adminlte/app.js?v=..."></script>
<script src="/Scripts/Site/Serene.Script.js?v=..."></script>
<script src="/Scripts/Site/Serene.Web.js?v=..."></script>
...
```

当 *Enabled* 为 true 时，脚本将被合并成为：

```
<script src="/DynJS.axd/Bundle.Libs.js?v=..."></script>
<script src="/DynJS.axd/Bundle.Site.js?v=..."></script>
...
```

这两个包在内存中生成，并包含所有在 *ScriptBundles.json* 文件配置的脚本。

他们同样使用 GZIP 压缩并在内存中缓存（以压缩的形式），因此我们的脚本将消耗更少的带宽和服务请求。

现在，我们的脚本文件压缩到 600KB，而不是之前的 3000KB，减少了 80% 的大小。

## Enabling Minification

在启用合并之后，你也可以在 *web.config* 设置中启用脚本压缩，把 *Minimize* 属性设置为 true:

```
<add key="ScriptBundling" value="
  { Enabled: true, Minimize: true, UseMinJS: false }" />
```

请注意，*Minimize* 属性只有同时启用合并属性才能工作。

UglifyJS 库是用来压缩 js 文件。它被用在 合并/压缩 之前，因此我们的合并将会减少 40%，但是更难以阅读，因此只在生产发布时启用压缩设置。

现在我们合并及压缩后的脚本文件将变为 375 KB，而不是之前的 3000 KB，与初始大小相比降低了 87% 或 1/8。

## UseMinJS Setting

压缩操作将花点时间，并且第一次请求站点时，根据服务器的速度，可能需要花费 5-40 秒甚至更长时间。

由于压缩只在应用程序启动时执行一次，所以不会影响其他请求。

无论如何，如果在第一个请求时，你还需要更高的性能，你可以要求 Serenity 重用在磁盘上可用的已压缩文件。

把 `UseMinJS` 设置为 `true`：

```
<add key="ScriptBundling" value="
    { Enabled: true, Minimize: true, UseMinJS: true }" />
```

当该设置为 ON 时，在压缩文件前，例如 `jquery-ui-1.11.4.js`，Serenity 将首先检查 `jquery-ui-1.11.4.min.js` 文件是否存在磁盘上。如果存在，将直接使用该文件，而不再使用 UglifyJS 压缩文件。否则它将运行 UglifyJS 压缩文件。

Serene 包含所有脚本库的压缩版本，也包括 Serenity 的脚本，因此该设置将加速初始启动时间。

还有一个你应该小心避免的风险。如果你手动修改脚本库，请确保你手动压缩并同时修改它的 `.min.js` 文件，否则启用合并时，生产环境可能会运行旧版本的脚本。

## Serenity 如何修改 `_LayoutHead.cshtml` 包含的脚本文件？

如果查看 `_LayoutHead.cshtml`，你可能会注意到文件中的这些代码：

```
@Html.Script("~/Scripts/jquery.cropzoom.js")
@Html.Script("~/Scripts/jquery.fileupload.js")
@Html.Script("~/Scripts/jquery.iframe-transport.js")
```

当禁用合并时，这些声明生成如下 HTML 代码：

```
<script src="/Scripts/jquery.cropzoom.js"></script>
<script src="/Scripts/jquery.fileupload.js"></script>
<script src="/Scripts/jquery.iframe-transport.js"></script>
```

Html.Script 是 Serenity 的扩展方法，因此当开启合并时，Serenity 将首先检查合并内容是否包含该脚本，而不是生成 HTML 代码。

假设合并包含 Bundle.Lib.js 脚本，Serenity 将生成下面的代码：

```
<script src="/DynJS.axd/Bundle.Libs.js?v=... "></script>
```

但是，对于同一合并中包括其他 Html.Script 调用，Serenity 将不会生成任何东西。因此，即使你调用 Html.Script 50 次，在页代码中只会得到一个 `<script>` 的输出。

### 脚本标签中的 v=p53uqJ... 是什么？

这是脚本的版本号或哈希值。不管是否启用合并，当你使用 Html.Script 时，都会在包含的脚本中添加这些哈希值。该哈希值允许浏览器缓存脚本直到哈希值发生变化。当脚本内容更改时，其哈希值也会变化，所以浏览器不会缓存并使用较旧版本的脚本。

这就是使用 Serenity 应用程序不会有脚本缓存问题的原因。

## 常见问题

### 代码生成器 (**Sergen**)

在表中添加新的字段之后，我应该重新生成代码吗？

建议只生成一次代码，你应该以现有字段为例，在行 (row) 、列 (column) 和表单 (form) 类手动添加新字段。

但是如果你做了太多的更改，并且再次生成代码。**Sergen** 将启动 **Kdiff3** 让你合并更改，以便不会覆盖你之前对生成代码所做的更改。

---

在 **Sergen** 中，我有关于 **KDiff3** 的错误，如何定位该错误呢？

**Sergen** 在其默认程序文件目录下查找 **KDiff3**。如果你还没有该程序，请安装它。

如果 **Kdiff3** 在另一位置，编辑你的解决方案目录下的 **Serenity.CodeGenerator.config**。这是一个包含 **Sergen** 设置和首选项的 JSON 文件。

## Permissions

如何从 **UPDATE** 权限中分离出 **INSERT** 权限？:

在行 (rows) 中使用 [**InsertPermission**] 和 [**UpdatePermission**] 替代 [**ModifyPermission**] 特性。

默认情况下，插入、保存处理程序 (**INSERT, save handler**) 按如下顺序检索行中的权限：

- 1) **InsertPermission**
- 2) **ModifyPermission**
- 3) **ReadPermission**

只有第一个被选中的权限才会被找到。

类似的，更新、保存处理程序（UPDATE, save handler）按如下顺序检索行中的权限：

- 1) UpdatePermission
- 2) ModifyPermission
- 3) ReadPermission

删除处理程序（delete handler）按如下顺序检索行中的权限：

- 1) DeleteInsertPermission
- 2) ModifyPermission
- 3) ReadPermission

列表/检索处理程序（LIST / RETRIEVE handler）只有一个权限被选中：

- 1) ReadPermission

## 发布和部署

怎样发布 **Serenity** 应用程序？

Serenity 应用程序使用 x-copy 部署。在部署之后，你只需要设置安装程序的连接字符串。你也可以从部署中排除源文件。

请确保在 `SiteInitialization.Migrations` 文件的 `RunMigrations` 方法中删除了数据库迁移安全检查。

你也可以使用 Visual Studio 的发布功能。但需保证所有内容文件（content files）的生成操作设置为 `Content`（而不 `None`）。

你只需要发布 `MyApplication.Web` 项目，不需要发布 `script` 项目。

Serenity 使用 ASP.NET MVC 的 NuGet 版本。因此无需在服务器安装 MVC。如果你得到一些缺少 DLL 的错误，请检查 VS 项目引用的 复制到本地 选项是否设置为 `True`。

## 表单 和 编辑器（Editors）

如何在 **DecimalEditor** 中允许负值？

在 *DecimalEditor* 特性设置 *MinValue* 和 *.MaxValue* 属性：

```
[DecimalEditor(MinValue = "-999999999.99", MaxValue = "999999999  
.99")]  
public Decimal MyProperty { get; set; }
```

请确保最大值和最小值使用相同的位数。

如何重新加载/刷新检索编辑器数据？

使用 *Q.ReloadLookup("MyModule.MyLookupKey")* 重新加载检索其 *key* 的结果。

如何为枚举创建过滤编辑器？

```
AddEqualityFilter<EnumEditor>(SomeRow.Fields.TheEnumField,  
    options: new EnumEditor { EnumKey = "MyModule.MyEnumType" })  
;
```

如何在新记录模式下把日期编辑器设置为当前日期？

在表单定义中为日期时间编辑器的日期添加 *[DefaultValue("today")]* 或者 *[DefaultValue("now")]*。

请不要在行 (*row*) 中添加这些特性，否则将导致错误。

另一做法是在对话框中处理，重写 *AfterLoadEntity*：

```
form.MyDateField.AsDate = JsDate.Today;
```

## 故障排除

### 初始设置

在创建并启动新的 **Serene** 应用程序之后，不能显示登录页，而当你打开浏览器控制台，却得到一条错误消息：找不到 **Template.LoginPanel**：

你可能使用了无效的解决方案名称，如 `MyProject.Something`（包含点`.`）。

当项目以这种方式命名时，模板系统将不能定位模板。

请不要在解决方案名称中使用点符号（`.`），如果必须使用点符号，可在创建解决方案之后再重命名。

### 编译错误

在 **Script** 项目添加文件后，得到几个引用不明确的错误：

从 `script` 项目移除 `System.dll` 引用。Visual Studio 在使用 `添加新文件` 对话框时会添加该引用，而 Saltarelle 编译器不能与该程序集一起工作，因为它有完全不一样的运行时。

在 `Script` 项目中，请使用复制/粘贴的方式创建代码文件。

**Error: System.ComponentModel.DisplayName attribute exists in both  
...\\Serenity.Script.UI.dll and ...\\v2.0...\\System.dll**

与上面一样，从 `Script` 项目移除 `System.dll` 引用。

### 运行时错误

当上传文件或添加备注（**notes**）时，得到 **NotImplementedException** 的异常：

该功能需要表的标识列是整数。在最近版本的 Serenity 中，已添加对 String/Guid 主键的支持，但一些旧行为（old behaviors）不能与这种键一起工作。

### SQL 和连接字符串

当我在网格列表切换页时，得到错误：“**OFFSET**”附近语法错误，在 **FETCH** 声明中使用无效 **NEXT** 项”：

你的 SQL server 是 2008 或更旧的版本。默认情况下，SQL Server 连接使用 SQL2012 方言。在 SiteInitialization.cs 中把连接修改为 SqlServer2005 或 SqlServer2008 方言，如：

```
SqlConnections.GetConnectionString("Default").Dialect =
SqlServer2008Dialect.Instance;
```

## T4 模板问题

在转换模板后，发现枚举不能被转换到脚本端：

如果你在行（row）或者请求/响应服务中使用枚举类型，枚举会被转换，否则默认情况下，枚举不会被转换。如果此时你需要包含该枚举，请向其添加 [ScriptInclude] 特性。

## 编辑器和表单

试图设置级联下拉列表，但第二个下拉列表始终为空：

确保你正确地设置了 CascadeField，CascadeField 应该匹配第二检索的属性名称。例如，CountryID 在脚本端没有匹配 CountryId。你可以使用 nameof() 操作确认，如 CascadeField = nameof(CityRow.CountryId)。

如果你错误地设置 CascadeFrom 项，也会发生类似的问题。它对应于你表单的第一个下拉列表 ID。例如，如果在表单中有 MyCountryId 和 CustomerCityId 属性，CascadeFrom 应该是 CustomerCountryId。同样，你也可以使用 nameof(MyCountryId) 确认。

CascadeFrom 是表单的编辑器 ID，而 CascadeField 是行中的字段属性名称。

另一种可能性是 CascadeField 没有包括在发送到脚本端的检索数据中。例如，如果第二个下拉列表是城市，通过 CountryId 与国家下拉列表对应，请确保 CityRow 的 CountryId 属性有 [LookupInclude] 特性。默认情况下，只有 ID 和 Name 属性被发送到脚本端进行检索。

试图使用对话框模板创建选项卡，但是选项卡不能显示或是空的：

确保你没有把选项卡内容放到另一选项卡内容中，如 选项卡 DIV 包含在其他选项卡的 DIV 内。

## 主/从 编辑

在内存中创建类似于 **Movie** 教程的演员编辑器的主从信息，但是更新记录时，我得到两条重复的实体：

确保 **EditDialog** 类没有 **[IdProperty]**。因为编辑对话框的记录是在内存中操作，此时该记录还没有实际的 ID，如果你对它们使用实际的 ID 属性，对话框会认为你在编辑时要添加新记录（由于它们的实际 ID 总为 null）。

正如你看到下面的代码，**GridEditorDialog** 基类使用伪造 ID：

```
[IdProperty("___id")]
public abstract class GridEditorDialog<TEntity> : EntityDialog<TEntity>
{
    where TEntity : class, new()
```

因此，当你在编辑对话框中设置 **[IdProperty]**，你是在重写该伪造 ID 并导致不可预料的行为。

在成功地添加详细信息后，打开一个现有记录，发现某些可视字段 (**view fields**) 为空：

请在 **YourDetailRow.cs** 的可视字段添加 **[MinSelectLevel(SelectLevel.List)]** 特性。默认情况下，List handlers 和 MasterDetailBehavior 只加载详细行的表字段（没有可视字段）。

## 权限

页面不会显示在导航中

Page 的访问权限是读取 XYZPage.cs 文件 Index 操作的 `PageAuthorize` 特性，这是一个 MVC 页面控制器。请确保你将此设置为用户权限。

---

已经在 **PermissionKeys.cs** 添加了权限，但是不能在权限对话框显示它：

**PermissionKeys** 类仅作为智能感知的目的。更多关于注册 keys 的信息请查看下面连接。

- 如何在 Serene 中注册权限？
- 

在 **row** 中修改访问许可键（**permission keys**），但是当打开页面时得到错误且没有显示记录：

XYZEndpoint.cs 也需要有 `[ServiceAuthorize("SomePermission")]` 特性。这是为了提供中等级别的安全（secondary level security）。在 Row 中替换访问键。

## 本地化

发布网站之后，本地化不能用：

把翻译的文件保存到 `~/App_Data` 目录下。将这些文件复制到服务器，或将相关文本文件移动到 `~/Scripts/site/texts` 下。

---

已经添加了自定义本地文本键，但是不能在脚本端访问它们：

不是所有的翻译都会转换到脚本端。在 `web.config` 有一个控制着这些前缀的 `LocalTextPackages` 键设置。如果查看该配置，你可以看到只有以 `Db.`、`Dialogs.`、`Forms.` 等开头的文本键会被转换到客户端。这是为了限制大小的文本，因为不是所有的脚本代码都会使用这些文本。

请在该配置中添加你自己的前缀，或者把你的文本键更改为以其中的默认前缀开头。

---

## NuGet 程序包 和 升级

更新 **Select2** 后，得到一些错误：

请不要把 Select2 更新到 3.5.1+ 的版本，它最近的版本有一些已知的兼容性问题。

把 Select2 还原到 Select2 3.5.1，请在包管理器控制台输入下面指令：

```
Update-Package Select2.js -Version 3.5.1
```

## 部署与发布

发布项目之后，一些内容找不到或不能显示：

如果你使用 Visual Studio 发布，确保 css、图片文件等被包含在 web 项目，并且把它们的生成操作属性设置为 内容。

另一种可能是 IIS\_IUSRS 用户组不能访问文件。检查其是否有访问 web 文件夹（发布的文件夹）权限。

---

发布之后，找不到表（如 **User**）：

Serene 有一个避免在任意的数据库上运行迁移的检查。在 *SiteInitialization.Migrations* 文件的 *RunMigrations* 方法中可找到该检查，删除该检查。

---

得到“不能设置常量字段”的 **FieldAccessExceptions**:

托管服务提供商已将你的 web 应用程序池设置为中等信任安全等级。请要求他们授予高度信任安全等级，或考虑更换供应商。

如果你的托管服务提供商没有锁住 web.config，也可以在 web.config 设置：

```
<configuration>
  <system.web>
    <trust level="Full" />
  </system.web>
</configuration>
```

Serenity 使用反射初始化字段对象。在中等信任安全等级时，它不能初始化对象。你可以尝试把 *Row.cs* 所有的 *public readonly*\* 字段定义替换为 *public static\**，但是不确定这样能解决所有问题。

ASP.NET 的中等信任安全等级已经过时了，并且微软不会解决与此相关的任何问题。详见 <http://stackoverflow.com/questions/16849801/is-trying-to-develop-for-medium-trust-a-lost-cause>

强烈建议你更换托管服务提供商。

## 服务定位器(**Service Locator**) & 初始化

Serenity 使用服务定位器模式 (service locator pattern) 抽象其依赖，以使其可以与你所选的库 (libraries) 和服务提供者 (service providers) 一起工作。

例如，Serenity 并不关心你是如何存储用户，但是它可以通过抽象 (IAuthorizationService、IUserRetrieveService 等) 查询当前用户。

与你在应用程序中使用 Redis、Couchbase、Memcached 或其他分布式缓存类似。只要实现 IDistributedCache 接口，并向其注册服务定位器，Serenity 便开始使用 NoSQL 数据库。

有些人可能认为服务定位器是一种反模式，应该避免使用该模式，并把它替换为依赖注入模式。但服务定位器模式不需要了解对象的每个依赖（及依赖的依赖……），就可以使用它（你不必知道移动运营商如何发送短信的细节）。也许 DI 是过度设计 (over-engineering) 的示例。

# 静态依赖类

[命名空间: *Serenity.Abstractions*, 程序集: *Serenity.Core*]

依赖类是 Serenity 的服务定位器 (service locator)。所有依赖都通过其方法进行查询：

```
public static class Dependency
{
    public static TType Resolve<TType>() where TType : class;
    public static TType Resolve<TType>(string name) where TType
        : class;
    public static TType TryResolve<TType>() where TType : class;
    public static TType TryResolve<TType>(string name) where TType
        : class;

    public static IDependencyResolver SetResolver(IDependencyRes
olver value);
    public static IDependencyResolver Resolver { get; }
    public static bool HasResolver { get; }
}
```

在你应用程序的启动方法 (如: global.asax.cs) 中, 你应该使用 SetResolver 方法设置依赖解析器 (IDependencyResolver) 的实现 (IoC 容器) 来初始化服务定位器。

## Dependency.SetResolver 方法

配置要使用的依赖解析器的实现。

你可以使用你喜欢的 IoC 容器, 但 Serenity 已经包含了基于 Munq 的 IoC 容器：

```
var container = new MunqContainer();
Dependency.SetResolver(container);
```

`SetResolver` 方法返回之前配置的 `IDependencyResolver` 实现。如果之前没有配置，则返回 `null`。

## Dependency.Resolver 属性

返回当前配置的 `IDependencyResolver` 的实现。

如果尚未配置，则抛出 `InvalidOperationException` 异常。

## Dependency.HasResolver 属性

如果 `IDependencyResolver` 的实现已通过 `SetResolver` 配置，则返回 `true`。否则，返回 `false`。

## Dependency.Resolve 方法

返回请求类型注册的实现。

如果被注册的类型尚未实现，则抛出 `KeyNotFoundException` 异常。

如果尚未配置依赖解析器，则抛出 `InvalidOperationException` 异常。

`Resolve` 方法的第二个重载接收一个 `name` 参数。如果不同提供者根据域（`scope`）为接口注册实现，就应使用这个重载方法。

例如，Serenity 有一个 `IConfigurationRepository` 接口，可以根据设置范围有不同的提供者。有些设置可能是 应用程序 范围（该应用程序的所有服务之间共享），而有些可能是 服务 范围（每个服务可能使用不同的唯一标识符）。

因此，要为这些范围的每个域注册 `IConfigurationRepository` 提供者，你应该像下面这样调用方法：

```
var appConfig = Dependency.Resolve< IConfigurationRepository>("Application");  
  
var srvConfig = Dependency.Resolve< IConfigurationRepository>("Server");
```

## Dependency.TryResolve 方法

该方法在功能上与 Resolve 方法相同，只是使用不同的方式实现。

如果请求类型没有注册提供者，或依赖解析器尚未配置，TryResolve 不会引发异常，而是返回 null。

# IDependencyResolver 接口

[命名空间: *Serenity.Abstractions*, 程序集: *Serenity.Core*]

此接口通常定义依赖解析器（dependency resolvers）的契约，以使 IoC 容器处理服务（services）与提供者（providers）之间的映射。

```
public interface IDependencyResolver
{
    TService Resolve<TService>() where TService : class;
    TService Resolve<TService>(string name) where TService : class;
    TService TryResolve<TService>() where TService : class;
    TService TryResolve<TService>(string name) where TService : class;
}
```

所有方法在功能上都等效于静态依赖类相应的方法。

# IDependencyRegistrar 接口

[命名空间: *Serenity.Abstractions*, 程序集: *Serenity.Core*]

依赖解析器（Dependency resolvers）应该实现 IDependencyRegistrar 接口来注册依赖：

```
public interface IDependencyRegistrar
{
    object RegisterInstance<TType>(TType instance) where TType : class;
    object RegisterInstance<TType>(string name, TType instance)
        where TType : class;
    object Register<TType, TImpl>() where TType : class where TImpl : class, TType;
    object Register<TType, TImpl>(string name) where TType : class where TImpl : class, TType;
    void Remove(object registration);
}
```

MunqContainer 和其他 IoC 容器同样依赖注册（它们都实现 IDependencyRegistrar 接口），所以你只需对它查询：

```
var registrar = Dependency.Resolve<IDependencyRegistrar>();
registrar.RegisterInstance<ILocalTextRegistry>(new LocalText
Registry());
registrar.RegisterInstance<IAuthenticationService>(...)
```

## IDependencyRegistrar.RegisterInstance 方法

注册器（Registers）是一个类型（TType，通常是一个接口）的单例，作为该类型的提供者。

```
object RegisterInstance<TType>(TType instance) where TType : class;
```

当你使用该重载方法注册一个对象实例，每当请求 `TType` 的实例时，会从依赖解析器返回你注册的该实例。这类似于 单例模式。

```
var registrar = Dependency.Resolve<IDependencyRegistrar>();
registrar.RegisterInstance<ILocalTextRegistry>(new LocalText
Registry());
```

如果已经有 `TType` 的注册，它将被重写。

此重载是注册依赖最常用的方法。

确保注册的提供者是线程安全的，因为所有线程将在同一时间使用这个实例。

`RegisterInstance` 有一个不太常用的含 `name` 参数的重载：

```
object RegisterInstance<TType>(string name, TType instance) where
TType : class;
```

使用该重载，你可以为同一接口通过不同的字符串标识（`string key`）注册不同的提供者。

例如，Serenity 有一个 `IConfigurationRepository` 接口，可以根据设置范围有不同的提供者。有些设置可能是 应用程序 范围（该应用程序的所有服务之间共享），而有些则可能是 服务 范围（每个服务可能使用不同的唯一标识符）。

因此，要为这些范围的每个域注册 `IConfigurationRepository` 提供者，你应该像下面这样调用方法：

```
var registrar = Dependency.Resolve<IDependencyRegistrar>();

registrar.RegisterInstance< IConfigurationRepository >(
    "Application", new MyApplicationConfigurationRepository());

registrar.RegisterInstance< IConfigurationRepository >(
    "Server", new MyServerConfigurationRepository());
```

当查询这些依赖时：

```
var appConfig = Dependency.Resolve< IConfigurationRepository>("Application");
// ...
var srvConfig = Dependency.Resolve< IConfigurationRepository>("Server");
// ...
```

## IDependencyRegistrar.Register 方法

不像 *RegisterInstance*，当一个类型使用这种方式注册时，每次请求该类型的提供者都会返回一个新的实例（所以每个请求获得一个唯一的实例）。

```
var registrar = Dependency.Resolve< IDependencyRegistrar>();
registrar.Register< ILocalTextRegistry, LocalTextRegistry>();
```

## IDependencyRegistrar.Remove 方法

所有 **IDependencyRegistrar** 接口的注册方法返回一个对象，你可以在以后使用该对象删除此注册。

在普通应用程序中应该避免使用这个方法，因为所有的注册应该在一个集中的位置进行，并且在应用程序的生命周期中只注册一次。但是，该方法可能对单元测试非常有用。

```
var registrar = Dependency.Resolve< IDependencyRegistrar>();
var registration = registrar.Register< ILocalTextRegistry, LocalTextRegistry>();
// ...
registrar.Remove(registration);
```

这不是撤消操作。如果你为接口 A 注册类型 C，而类型 B 已经注册了同一个接口，先前的注册会被重写且被丢失。你不能通过删除注册 C 找回以前的状态。

# MunqContainer 类

[命名空间: *Serenity*, 程序集: *Serenity.Core*]

Serenity 包含稍加修改的 Munq IoC 容器 (<http://munq.codeplex.com/>)。

MunqContainer 类实现 **IDependencyResolver** 和 **IDependencyRegistrar** 接口（所有容器都应该实现这两个接口）。

一旦你像下面这样设置 MunqContainer 实例作为依赖解析器（dependency resolver）：

```
var container = new MunqContainer();
Dependency.SetResolver(container);
```

你就可以通过查询 **IDependencyRegistrar** 接口访问其注册接口：

```
var registrar = Dependency.Resolve<IDependencyRegistrar>();
```

在这里，*registrar* 是我们在先前示例中创建的相同 MunqContainer 实例（容器）。

如果你想使用其他的 IoC 容器，你只需创建一个类并使用你最喜欢的 IoC 容器实现 **IDependencyResolver** 和 **IDependencyRegistrar** 接口。

# CommonInitialization 静态类

[命名空间: *Serenity.Web*, 程序集: *Serenity.Web*]

如果要在 web 环境中使用默认设置，而不是手动组织服务定位器和其他配置，你只需在应用程序启动方法中调用 *CommonInitialization.Run()*。CommonInitialization 为一些 Serenity 抽象注册默认实现。

```
CommonInitialization.Run();
```

如果已经为一些抽象注册了提供者，CommonInitialization 不会覆盖它们。

此方法包含调用一些其他的方法来初始化 Serenity 平台的默认值：

```
public static class CommonInitialization
{
    public static void Run()
    {
        InitializeServiceLocator();
        InitializeSelfAssemblies();
        InitializeConfigurationSystem();
        InitializeCaching();
        InitializeLocalTexts();
        InitializeDynamicScripts();
    }

    public static void InitializeServiceLocator()
    {
        if (!Dependency.HasResolver)
        {
            var container = new MunqContainer();
            Dependency.SetResolver(container);
        }
    }

    //...
}
```

CommonInitialization.InitializeServiceLocator 和其他方法也可以单独地使用，  
而不需要调用 CommonInitialization.Run。

*InitializeServiceLocator()* 注册一个 MunqContainer 实例作为  
IDependencyResolver 的默认实现。

# 认证 & 授权

Serenity 使用一些抽象（abstractions）来与你的应用程序自身的用户身份验证和授权机制一起工作。

```
Serenity.Abstractions.IAuthenticationService  
Serenity.Abstractions.IAuthorizationService  
Serenity.Abstractions.IPermissionService  
Serenity.Abstractions.IUserRetrieveService
```

由于 Serenity 没有为这三个抽象提供默认实现，你应该使用依赖注册系统为它们提供一些实现。

例如，Serenity 基本应用程序示例在其 `SiteInitialization.ApplicationStart` 方法注册权限，如：

```
var registrar = Dependency.Resolve<IDependencyRegistrar>();  
  
registrar.RegisterInstance<IAuthorizationService>(  
    new Administration.AuthorizationService());  
  
registrar.RegisterInstance<IAuthenticationService>(  
    new Administration.AuthenticationService());  
  
registrar.RegisterInstance<IPermissionService>(  
    new Administration.PermissionService());  
  
registrar.RegisterInstance<IUserRetrieveService>(  
    new Administration.UserRetrieveService());
```

在编写你自己的权限之前，你可能需要看看这些示例的实现。

# IAuthenticationService 接口

[命名空间: *Serenity.Abstractions*, 程序集: *Serenity.Core*]

```
public interface IAuthenticationService
{
    bool Validate(ref string username, string password);
}
```

通常这是由登录页面调用检查输入的凭据是否正确的服务。如果用户名/密码正确，你的实现就应该返回 `true`。

模拟身份验证服务可以这样写：

```
public class DummyAuthenticationService : IAuthenticationService
{
    public bool Validate(ref string username, string password)
    {
        return username == password;
    }
}
```

如果用户名等于指定的密码（仅用于演示），该服务则返回 `true`。

第一个参数是含 `ref` 的参数，在登录之前，你可以把用户名修改为其在数据库中的实际表示形式。例如，用户可能在登录表单中输入大写的 `JOE`，但是实际上在数据库中的用户名是 `Joe`。这不是一项强制要求，但是如果你的数据库是大小写敏感的，可能在登录或登录后的过程中出现问题。

你可以在 `global.asax.cs` / `SiteInitialization.ApplicationStart` 注册该服务，如：

```
protected void Application_Start(object sender, EventArgs e)
{
    Dependency.Resolve<IDependencyRegistrar>()
        .RegisterInstance(new DummyAuthenticationService());
}
```

并在你的登录表单中使用它：

```
void DoLogin(string username, string password)
{
    if (Dependency.Resolve<IAuthenticationService>()
        .Validate(ref username, password))
    {
        // FormsAuthentication.SetAuthenticationTicket etc.
    }
}
```

# IAuthorizationService 接口

[命名空间: *Serenity.Abstractions*, 程序集: *Serenity.Core*]

这是 Serenity 通过检查当前请求判断用户是否已登录的接口。

```
public interface IAuthorizationService
{
    bool IsLoggedIn { get; }
    string Username { get; }
}
```

对于 web 应用程序的一些基本实现可能是：

```
using Serenity;
using Serenity.Abstractions;

public class MyAuthorizationService : IAuthorizationService
{
    public bool IsLoggedIn
    {
        get { return !string.IsNullOrEmpty(Username); }
    }

    public string Username
    {
        get { return WebSecurityHelper.HttpContextUsername; }
    }
}

//// ...

void Application_Start(object sender, EventArgs e)
{
    Dependency.Resolve<IDependencyRegistrar>()
        .RegisterInstance(new MyAuthorizationService());
}
```



# IPermissionService 接口

[命名空间: *Serenity.Abstractions*, 程序集: *Serenity.Core*]

权限 (permission) 是授权做一些操作 (访问页面、调用指定服务)。在 Serenity 中，权限是一些分配给单个用户的字符串 keys (类似于 ASP.NET 角色)。

例如，如果一些用户有 `Administration` 权限，该用户可以看到该权限所能访问的导航连接或调用该权限所允许的服务。

你也可以使用像 `ApplicationID:PermissionID` (例如：`Orders:Create`) 的复合访问许可键 (permission keys)，但是 Serenity 并不关心 ApplicationID 和 PermissionID，它只使用作为一个整体的复合访问许可键。

```
public interface IPermissionService
{
    bool HasPermission(string permission);
}
```

你可能有一个表.....

```
CREATE TABLE UserPermissions (
    UserID int,
    Permission nvarchar(20)
)
```

然后查询该表以实现这个接口。

对于应用程序的简单示例，有一个唯一有 `Administration` 权限的 `admin` 用户：

```
using Serenity;
using Serenity.Abstractions;

public class DummyPermissionService : IPermissionService
{
    public bool HasPermission(string permission)
    {
        if (Authorization.Username == "admin")
            return true;

        if (permission == "Administration")
            return false;

        return true;
    }
}
```

# IUserDefinition 接口

[命名空间: *Serenity*, 程序集: *Serenity.Core*]

很多应用程序存储用户常见的信息，如 **ID**、显示名称（别名/全名）、邮箱等。*Serenity* 提供基础接口以独立应用程序方式访问这些信息。

```
public interface IUserDefinition
{
    string Id { get; }
    string Username { get; }
    string DisplayName { get; }
    string Email { get; }
    Int16 IsActive { get; }
}
```

你的应用程序应该实现该接口，但并不是所有这些信息都是 *Serenity* 自身必须的。只有 **Id**、**Username** 及 **IsActive** 属性是必需的。

**Id** 可以是整型、字符串或者 GUID，用以唯一标识用户。

**Username** 应该是唯一的用户名，但是你也可以使用邮箱地址作为用户名。

**IsActive** 活跃状态的用户返回 1；已删除(如果你没有在数据库中删除用户)的用户返回 -1；暂时禁用（锁住账号）的用户返回 0。

**DisplayName** 和 **Email** 是可选的，且当前并没有被 *Serenity* 使用，可能你的应用程序需要它们。

# IUserRetrieveService 接口

[命名空间: *Serenity.Abstractions*, 程序集: *Serenity.Core*]

当 Serenity 需要使用给定的用户名或者用户 ID 访问 IUserDefinition 对象时，它需要使用该接口。

```
public interface IUserRetrieveService
{
    IUserDefinition ById(string id);
    IUserDefinition ByUsername(string username);
}
```

在你的实现中，缓存用户定义对象是一个好主意，因为通常 WEB 应用程序的同一用户可能重复使用此接口。

Serenity 基本应用程序示例已经有该接口的实现，如：

```
public class UserRetrieveService : IUserRetrieveService
{
    private static MyRow.RowFields fld { get { return MyRow.Fields; } }

    private UserDefinition GetFirst(IDbConnection connection, BaseCriteria criteria)
    {
        var user = connection.TrySingle<Entities.UserRow>(criteria);
        if (user != null)
            return new UserDefinition
            {
                UserId = user.UserId.Value,
                Username = user.Username,
                //...
            };
        return null;
}
```

```
}

public IUserDefinition GetById(string id)
{
    if (id.IsEmptyOrNull())
        return null;

    return TwoLevelCache.Get<UserDefinition>("UserByID_" + id, CacheExpiration.Never, CacheExpiration.OneDay, fld.GenerationKey, () =>
    {
        using (var connection = SqlConnections.NewByKey("Default"))
            return GetFirst(connection,
                new Criteria(fld.UserId) == id.TryParseID32().Value);
    });
}

public IUserDefinition ByUsername(string username)
{
    if (username.IsEmptyOrNull())
        return null;

    return TwoLevelCache.Get<UserDefinition>("UserByName_" + username, CacheExpiration.Never, CacheExpiration.OneDay, fld.GenerationKey, () =>
    {
        using (var connection = SqlConnections.NewByKey("Default"))
            return GetFirst(connection, new Criteria(fld.Username) == username);
    });
}
```

# 静态 Authorization 类

[命名空间: *Serenity*, 程序集: *Serenity.Core*]

Authorization 类提供访问一些像 `IAuthService`、`IPermissionService` 等服务提供的信息的捷径。

例如，要替换下面的写法

```
Dependency.Resolve<IAuthorizationService>().HasPermission("SomePermission")
```

你可以使用

```
Authorization.HasPermission("SomePermission")
```

```
public static class Authorization
{
    public static bool IsLoggedIn { get; }
    public static IUserDefinition UserDefinition { get; }
    public static string UserId { get; }
    public static string Username { get; }
    public static bool HasPermission(string permission);
    public static void ValidateLoggedIn();
    public static void ValidatePermission(string permission);
}
```

`IsLoggedIn`、`UserDefinition`、`UserId`、`Username` 和 `HasPermission` 是为了让使用对应服务更容易访问当前用户信息。

`ValidateLoggedIn` 检查是否有登录用户，如果没有登录用户，则抛出含 `NotLoggedIn` 错误码的 `ValidationException` 异常。

`ValidatePermission` 检查登录用户是否有指定的权限，如果没有相应权限，则抛出含 `AccessDenied` 错误码的 `ValidationException` 异常。



## 配置系统

.NET 应用程序通常在 `app.config`（桌面应用程序）或 `web.config`（web应用程序）文件保存配置。

虽然，在 Web 应用程序用文件存储配置很普遍，但有时也需要在数据库表中存储一些配置，使其可用于网站群中的所有服务器，且只须在一个位置设置。

就像 `IsolatedStorage` 有应用程序（Application）、机器（Machine）、用户（User）等作用域，配置设置可能有不同的作用域：

- Application - 运行 Web 应用程序的所有服务器之间共享
- Server - 只应用于当前服务器
- User - 只应用于当前用户

还有举出更多的例子。

如果你只有一台服务器，`Application` 和 `Server` 可以存储在 `web.config` 文件，但是在网站群的结构下，`Application` 设置应存储在所有服务器可访问的位置（数据库或共享的文件夹）。

用户设置通常与用户 ID 一道存储在数据库中。

Serenity 提供了一个可扩展的配置系统。

# 定义配置设置

在 Serenity 平台，配置设置是只是简单的类，如：

```
[SettingScope("Application"), SettingKey("Logging")]
private class LogSettings
{
    public LoggingLevel Level { get; set; }
    public string File { get; set; }
    public int FlushTimeout { get; set; }
}
```

如果需要，可以在类构造函数中设置默认设置。

## SettingScope 特性

如果有指定该特性，由该特性确定作用域的设置。

如果未指定该特性，默认作用域是 应用程序（*Application*）。

## SettingKey 特性

如果有指定该特性，此特性确定设置类的键（如 web.config 的 appSettings 键）。

如果未指定该特性，将使用类名作为键（key）。

# IConfigurationRepository 接口

[命名空间: *Serenity.Abstractions*, 程序集: *Serenity.Core*]

所有应用程序都有一些类型的配置。作用域（scope）、存储介质和格式化器的设置在不同应用程序间都是不相同的，因此，Serenity 提供 IConfigurationRepository 接口对此配置进行访问。

```
public interface IConfigurationRepository
{
    object Load(Type settingType);
    void Save(Type settingType, object value);
}
```

## IConfigurationRepository.Load 方法

此方法返回一个设置类型（settingType）实例。提供者会检查 SettingKey 特性来决定设置类型的键（key）。

如果一些提供者被注册了多个作用域（scopes），提供者还应检查 SettingScope 特性。

即使没有找到设置（设置类型的默认构造函数会创建一个对象），提供者应返回一个对象实例。

## IConfigurationRepository.Save 方法

保存设置类型（settingType）实例。提供者会检查 SettingKey 特性来决定设置类型的键（key）。

如果一些提供者被注册了多个作用域（scopes），提供者还应检查 SettingScope 特性。

当你不想设置被改变时，此方法的实现是可选的。在这种情况下，只需抛出 *NotImplementedException* 异常。



# AppSettingsJsonConfigRepository

[命名空间: *Serenity.Configuration*, 程序集: *Serenity.Data*]

大多数 Web 应用程序在 web.config 中的 appSettings 节点存储配置设置。

Serenity 提供一个 IConfigurationRepository 接口的默认实现，可以使用 appSettings 作为配置存储。

```
public class AppSettingsJsonConfigRepository : IConfigurationRepository
{
    public void Save(Type settingType, object value)
    {
        throw new NotImplementedException();
    }

    public object Load(Type settingType)
    {
        return LocalCache.Get("ApplicationSetting:" + settingType.FullName,
            TimeSpan.Zero, delegate()
        {
            var keyAttr = settingType.GetCustomAttribute<SettingKeyAttribute>();
            var key = keyAttr == null ? settingType.Name : keyAttr.Value;
            return JSON.Parse(ConfigurationManager.AppSettings[key].TrimOrNull() ??
                "{}", settingType);
        });
    }
}
```

需要手动注册该提供者：

```
var registrar = Dependency.Resolve<IDependencyRegistrar>();
RegisterInstance< IConfigurationRepository>("Application",
    new AppSettingsJsonConfigRepository())
```

当调用 `Serenity.Web.CommonInitialization.Run()` 时，如果你没有注册另一个实现，它将注册该类作为 `IConfigurationRepository` 的默认提供者(在 `Application` 作用域)。

该提供者希望在 `web.config / app.config` 文件以 JSON 格式定义设置：

```
<appSettings>
    <add key="Logging" value="{
        File: '~\\App_Data\\Log\\App_{0}_{1}.log',
        FlushTimeout: 0, Level: 'Debug' }" />
</appSettings>
```

`Serenity` 默认只包含此配置提供者。你可以它为例，编写另一个适合你的设置（从数据库加载）。

在实现中对返回对象进行缓存是一个好主意，以避免每次读取设置的反序列化成本。

# 静态 Config 类

[命名空间: *Serenity*, 程序集: *Serenity.Core*]

这是访问配置设置的主要位置。它包含注册 *IConfigurationRepository* 提供者的快捷方法。

```
public static class Config
{
    public static object Get(Type settingType);
    public static TSettings Get<TSettings>() where TSettings: class, new();
    public static object TryGet(Type settingType);
    public static TSettings TryGet<TSettings>() where TSettings : class, new();
}
```

## Config.Get 方法 Config.Get Method

用于读取指定类型的配置设置。

如果没有为设置类型的作用域注册提供者，将抛出 *KeyNotFoundException* 异常。

如果没有找到设置，提供者通常返回一个默认实例。

我更喜欢使用泛型重载，因为可以避免强制转换返回对象。

```
if (Config.Get<LogSettings>().LoggingLevel != LogginLevel.Off)
{
    // ...
}
```

## Config.TryGet 方法

用于读取指定类型的配置设置。

在功能上等效于 `Get`，但是如果没有为设置作用域注册配置提供者，`Get` 将引发异常，`TryGet` 则返回 `null`。

```
if ((Config.TryGet<LogSettings>() ?? new LogSettings()).LoggingLevel != LogLevel.Off)
{
    // ...
}
```

我更喜欢使用该方法，而不是 `Get`，因为当配置系统还没有被初始化时，该方法可以避免异常。

从安全角度看，更推荐使用 `Get` 方法。

## 本地化

大多数 web 应用程序必须支持多种语言。例如：Youtube、 Wikipedia、 Facebook 网站都支持多语言。

当用户第一次访问网站时，根据用户浏览器（预先设置的地区）自动选择显示的语言。

如果自动选择的语言不是用户所期望的，用户可以设置他们的首选语言，用户的选  
择存储在客户端 cookie（或服务器端用户简介表（user profile table））。

一旦选择了语言，所有的文本都将显示为所选的语言。

Serenity 平台从开始设计之初就考虑到本地化功能。

如果你在使用 Serenity 基本应用程序示例，可以通过设置浏览器语言或更改  
web.config 的设置来查看本地化功能：

```
<system.web>
    <globalization culture="en-US" uiCulture="auto:en-US" />
</system.web>
```

在这里，UI 文化（culture）设置为自动，如果自动检测失败，则使用 EN-US。

The screenshot shows a web browser window titled 'Customers' with the URL 'localhost:55555/Northwind/Customer'. The page is part of a 'Serenity Sample Site'. On the left, there is a sidebar with a dark theme containing navigation links: Dashboard, Northwind (selected), Customers, Products, Suppliers, Shippers, Categories, Regions, Territories, and Administration. The main content area is titled 'Customers' and displays a table with 91 rows of data. The columns are: ID, Company Name, Contact Name, Contact Title, City, Region, Postal Code, and Country. The data includes entries like 'ALFKI' (Alfreds Futterkiste) from Berlin, Germany, and 'BSBEV' (B's Beverages) from London, UK. At the bottom of the table, it says 'Showing 1 to 91 of 91 total records'. The browser's address bar shows the full URL.

像下面这样修改该配置，并刷新浏览器，你的网站将使用土耳其语。

```
<system.web>
    <globalization culture="en-US" uiCulture="tr" />
</system.web>
```

Kayıt No	Firma Adı	İlgili Kişi Adı	İlgili Kişi Unvanı	Şehir	Bölge	Posta Kodu	Ülke
ALFKI	Alfreds Futterkiste	Maria Anders	Sales Representative	Berlin		12209	Germany
ANATR	Ana Trujillo Emparedados y helados	Ana Trujillo	Owner	México D.F.		05021	Mexico
ANTON	Antonio Moreno Taquería	Antonio Moreno	Owner	México D.F.		05023	Mexico
AROUT	Around the Horn	Thomas Hardy	Sales Representative	London		WA1 1DP	UK
BERGS	Berglunds snabbköp	Christina Berglund	Order Administrator	Luleå		S-958 22	Sweden
BLAUS	Blauer See Delikatessen	Hanna Moos	Sales Representative	Mannheim		68306	Germany
BLONP	Blondesdssl père et fils	Frédérique Citeaux	Marketing Manager	Strasbourg		67000	France
BOLID	Bólido Comidas preparadas	Martín Sommer	Owner	Madrid		28023	Spain
BONAP	Bon app'	Laurence Lebihan	Owner	Marseille		13008	France
BOTTM	Bottom-Dollar Markets	Elizabeth Lincoln	Accounting Manager	Tsawassen	BC	T2F 8M4	Canada
BSBEV	B's Beverages	Victoria Ashworth	Sales Representative	London		EC2 5NT	UK
CACTU	Cactus Comidas para llevar	Patricia Simpson	Sales Agent	Buenos Aires		1010	Argentina
CENTC	Centro comercial Moctezuma	Francisco Chang	Marketing Manager	México D.F.		05022	Mexico
CHOPS	Chop-suey Chinese	Yang Wang	Owner	Bern		3012	Switzerland
COMMI	Comércio Mineiro	Pedro Afonso	Sales Associate	Sao Paulo	SP	05432-043	Brazil
CONSH	Consolidated Holdings	Elizabeth Brown	Sales Representative	London		WX1 6LT	UK
DRACD	Drachenblut Delikatessen	Sven Ottlieb	Order Administrator	Aachen		52066	Germany
DUMON	Du monde entier	Janine Labrune	Owner	Nantes		44000	France
EASTC	Eastern Connection	Ann Devon	Sales Agent	London		WX3 6FW	UK
ERNSH	Ernst Handel	Roland Mendel	Sales Manager	Graz		8010	Austria

在这里，数据没有被翻译，但也可以使用一些像文化扩展表（culture extension tables）的方法来翻译用户输入的数据。

# LocalText 类

[命名空间: *Serenity*, 程序集: *Serenity.Core*]

字符串本地化的核心是 LocalText 类。

```
public class LocalText
{
    public LocalText(string key);
    public string Key { get; }
    public override string ToString();
    public static implicit operator string(LocalText localText);
    public static implicit operator LocalText(string key);
    public static string Get(string key);
    public static string TryGet(string key);

    public const string InvariantLanguageID = "";
    public static readonly LocalText Empty;
}
```

它的构造函数接受一个 `key` 参数，它定义将要包含的本地化文本键（local text key）。一些键的示例：

- Enums.Month.January
- Enums.Month.December
- Db.Northwind.Customer.CustomerName
- Dialogs.YesButton

虽然它不是一个规则，但遵循点约定（如，本地化文本键）的命名空间是一个好的主意。

在运行时，通过 `ToString()` 函数，本地化文本键被翻译为当前语言（即 `CultureInfo.CurrentCulture`）的表示形式。

```
var text = new LocalText("Dialogs.YesButton");
Console.WriteLine(text.ToString());
```

```
> Yes
```

如果本地化文本表（我们将在后面谈论它）中找不到该翻译，则返回 `key` 自身。

```
var text = new LocalText("Unknown.Local.Text.Key");
Console.WriteLine(text.ToString());
```

```
> Unknown.Local.Text.Key
```

这是特意设计的，以便开发人员可以查明没有被翻译的文本。

## LocalText.Key 属性

获取 `LocalText` 实例包含的本地化文本键。

### 从字符串中隐式转换

`LocalText` 可以从 `String` 类型隐式转换。

```
LocalText someText = "Dialogs.YesButton";
```

这里的 `someText` 变量得到一个键为 `Dialogs.YesButton` 的新 `LocalText` 实例引用。所以它是只是一个 `LocalText` 构造函数的快捷方式。

### 隐式转换为字符串

`LocalText` 也实现了 `String` 类型的隐式转换，但是它返回翻译而不是键（像调用 `ToString` 方法）。

```
var lt = new LocalText("Dialogs.NoButton");
string text = lt;
Console.WriteLine(text);
```

> No

## LocalText.Get 静态方法

若要访问本地化文本键翻译而无须创建一个 LocalText 实例，请使用 Get 方法：

```
Console.WriteLine(LocalText.Get("Dialogs.YesButton"));
```

> Yes

*ToString()* 在 Get 方法内部被调用。

## LocalText.TryGet 静态方法

不像 Get 方法在找不到翻译时返回本地化文本键，TryGet 返回 null。因此，合并运算符（coalesce operator）必须与 TryGet 一起使用：

```
var translation = LocalText.TryGet("Looking.For.This.Key") ?? "Default Text";
Console.WriteLine(translation);
```

> Default Text

## LocalText.Empty 字段

类似于 String.Empty，LocalText 包含一个含空键（empty key）的空本地化文本对象。

## LocalText.InvariantLanguageID 常量

这只是固定语言 ID (invariant language ID) 的空字符串，它是固定区域语言标识符（通常默认语言为英语）。

我们将在后面的章节谈论语言标识符（language identifiers）。

## 语言标识符

LanguageID 是指定字母和数字的组合作为语言标识符或分类的代码。

LanguageID 遵循 RFC 1766 标准格式：`<languagecode2>-<country/regioncode2>`，其中 *languagecode2* 是来自 ISO 639-1 的两个小写字母代码，而 *country/regioncode2* 是来自 ISO 3166 的两个大写字母代码。

这是一些 LanguageID 示例：

- `en` : 英语
- `en-US` : 美式英语（US 是 ISO 3166-1 国家代码）
- `en-GB` : 英式英语（GB 是 ISO 3166-1 国家代码）
- `es` : 西班牙语
- `es-AR` : 阿根廷式西班牙语

## 固定语言（Invariant Language）

类似于 `CultureInfo.InvariantCulture`，固定语言是空标识符的默认语言。

除非另有指定，否则在程序集嵌入的文本被认为写在固定语言。

虽然通常固定语言认为是英语，但可以使用你的母语作为固定语言。

# 语言回退 (Language Fallbacks)

## 非特定(Neutral)语言回退

当在 `en-US` 中找不到翻译，它可以接受在 `en` 语言中寻找翻译，因为它们密切相关。

两个字母的 `languageID` (非特定语言) 是国家的 4 个特定字母代码的隐式语言回退。

因此，`es` 是 `es-AR` 语言回退，`en` 是 `en-US` 和 `en-GB` 的语言回退。

## 固定语言回退

空代码的固定语言是所有语言的最终隐式回退语言。

## 实现

语言回退功能应该通过 `ILocalTextRegistry` 提供者 (如，`LocalTextRegistry` 类) 实现。

提供者也支持显式设置语言回退。因此如果需要，你也可以设置 `en-US` 作为 `en-UK` 的语言回退。

这是检索本地化文本键翻译的步骤：

- 如果当前语言有键的翻译，则返回该翻译。
- 检查每个显式定义的语言回退翻译。
- 如果 `languageID` 是 4 个国家特定字母代码，检查非特定语言的翻译。
- 检查固定语言的翻译。
- `TryGet` 返回键本身或 `null`

假设我们设置 `en-US` 作为 `en-UK` 的语言回退。

如果要在 `en-UK` 查找翻译，按如下排序检索：

1. `en-UK`

2. en-US
3. en
4. invariant

# ILocalTextRegistry 接口

[命名空间: *Serenity.Abstractions*, 程序集: *Serenity.Core*]

`LocalText` 类通过该接口的提供者访问本地化文本键的翻译。

```
public interface ILocalTextRegistry
{
    string TryGet(string languageID, string key);
    void Add(string languageID, string key, string text);
}
```

## ILocalTextRegistry.TryGet 方法

获得目标语言指定键的翻译。

由 `CultureInfo.CurrentCulture` 决定当前语言。

如果没有在请求语言中找到翻译，提供者的职责是检查键的语言回退。

如果在语言层级（从请求语言往固定语言搜索）没有找到翻译，该方法返回 `null`。

## ILocalTextRegistry.Add 方法

向内部由本地化文本注册保存的本地化文本表添加翻译。

本地化文本表是内存表（字典），如：

Key	LanguageID	Text (Translation)
Dialogs.YesButton	en	Yes
Dialogs.YesButton	tr	Evet
Dialogs.NoButton	en	No
Dialogs.NoButton	tr	Hayır

如果 Key/LanguageID 对被重复添加，该方法不会抛出异常。它会覆盖已存在的翻译。



# LocalTextRegistry 类

[命名空间: *Serenity.Localization*, 程序集: *Serenity.Core*]

此类是 *ILocalTextRegistry* 接口可嵌入的默认实现。

```
public class LocalTextRegistry : ILocalTextRegistry
{
    public void Add(string languageID, string key, string text);
    public string TryGet(string languageID, string key);
    public void SetLanguageFallback(string languageID, string languageFallbackID);

    public void AddPending(string languageID, string key, string text);
    public string TryGet(string languageID, string textKey, bool isApprovalMode);

    public Dictionary<string, string> GetAllAvailableTextsInLanguage(
        string languageID, bool pending);
}
```

Add 和 TryGet 实现了 *ILocalTextRegistry* 接口的相应方法。

## LocalTextRegistry.SetLanguageFallback 方法

为语言回退（language fallback）设置指定的语言。

```
var registry = (LocalTextRegistry)(Dependency.Resolve<ILocalText
Registry>());
registry.SetLanguageFallback('en-UK', 'en-US');
// from now on if a translation is not found in "en-UK" language,
// it will be looked up in "en-US" language first, followed by "en".
```

1

▶

可以在有关章节中找到有关语言回退的详细信息。

## 注册 LocalTextRegistry 作为提供者

通常在应用程序启动方法中做此操作：

```
var registrar = Dependency.Resolve<IDependencyRegistrar>();
registrar.RegisterInstance<ILocalTextRegistry>(new LocalTextRegi
stry());
```

如果没有已注册的提供者，*CommonInitialization.Run* 或  
*CommonInitialization.InitializeLocalTexts* 方法将注册一个 LocalTextRegistry  
实例作为 ILocalTextRegistry 提供者。

## 待审状态

LocalTextRegistry 还支持可选的待审状态。

在一些网站，翻译可能需要在发布之前经过一些版主的审核。

所以你可能将未经审核的文本添加到本地化注册表（local text registry），但希望在批准时，只有版主可以检查它们将如何在直播网站显示。

LocalTextRegistry 允许你将一些文本标记为待审，并只有批准上下文（例如，当版主已登录）可使用这些翻译文本。

## ILocalTextContext 接口

[命名空间: Serenity.Localization, 程序集: Serenity.Core]

```
public interface ILocalTextContext
{
    bool IsApprovalMode { get; }
```

实现该接口并通过服务定位器（依赖类）注册它。

IsApprovalMode 属性用来决定当前上下文是否处于审核状态（如，被版主使用）。

```

public class MyLocalTextContext : ILocalTextContext
{
    public bool IsApprovalMode
    {
        get
        {
            // use some method to determine if current user is a
            moderator
            return Authorization.HasPermission("Moderation");
        }
    }

    void ApplicationStart()
    {
        Dependency.Resolve<IDependencyRegistrar>()
            .RegisterInstance<ILocalTextContext>(new MyLocalTextCont
ext());
    }
}

```

## LocalTextRegistry.AddPending 方法

在本地化表（local text table）添加待审文本。只有当前处于待审状态的上下文可使用这些文本。

## 含语言和待审状态的 LocalTextRegistry.TryGet 重载

```

public string TryGet(string languageID, string textKey, bool isA
pprovalMode);

```

该重载获得指定语言的翻译并可以选择使用未审核的文本 (isApprovalMode = true)。

另外只有当 ILocalTextContext 提供者返回的 IsApprovalMode 属性为 true 时，TryGet 重载才返回未审核文本。

## **LocalTextRegistry.GetAllAvailableTextsInLanguage 方法**

返回语言中所有当前注册翻译的文本键（text keys）字典。

当字典的值（value）被翻译时，key 为本地化文本键（local text keys）。

它也包含回退文本，当文本不能被翻译成目标的语言时，则使用该文本。

## 注册翻译

有几种方法来定义本地文本键（local text keys）和翻译，包括：

- 通过 `ILocalTextRegistry.Add` 方法手工添加
- 在嵌套静态类中声明包含本地文本的对象
- 在枚举类添加 `Description` 特性
- 在预先确定的位置定义 JSON 文件（`~/scripts/serenity/texts`、`~/scripts/site/texts` 和 `~/App_Data/texts`）

我们谈论所有这些方法。

## 手工注册翻译

你可以在应用程序的启动方法中向本地化文本注册表（local text registry）添加翻译。

这些翻译可来自数据库表、xml 文件、嵌入的资源等。

```
void Application_Start()
{
    // ...
    var registry = Dependency.Resolve<ILocalTextRegistry>();
    registry.Add("es", "Dialogs.YesButton", "Sí");
    registry.Add("fr", "Dialogs.YesButton", "Oui");
    // ...
}
```

# 嵌入本地化文本

Serenity 允许你定义包含 `LocalText` 对象的静态嵌套类来定义翻译，如：

```
[NestedLocalTexts]
public static partial class Texts
{
    public static class Site
    {
        public static class Dashboard
        {
            public static LocalText WelcomeMessage =
                "Welcome to Serenity BasicApplication home page.
" +
                "Use the navigation on left to browse other page
S...";
        }
    }

    public static class Validation
    {
        public static LocalText DeleteForeignKeyError =
            "Can't delete record. '{0}' table has records that d
epends on this one!";

        public static LocalText SavePrimaryKeyError =
            "Can't save record. There is another record with the
same {1} value!";
    }
}
```

此定义允许使用智能感知来引用本地化文本，而不必记住字符串键。

这些嵌入式的翻译通常用于定义固定语言（最终回退）的默认翻译。

这是该 `Texts` 类定义的翻译表：

Key	LanguageID	Text (Translation)
Site.Dashboard.WelcomeMessage		Welcome to Serenity BasicApp...
Validation.DeleteForeignKeyError		Can't delete record...
Validation.SavePrimaryKeyError		Can't save record...

本地化文本键由静态嵌套类之间使用点连接类名组成。尽管使用 *Texts* 命名以保持一致性是好主意，但是最顶层静态类（*Texts*）的名称将被忽略。

除非另有说明，这些文本的 *LanguageID* 都被认为是固定语言（空字符串）。

## NestedLocalTexts 特性

嵌套本地化文本注册类的最顶层类（如 *Texts*）必须有该属性。

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple=false)]
public sealed class NestedLocalTextsAttribute : Attribute
{
    public NestedLocalTextsAttribute()
    {
    }

    public string LanguageID { get; set; }
    public string Prefix { get; set; }
}
```

它包含两个可选属性：*LanguageID* 和 *Prefix*。

*LanguageID* 允许你定义翻译的目标语言。

如果没有指定该属性，将使用固定语言。

在固定语言中注册默认文本是一个好主意，即使注册的文本不是英文。因为它是所有语言的最终回退语言。

如果我们这样使用：

```
[NestedLocalTexts(LanguageID = "en-US")]
public static partial class Texts
{
    // ...
}
```

在翻译表的 LanguageID 列将是 "en-US" :

Key	LanguageID	Text (Translation)
Site.Dashboard.WelcomeMessage	en-US	Welcome to Serenity BasicApp...
Validation.DeleteForeignKeyError	en-US	Can't delete record...

Prefix 属性值被用来作为本地化文本键 (local text keys) 前缀 :

```
[NestedLocalTexts(LanguageID = "en-US", Prefix = "APrefix.")]
public static partial class Texts
{
    // ...
}
```

Key	LanguageID	Text (Translation)
APrefix.Site.Dashboard.WelcomeMessage	en-US	Welcome to Serenity BasicApp...
APrefix.Validation.DeleteForeignKeyError	en-US	Can't delete record...

## NestedLocalTextRegistration 类

[命名空间: *Serenity.Localization*, 程序集: *Serenity.Core*]

要注册嵌套本地化文本定义，需要在应用程序启动时调用 *NestedLocalTextRegistration.Initialize()* 方法：

```
void Application_Start()
{
    NestedLocalTextRegistration.Initialize();
}
```

默认调用 CommonInitialization.Run 和  
CommonInitialization.InitializeLocalTexts 方法。

一旦运行应用程序，所有自动生成键的翻译都添加到当前的 ILocalTextRegistry 提供者，并且在静态嵌套类中的 LocalText 实例被替换为包含生成键（通过反射设置）的实际 LocalText 实例。

# 枚举文本

可以用 `Description` 特性显示指定枚举值的文本。

```
namespace MyApplication
{
    public enum Sample
    {
        [Description("First Value")]
        Value1 = 1,
        [Description("Second Value")]
        Value2 = 2
    }
}
```

此枚举和它的 `Description` 特性定义以下的本地化文本键和翻译：

Key	LanguageID	Text (Translation)
Enums.MyApplication.Sample.Value1		First Value
Enums.MyApplication.Sample.Value2		Second Value

默认情况下，所有的文本都作为固定语言 ID 的翻译。

可以使用这些键访问枚举值的翻译描述，或者使用枚举类型（需引用 `Serenity` 命名空间）定义的扩展方法 `GetText()`。

```
using Serenity;
//...
Console.WriteLine(MyApplication.Sample.Value1.GetText());
```

> First Value

## EnumKey 特性

枚举翻译使用枚举类型的全名作为生成本地文本键的前缀。该前缀也以被 `EnumKeyAttribute` 重载：

```
namespace MyApplication
{
    [EnumKey("Something")]
    public enum Sample
    {
        [Description("First Value")]
        Value1 = 1,
        [Description("Second Value")]
        Value2 = 2
    }
}
```

现在定义的键和翻译是：

Key	LanguageID	Text (Translation)
Enums.Something.Value1		First Value
Enums.Something.Value2		Second Value

## EnumLocalTextRegistration 类

[命名空间: `Serenity.Localization`, 程序集: `Serenity.Core`]

若要枚举注册的本地化文本定义，你需要在应用程序启动时调用

`EnumLocalTextRegistration.Initialize()` 方法：

```
void Application_Start()
{
    EnumLocalTextRegistration.Initialize(ExtensibilityHelper.SelfAssemblies);
}
```

它获取程序集列表以检索枚举类型。你可以手工传递程序集列表或使用 `ExtensibilityHelper.SelfAssemblies` (包含 Serenity 程序集的所有引用)。

默认情况下，`CommonInitialization.Run` 和  
`CommonInitialization.InitializeLocalTexts` 方法会调用它。

# JSON 本地化文本

Serenity 支持通过 JSON 文件注册包含键/值字典的本地化文本：

```
{
  "Forms.Administration.User.DisplayName": "Display Name",
  "Forms.Administration.User.Email": "E-mail",
  "Forms.Administration.User.EntitySingular": "User",
  "Forms.Administration.User.EntityPlural": "Users"
}
```

要从某一文件夹中的 JSON 文件注册本地化文本键和翻译，可调用 `JsonLocalTextRegistration.AddFromFilesInFolder`：

```
JsonLocalTextRegistration.AddFromFilesInFolder(@"C:\SomeFolder")
;
```

文件夹中的文件名称必须遵循如下约定：

`{前缀}.{LanguageID}.json`

`{LanguageID}` 是两个或四个字母的语言代码。使用 `invariant` 作为固定语言的语言代码。

一些文件名称示例：

- `site.texts.en-US.json`
- `MyCoolTexts.es.json`
- `user.texts.invariant.json`

文件夹的文件被解析并按文件名称的顺序注册。因此，以上面的文件名称为例，注册顺序应为：

1. `MyCoolTexts.es.json`
2. `site.texts.en-US.json`
3. `user.texts.invariant.json`

因为添加具有相同键的翻译会覆盖先前的翻译，所以该顺序是很重要的。

## CommonInitialization 和 Predetermined 文件夹

*CommonInitialization.Run* 和 *CommonInitialization.InitializeLocalTexts* 方法将在 web 站点下的三个预定位置获得翻译文本：

1. ~/Scripts/serenity/texts (serenity 翻译)
2. ~/Scripts/site/texts (应用程序特定翻译)
3. ~/App\_Data/texts (用户通过翻译窗体的翻译)

开发者自己的翻译文件请放在第二个路径下，因为第一个路径下的文件是 Serenity 资源。

第三个路径包含用户的翻译文本。在发布之前，建议把这些文本转移到应用程序 ~/Scripts/site/texts 下的翻译文件。

## 缓存

缓存是现代高并发应用程序的重要组成部分。即使你的 web 应用程序目前还没有那么高的并发量，但在之后的发展中极有可能会遇到高并发的应用场景，因此从一开始就使用缓存设计程序是一个好主意。

- 本地缓存
- 分布式缓存
- 二级缓存

## 本地缓存

Serenity 提供一些缓存抽象和实用功能让你更容易地使用本地缓存。

术语 本地 (*local*) 的意思是指在本地内存中缓存项目（因此没有涉及到序列化）。

当你的应用程序在网站群(web farm) 中部署时，本地缓存可能还不够或者有时合适。我们将在 [分布式缓存](#) 章节中讨论该场景。

# ILocalCache 接口

[命名空间: *Serenity.Abstractions*] - [程序集: *Serenity.Core*]

定义一个基本的本地缓存接口。

```
public interface ILocalCache
{
    void Add(string key, object value, TimeSpan expiration);
    TItem Get<TItem>(string key);
    object Remove(string key);
    void RemoveAll();
}
```

使用 `Serenity.Web` 程序集中的 `System.Web.Cache` 默认实现 `ILocalCache` (`Serenity.Caching.HttpRuntimeCache`)。

## ILocalCache.Add 方法

添加指定键的值到缓存。如果在缓存中存在该值，则更新该值。

项目保存在缓存中直至 `到期(expiration)` 期限。你可以指定项目的 `TimeSpan.Zero` 为不自动过期。

添加到缓存的值使用绝对过期时间（因此它们在特定时间过期，不会逾期）。

```
Dependency.Resolve<ILocalCache>.Add("someKey", "someValue", TimeSpan.FromMinutes(5));
```

此方法默认使用 `HttpRuntime.Cache.Insert` 方法实现。

避免使用 `HttpRuntime.Cache.Add` 方法，因为它不会更新缓存中相同键的值，甚至不会抛出错误，因此使用该方法你不会得到任何通知。

## ILocalCache.Get <TItem> 方法

获取本地缓存中指定键的值。

如果缓存中没有该键，只有在值类型是 `TItem` 时才抛出错误。若 `TItem` 是引用类型，返回的值为 `null`。

如果值的类型不是 `TItem`，则抛出一个异常。

你可以使用 `object` 作为 `TItem` 参数阻止不存在的值或不是请求类型情况下的错误。

## ILocalCache.Remove 方法

删除本地缓存中指定键的项目，并返回其值。

如果缓存中没有指定键的值，不会抛出错误，而是返回 `null`。

```
Dependency.Resolve<ILocalCache>.Remove("someKey");
```

## ILocalCache.RemoveAll 方法

删除本地缓存中所有的项目。除了单元测试这种特殊情况，避免使用该方法，否则会影响性能。

# 静态 LocalCache 类

[命名空间: *Serenity*] - [程序集: *Serenity.Core*]

一个包含快捷访问注册 `ILocalCache` 提供者的静态类。

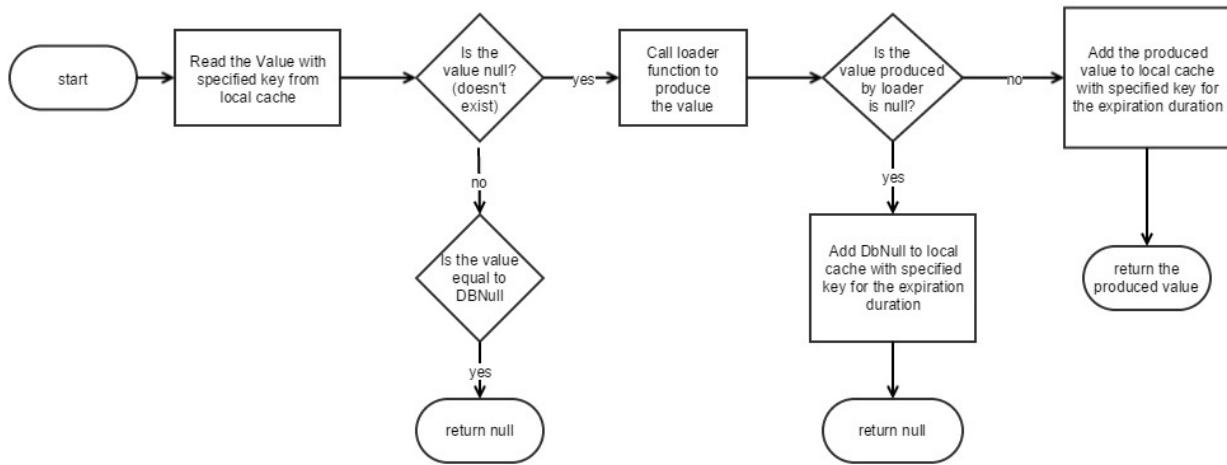
```
public static class LocalCache
{
    public static void Add(string key, object value, TimeSpan ex
piration);
    public static TItem Get<TItem>(string key, TimeSpan expirati
on,
        Func<TItem> loader) where TItem : class;
    public static void Remove(string key);
    public static void RemoveAll();
}
```

`Add`、`Remove` 和 `RemoveAll` 方法是对应 `ILocalCache` 接口的快捷访问方法，但 `Get` 方法不同于 `ILocalCache.Get`。

## LocalCache.Get <TItem> 方法

获取本地缓存指定键的值。

如果缓存中没有该键，使用加载 (`loader`) 函数产生值，并使用指定的键添加到缓存。



- 如果缓存中的值是 `DBNull.Value`，返回 `null`（例如，使用这种方式，如果数据库中不存在该 ID 的用户，则阻止在数据库中重复查询该 ID）。
- 如果缓存中存在该值，但不是 `TItem` 类型，将抛出异常；否则返回该值。
- 如果缓存中不存在该值，调用加载（`loader`）函数产生该值（如，从数据库加载）并……
  - 如果加载（`loader`）函数产生的值为 `null`，在缓存中存储为 `DBNull.Value`。
  - 否则，对产生的值指定过期时间后添加到缓存。

# 用户简介（User Profile）缓存示例

假设我们网站有一个使用多个查询生成的简介页面。我们有此页的模型，如 `UserProfile` 类，它包含用户所有简介数据，及一个获取指定用户 `id` 简介的 `GetProfile` 方法。

```
public class UserProfile
{
    public string Name { get; set; }
    public List<CachedFriend> Friends { get; set; }
    public List<CachedAlbum> Albums { get; set; }

    ...
}
```

```
public UserProfile GetProfile(int userID)
{
    using (var connection = new SqlConnection("..."))
    {
        // load profile by userID from DB
    }
}
```

通过使用 `LocalCache.Get` 方法，我们可以轻松地缓存此信息一小时，并避免每次请求该信息时对数据库进行调用。

```
public UserProfile GetProfile(int userID)
{
    return LocalCache.Get<UserProfile>(
        cacheKey: "UserProfile:" + userID,
        expiration: TimeSpan.FromHours(1),
        loader: delegate {
            using (var connection = new SqlConnection("..."))
            {
                // load profile by userID from DB
            }
        }
    );
}
```

# 分布式缓存

Web 应用程序可能需要为成百上千甚至更多的用户同时提供服务。如果你没有采取必要的措施，在这种负载下，你的网站可能会崩溃或变得没有响应。

假设在主页显示最后 10 条新闻，并且平均每分钟有上千名用户访问此页面。你可能为每个用户通过查询数据库来显示页面视图信息：

```
SELECT TOP 10 Title, NewsDate, Subject, Body FROM News ORDER BY  
NewsDate DESC
```

即使我们认为主页只包含这些信息，但网站每分钟有 10000 个访客，每秒运行 150 次 SQL 查询。

如果这些查询的结果在用户间没有太大的不同（总是最后 10 条新闻），可以自动缓存在 SQL 服务器端。

但查询结果从 SQL 服务器传输到 WEB 服务器会消耗一些宝贵的网络带宽。由于数据传输需要一些时间（数据大小/带宽），并且在此期间内保持连接处于打开状态，即使 SQL 服务器响应及时，得到的结果也不会太快。传输的时间与新闻内容的大小可能不同。

况且 SQL 连接保持开启的数目也有上限（连接池限制），当达到这一上限，连接便开始在队列中等待并相互阻塞。

考虑到新闻不是每一秒都在改变，我们可以把它们缓存在 WEB 服务器内存 5 分钟。

因此，从 SQL 数据库传输得到新闻列表就立即将它们存储在本地缓存中。在接下来的 5 分钟内，每个用户访问的主页，其新闻列表都是从本地缓存中即时获取的，甚至不用使用 SQL：

```
public List<News> GetNews()
{
    var news = HttpRuntime.Cache["News"] as List<News>;
    if (news == null)
    {
        using (var connection = new SqlConnection("....."))
        {
            news = connection.Query<News>(
                "SELECT TOP 10 Title, NewsDate, Subject, Body
                 FROM News
                 ORDER BY NewsDate DESC")
                .ToList();
        }

        HttpRuntime.Cache.Insert("News", ...,
            TimeSpan.FromMinutes(5), ...);
    }
}

return news;
}
```

这使我们从每秒 150 次查询下降到每秒 1/300 次查询（每 300 秒查询一次）。

此外，这些新闻项目应为每一位访客转换为 HTML。更进一步，我们还可以缓存转换为 HTML 状态的新闻。

所有这些缓存信息都存储在 WEB 服务器内存中，内存是访问这些信息的最快位置。

请注意，缓存信息并不总是意味着你的应用程序将工作得更快。如何有效地使用缓存比单独缓存更重要。如果没有正确使用缓存，它甚至有可能拖慢你的应用程序。

## 网站群和缓存

现在假设我们有一个社交网站，有数以百万的用户简介，一些著名用户的简介页面每分钟有数百或数千人访问。

要生成一个用户简介，需要多个 SQL 查询（朋友、相册名称及照片总数、简介信息、最后状态等）。

只要用户没有更新个人资料，在个人资料页显示的信息几乎是静态的。因此，个人资料页的快照可以缓存 5 分钟或 1 小时等。

但这或许还不够。我们正在谈论数以百万的简介和用户。用户不会只查看一些简介页。我们需要将多台服务器分布在全球的各个位置（WEB 网站群）。

在特定的时间，所有服务器可能在本地缓存中缓存一个贵宾 (VIP) 简介。当 VIP 更改了简介，所有服务器应更新本地缓存简介，并且这要在几秒内发生。现在，我们有服务器（而不是用户）的负载问题。

实际上，一旦服务器从 SQL 数据库加载并缓存 VIP 简介，其他服务器可以使用相同的信息而不用访问数据库。但是，由于每个服务器在其本地内存中存储缓存信息，其他服务器要访问此信息可不是小事。

如果我们有所有服务器可以访问的共享内存：

Information key	Value
Profile:VeryFamousOne	(Cached information for VeryFamousOne)
Profile:SomeAnother	...
...	...
...	...
Profile:JohnDoe	...

我们称之为内存分布式缓存（memory the distributed cache）。如果所有服务器在尝试查询 DB 之前检索该公共内存，我们就可避免服务器的负载问题。

```

public CachedProfileInformation GetProfile(string profileID)
{
    var profile = HttpRuntime.Cache["Profil:" + profileID]
        as CachedProfileInformation;

    if (profile == null)
    {
        profile = DistributedCache.Get<CachedProfileInformation>
(
    "Profil:" + profileID);

        if (profile == null)
        {
            using (var connection = new SqlConnection("....."))
            {
                profile = GetProfileFromDBWithSomeSQLQueries(pro
fileID)
                    profile, TimeSpan.FromMinutes(5));

                DistributedCache.Set("Profil:" + profileID, prof
ile,
                    TimeSpan.FromHours(1));
            }
        }
    }

    return news;
}

```

你可以找到许多不同的分布式缓存系统，包括 Memcached、Couchbase 和 Redis。它们也被称为 NoSQL 数据库。你可以简单地认为它们只是一个远程字典（remote dictionary）。它们将键/值对存储在内存中以使你尽可能快地访问。

**警告！**当使用得当时，分布式缓存就像本地缓存那样可以提高应用程序的性能。而使用不当时，由于涉及网络传输和序列化成本，它可能有比使用本地缓存更糟糕的效果，所以“如果使用分布式缓存，我们的网站将会跑得快”将会是一个神话。

当缓存的数据变得越来越多时，计算机内存可能不足以存储所有的键/值对。在这种情况下，服务器（如 **memcached**）可通过向集群分发数据。可以通过键的第一个字母完成：其中一台服务器存储以 A 开头的键值对，另一台以 B 开头，以此类推。事实上，为了实现该目的它们使用哈希值的键。

# IDistributedCache 接口

[命名空间: *Serenity.Abstractions*, 程序集: *Serenity.Core*]

所有 NoSQL 服务器类型提供了一个类似的接口，像"使用该键存储此值"、"给我该键对应的值"等。

Serenity 通过一个没有依赖特定 NoSQL 数据库类型的通用接口提供分布式缓存的支持：

```
public interface IDistributedCache
{
    long Increment(string key, int amount = 1);
    TValue Get<TValue>(string key);
    void Set<TValue>(string key, TValue value);
    void Set<TValue>(string key, TValue value, TimeSpan expiration);
}
```

Set 方法的第一个重载使用参数 **key** 和 **value** 在分布式缓存中存储键/值对。

```
IoC.Resolve<IDistributedCache>().Set("someKey", "someValue");
```

然后我们可以通过 Get 方法读取该值：

```
var value = IoC.Resolve<IDistributedCache>().Get<string>("someKey")
    // someValue
```

如果想要值保存预定的时间，可以使用 Get 方法的第二个重载：

```
IoC.Resolve<IDistributedCache>().Set("someKey", "someValue",
    TimeSpan.FromMinutes(10));
```

## IDistributedCache.Increment 方法

分布式缓存系统上的操作通常不是原子性的，并且没有提供任何事务性的系统。

相同的键值可以由多个服务器同时修改，并且按随机顺序重写各自的值。

假设我们需要一个独特的计数器（生成 ID）并通过分布式缓存同步它（以防止使用相同的 ID）：

```
int GetTheNextIDValue()
{
    var lastID = IoC.Resolve<IDistributedCache>().Get("LastID");
    IoC.Resolve<IDistributedCache>().Set("LastID", lastID + 1);
    return lastID;
}
```

这个代码块不会像预期那样运行。在读取 `LastID` 值 (get) 和设置自增 `LastID` 值 (set) 的期间，另一台服务器可能读过相同的 `LastID` 值。因此两个服务器可能使用相同的 ID 值。

为了该目的，可以使用 `Increment` 方法：

```
int GetTheNextIDValue()
{
    return IoC.Resolve<IDistributedCache>().Increment("LastID");
}
```

`Increment` 方法的行为就像在线程同步中使用的 `Interlocked.Increment` 方法。它增加标识值，但在增加的过程中会阻止其他请求，然后返回递增的值。所以即使两个 WEB 服务器在同一时刻递增相同的键，他们最终得到不同的 ID 值。

## 静态 **DistributedCache** 类

[命名空间: *Serenity*, 程序集: *Serenity.Core*]

**DistributedCache** 类提供访问当前注册的 **IDistributedCache** 实现的快捷方法。所以，下面两行代码有相同的功能：

```
IoC.Resolve<IDistributedCache>().Increment("LastID");
DistributedCache.Increment("LastID");
```

# DistributedCacheEmulator 类

[命名空间: *Serenity.Abstractions*, 程序集: *Serenity.Core*]

如果你现在不需要分布式缓存，但希望现在编写的代码在将来可以与分布式缓存一起工作，你可以使用 **DistributedCacheEmulator** 类。

**DistributedCacheEmulator** 也对单元测试和部署环境非常有用（因此，开发人员不需要访问分布式缓存系统而不会影响彼此的工作）。

**DistributedCacheEmulator** 模拟 **IDistributedCache** 接口以线程安全的方式使用内存中的字典。

要使用 **DistributedCacheEmulator**，需要使用 **Serenity** 服务定位器 (**IDependencyRegistrar**) 注册它。我们在应用程序启动时(*global.asax.cs* 等)调用一些方法做此事：

```
private static void InitializeDependencies()
{
    // ...
    var registrar = Dependency.Resolve<IDependencyRegistrar>();
    registrar.RegisterInstance<IDistributedCache>(new DistributedCacheEmulator());
    // ...
}
```

# CouchbaseDistributedCache 类

[命名空间: *Serenity.Caching*, 程序集: *Serenity.Caching.Couchbase*]

Couchbase 是一个分布式数据库，有像 Memcached 的访问接口。

可以从 NuGet 程序包 *Serenity.Caching.Couchbase* 获取 Serenity 对此服务类型的实现。

一旦你使用服务定位器注册它：

```
Dependency.Resolve<IDependencyRegistrar>()
    .RegisterInstance<IDistributedCache>(new CouchbaseDistribute
dCache())
```

就可以在应用程序配置文件（使用 JSON 格式）配置 CouchbaseDistributedCache：

```
<appSettings>
    <add key="DistributedCache" value='{
        ServerAddress: "http://111.22.111.97:8091/pools",
        BucketName: "primary-bucket",
        KeyPrefix: ""
    }' />
```

这里的 ServerAddress 是 Couchbase 服务器地址，BucketName 是 bucket 名称。

如果想为多个应用程序使用相同的 server / bucket，可以在 KeyPrefix 设置，如 DEV: 、 TEST: 。

# RedisDistributedCache 类

[命名空间: *Serenity.Caching*, 程序集: *Serenity.Caching.Couchbase*]

Redis 是另一种内存数据库，由于其优秀的性能和可靠性， StackOverflow 也在使用它，他们所有的 WEB 服务只用了一个 Redis 数据库。

你可以从 Serenity.Caching.Redis 的 NuGet 程序包获取该服务类型的 Serenity 实现。

它可以像基于云分布式缓存（CouchbaseDistributedCache）那样注册，并且配置非常类似（虽然没有大量的配置）：

```
<appSettings>
    <add key="DistributedCache" value="{
        ServerAddress: 'someredisserver:6379',
        KeyPrefix: ''
    }"/>
/>
```

## 二级缓存

当你使用本地（在内存中）缓存时，服务器可以缓存一些信息并快速地检索它，但是其他服务器不能访问这个缓存数据，他们需要到数据库中查询同样的信息。

如果你喜欢使用分布式缓存让其他服务器访问缓存的数据，由于它有一些序列化/反序列化和网络延迟开销，则需要注意：在某些情况下，它可能会降低性能。

缓存需要处理的另一个问题：缓存失效。

There are only two hard things in Computer Science: cache invalidation and naming things.

-- Phil Karlton

当你缓存一些信息时，你需要确保当源数据更改后，让缓存信息失效（重新生成或者从缓存中移除）。

# 同步本地和分布式缓存

我们可以通过下面的简单算法实现该目的：

1. 检查本地缓存的键(key)；
2. 如果本地缓存存在该键，则返回它的值；
3. 如果本地缓存不存在该键，则尝试在分布式缓存中找；
4. 如果分布式缓存存在该键，则返回它的值并把它添加到本地缓存；
5. 如果分布式缓存不存在该键，则从数据库中获取，并添加到本地和分布式缓存，最后返回该值。

当在本地缓存服务器中缓存一些信息时，使用这种方式，它还将信息缓存到分布式缓存，但这一次，如果其他服务器在内存中没有该数据副本，则可以在分布式缓存中重用信息。

一旦所有服务器都有本地副本，就不再需要访问分布式缓存，因此，避免序列化和延迟的开销。

## 验证本地副本

看起来一切都很好。但现在我们有一个缓存失效的问题：如果其中一台服务器的缓存数据发生改变了，我们如何把变化通知给其他服务器，以使其本地缓存的副本失效？

我们会在分布式缓存中更改值，但是由于它们不再检查分布式缓存（从最后一个算法中的步骤 2 的可知道），他们不会被通知。

对于这个问题，一种解决办法是保持本地副本一段时间，例如 5 秒。因此，当服务器更改缓存数据时，其他服务器都将使用过时的信息（通常是 5 秒）。

此方法对于反复请求相同缓存信息的批处理操作非常有用。但即使分布式缓存没有发生任何改变，也必须每隔 5 秒从分布式缓存获取副本到本地缓存中。如果缓存的数据是非常大，这会增加网络带宽和反序列化成本。

我们需要一种方法来获得分布式缓存中的数据是否不同于本地副本的信息。有几种我可以想到的方法：

- 将哈希与数据一起存储在本地和分布式缓存（有轻微的哈希计算成本）

- 存储含递增版本的数据（如何确保两台服务器不会产生相同的版本号？）
- 在分布式缓存中存储最后设置数据的时间（时间同步问题）
- 数据与随机数（代）一起存储

Serenity 使用代数（随机的整数）作为版本号。

所以当我们在分布式缓存存储值时，比方说是 `SomeCachedKey`，我们还存储含键 `SomeCachedKey$GENERATION$` 的随机数。

现在，之前的算法变为：

1. 检查本地缓存的键；
2. 如果本地缓存存在该键，
  - 与分布式缓存中的代数比较，
  - 如果相等，返回本地缓存值；
  - 如果不相等，继续步骤 4；
3. 如果本地缓存不存在该键，则尝试在分布式缓存中找；
4. 如果分布式缓存存在该键，则返回它的值并把它添加到本地缓存；
5. 如果分布式缓存不存在该键，则从数据库中获取，并添加到本地和分布式缓存，最后返回该值。

## 一次验证多个缓存项目

你可能已经从一些表中产生了缓存数据。在此表的分布式缓存中可能有多个键。

假设有一张简介（profile）表，并通过它们的 `UserID` 值缓存简介项目。

当用户的简介信息发生变化时，你可以尝试从缓存中删除简介信息。但是如果你不知道的其他服务器或应用程序缓存同一用户的什么简介数据呢？你可能不知道在分布式缓存中缓存什么信息键，它取决于一些 `userID`。

大多数分布式缓存的实现并没有提供以字符串开头查找所有键的方法，否则其将耗费大量的计算（因为它们是基于字典）。

所以当你想要根据一些数据集的日期让所有的项目过期，它是不可行的。

当缓存项目时，Serenity 允许你指定组键（group key），用于在组的数据发生变化时使之过期失效。

比方说一个应用程序从ID是17的用户简介数据生成 `CachedItem17`，并使用此ID作为一个组键 (`Group17_Generation`)：

Key	Value
<code>CachedItem17</code>	cxyzyxzcasd
<code>CachedItem17_Generation</code>	13579
<code>Group17_Generation</code>	13579

在这里，组随机生成（版本）是13579。随着缓存数据(`CachedItem17`)，我们存储产生此数据(`CachedItem17_Generation`)的任何组代。

假设另一台服务器，缓存来自用户17的数据 `AnotherItem17`：

Key	Value
<code>CachedItem17</code>	cxyzyxzcasd
<code>CachedItem17_Generation</code>	13579
<code>AnotherItem17</code>	uwsdasdas
<code>AnotherItem17_Generation</code>	13579
<code>Group17_Generation</code>	13579

这里，我们重用 `Group17_Generation`，因为在分布式缓存中已经有一组版本号，否则，我们将必须生成一个新的。

现在，缓存的两个项目 (`CachedItem17` 和 `AnotherItem17`) 都有效，因为它们的版本号匹配组版本号。

如果有人修改了用户17的数据，我们想让其相关的缓存项目过期失效，只需修改组代。

Key	Value
<code>CachedItem17</code>	cxyzyxzcasd
<code>CachedItem17_Generation</code>	13579
<code>AnotherItem17</code>	uwsdasdas
<code>AnotherItem17_Generation</code>	13579
<code>Group17_Generation</code>	54237

现在，所有的缓存项目都过期失效了。即使它们还存在缓存中，我们可以看到它们的代不匹配组代，因此认为它们是无效的。

我们使用的组键通常是产生数据的表名称。

# TwoLevelCache 类

[命名空间: *Serenity*] - [程序集: *Serenity.Core*]

开箱即用，TwoLevelCache 提供了我们讨论的所有甚至更多的功能。

```

public static class TwoLevelCache
{
    public static TItem Get<TItem>(
        string cacheKey, TimeSpan expiration,
        string groupKey, Func<TItem> loader)
        where TItem : class;

    public static TItem Get<TItem>(
        string cacheKey, TimeSpan localExpiration, TimeSpan
remoteExpiration,
        string groupKey, Func<TItem> loader)
        where TItem : class;

    public static TItem GetWithCustomSerializer<TItem, TSeri
alized>(
        string cacheKey, TimeSpan localExpiration, TimeSpan
remoteExpiration,
        string groupKey, Func<TItem> loader,
        Func<TItem, TSerialized> serialize,
        Func<TSerialized, TItem> deserialize)
        where TItem : class
        where TSerialized : class;

    public static TItem GetLocalStoreOnly<TItem>(
        string cacheKey, TimeSpan localExpiration,
        string groupKey, Func<TItem> loader)
        where TItem : class;

    public static void ChangeGlobalGeneration(string globalG
enerationKey);
    public static void Remove(string cacheKey);
}

```

## TwoLevelCache.Get 方法

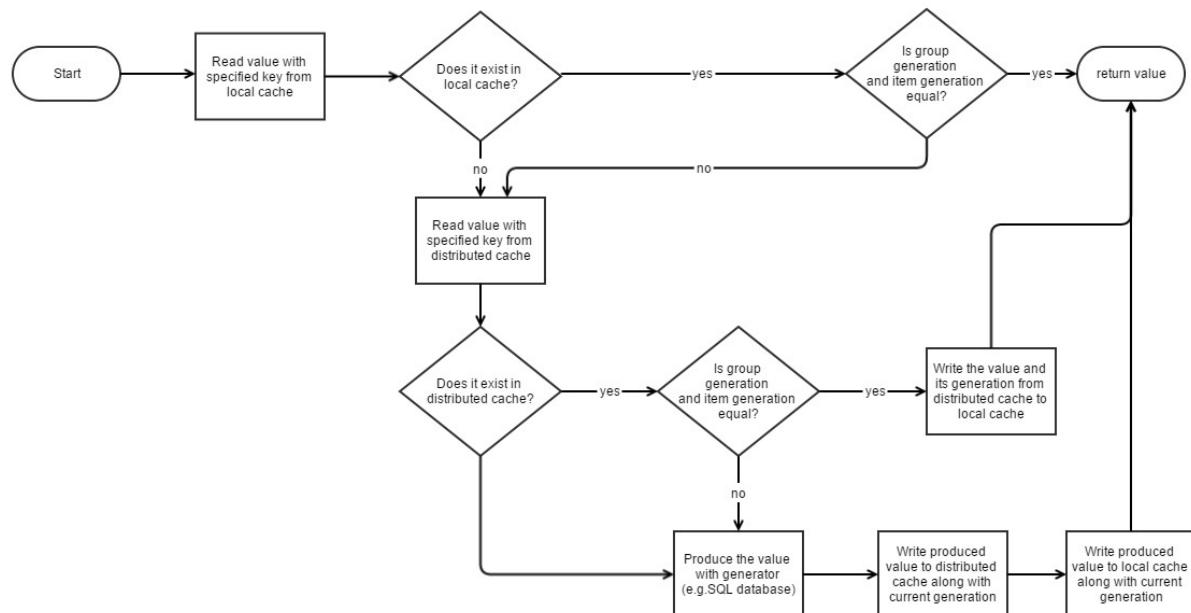
- 试图从本地缓存中读取值。如果在本地缓存中找不到值（或已过期的版本），则尝试从分布式缓存中读取值。

- 如果都不包含指定的键，通过调用加载函数生成值，并指定过期时间且将值添加到本地和分布式缓存。
- 这里有两个 `Get` 方法的重载：一个分别使用本地和分布式缓存过期时间；另一个让两种缓存方式都使用一个过期参数。
- 通过使用一个组键（group key），可以一次性让该组上的两种缓存类型的所有成员都过期（这一组缓存信息都过期了）。

为了避免组代（group generation）每次都检查是否包含该项目，组代本身也缓存在本地缓存中。因此，当组成员改变时，本地缓存的项目会在 5 秒后过期。

这意味着，如果你在组织网站群中使用这种策略，当其中一台服务器的缓存发生更改时，其他服务器可能会继续使用旧的本地缓存的数据（有 5 秒的延迟）。

如果这是配置问题，应该直接使用 `DistributedCache` 方法而不是依赖于 `TwoLevelCache`。



```

CachedProfile GetCachedProfile(int userID)
{
    TwoLevelCache.Get("CachedProfile:" + userID, TimeSpan.FromDays(1), "SomeGroupKey",
        () =>
    {
        using (var connection = new SqlConnection("..."))
        {
            connection.Open();
            return LoadProfileFromDB(connection, userID);
        }
    });
}

CachedProfile LoadProfileFromDB(IDbConnection connection, int userID)
{
    // ...
}

```

## TwoLevelCache.GetWithCustomSerializer 方法

TwoLevelCache.Get 将缓存的数据存储在本地和分布式缓存。当在本地缓存存储缓存项目时，不需要序列化（在内存）。但是在项目发送到分布式缓存之前，必须执行（取决于提供者和数据类型）一些序列化（二进制、json 等）。

有时该序列化/反序列化操作是昂贵的，所以你可能想自己实现数据类型序列化的功能。

GetWithCustomSerializer 有两个额外的序列化和反序列化委托参数。可以从序列化函数返回一个字符串或字节数组，并且在反序列化过程中使用此字符串或字节数组将其转换回原来的数据类型。

大多数提供者可有效地处理简单的类型，如 int、string 或 byte[]，所以对于这种数据类型，你不需要自定义序列化。

## TwoLevelCache.GetLocalStoreOnly 方法

如果你只想在本地缓存中存储数据，可以使用 `GetLocalStoreOnly` 方法。

当缓存的数据由一台服务器存储时，该数据对其他服务器没有帮助（从服务器到服务器的更改），因此，在分布式缓存中存储大的或者慢的序列化/反序列化数据是没有意义的。

因此，在这种情况下，为什么不直接使用本地缓存呢？

可以这么做。但是如果想指定一组键（group key），而且当该组的源数据改变时，本地缓存项目可以很容易过期（就好像它们存储在分布式缓存），就不应该这么做。

## TwoLevelCache.ExpireGroupItems 方法

此方法允许使一个组键（group key）的所有成员都过期。它只是从本地和分布式缓存中删除组键，因此在下一次查询时将会生成另一个版本的缓存数据。

```
TwoLevelCache.ExpireGroupItems("SomeGroupKey");
```

应该在修改数据的方法中调用此方法。

如果实体类有 `TwoLevelCached` 特性，`Create`、`Update`、`Delete` 和 `Undelete` 使用 `ConnectionKey.TableName` 作为组键自动做缓存过期处理。

## TwoLevelCache.Remove 方法

从本地和分布式缓存中移除项目和其版本。

# 实体(**Row**)

Serenity 实体系统是一个微 ORM，它像 Dapper 那样对 SQL 情有独钟。

不同于完全成熟的ORM，如 NHibernate/Entity Framework，Serenity 提供智能感知地映射和查询数据库所需的最小功能、编译时检查和容易的重构。

Serenity 的实体通常像 *XYZRow* 这样命名。它们是 *Serenity.Data.Row* 子类。

让我们来定义一个简单的行类：

```
using Serenity;
using Serenity.ComponentModel;
using Serenity.Data;

public class SimpleRow : Row
{
    public string Name
    {
        get { return Fields.Name[this]; }
        set { Fields.Name[this] = value; }
    }

    public Int32? Age
    {
        get { return Fields.Age[this]; }
        set { Fields.Age[this] = value; }
    }

    public static RowFields Fields = new RowFields().Init();

    public SimpleRow()
        : base(Fields)
    {
    }

    public class RowFields : RowFieldsBase
    {
        public StringField Name;
        public Int32Field Age;
    }
}
```

是的，与一个简单的 POCO 类相比，它看起来有点复杂。它不需要像一些 Orm (实体框架，NHibernate 等) 那样使用代理类来让一些功能工作。

这种结构使我们能够用零反射构建查询，在需要时启用 `INotifyPropertyChanged` 跟踪作业。它也可以与用户自定义的字段一起工作。`Row` 被序列化为 JSON，这样就可以从服务器返回。你不需要额外的 POCO/DTO 类，除非你有使用它们的好理由。

让我们先学习行的声明部分。

```
public class SimpleRow : Row
```

我们在这里定义了一个名为 `SimpleRow` 的实体，它可能映射数据库中名为 `Simple` 的表。

这里的 `Row` 后缀不是必须的，但通常加上该后缀可避免与其他的类名称发生冲突。

所有的实体类继承自 `Serenity.Data.Row` 基类。

```
public string Name
{
    get { return Fields.Name[this]; }
    set { Fields.Name[this] = value; }
}
```

我们现在声明第一个属性。此属性映射到数据库的 `Simple` 表 `Name` 列。

这里不能使用自动属性 (如 `get; set;`)。必须通过称为 `Field` 的特定对象来读取和设置字段的值。

`Field` 对象非常类似于 WPF 的依赖属性。下面是依赖属性的声明示例：

```

public static readonly DependencyProperty MyCustomProperty =
    DependencyProperty.Register("MyCustom", typeof(string), type
of(Window1));

public string MyCustom
{
    get { return this.GetValue(MyCustomProperty) as string; }
    set { this.SetValue(MyCustomProperty, value); }
}

```

我们在这里定义了一个静态依赖属性对象(`MyCustomProperty`)，它包含属性元数据可以允许我们通过 `GetValue` 和 `SetValue` 方法读取和设置属性的值。依赖属性允许 WPF 提供验证、数据绑定、动画及更多的功能。

类似于依赖属性，`Field` 对象包含列的元数据和一些如任务跟踪（`assignment tracking`） 、不使用表达式树构建的查询、变更通知等辅助功能。

虽然依赖属性被声明为所使用类的静态成员，`Field` 对象在一个名为 `RowFields` 的嵌套类中声明。这样可更容易分组和获取引用，而不用添加 `Field` 或 `Property` 后缀，并从实体中保持清晰的字段声明。

```

public Int32? Age
{
    get { return Fields.Age[this]; }
    set { Fields.Age[this] = value; }
}

```

这是我们的第二个属性：`Int32?` 类型的 `Age`。

Serenity 实体属性始终是可空的，即使数据库中的列类型不是可空类型。

Serenity 从不会在 `null` 的位置使用 `0`。

如果你有使用其他 ORM 的经验，会发现这似乎不合逻辑，但请考虑这种情况：

一个非空字段不可能有 `null` 值，但如果通过 `left/right` 联接查询呢？在这种情况下，如果检索到的值是 `null` 或 `0`，你要怎么处理呢？

引用类型已经是可空类型，所以你不能使用 `String?`。

```
public static RowFields Fields = new RowFields().Init();
```

我们注意到，Field 对象在名为 *RowFields*（通常）嵌套子类中声明。在这里，我们创建它的唯一静态实例。因此，每个行类型(row type)只有一个 RowFields 实例，并且每个行属性(row property)只有一个 Field 实例。

`Init` 是初始化 *RowFields* 成员的扩展方法，它将创建没有显式初始化的 Field 对象。

```
public SimpleRow()
    : base(Fields)
{}
```

现在我们定义 SimpleRow 带参数的构造函数。基类 *Row* 需要一个 RowFields 实例，我们为其传递静态对象 *Fields*。因此，行类型(SimpleRow)的所有实例共享单个 RowFields (SimpleRow.RowFields) 实例。这意味着它们共享所有元数据。

```
public class RowFields : RowFieldsBase
{
    public StringField Name;
    public Int32Field Age;
}
```

我们在这里定义包含字段对象的嵌套类。它继承自 `Serenity.Data.RowFieldsBase`。`RowFieldsBase` 是一个与 *Row* 相关的特殊类，包含表的元数据。

我们分别声明一个 *StringField* 和 *Int32Field* 类型的字段。它们的类型基于其属性的类型，并且必须完全匹配。

它们的名称必须与属性名称匹配，否则你将得到一个初始化错误。

我们没有初始化这些字段对象，因此它们的初始值为 `null`。

记得我们上面写的 `new RowFields().Init()`，这是字段对象自动创建的地方。

也可以在 `RowFields` 构造函数中手动初始化它们，但是除了需要特殊定制，不建议这样做。

# Mapping 特性

Serenity 提供一些映射特性，以匹配数据库的表、行的列名称。

## 列与表的映射约定

默认情况下，行(row)类移除 *Row* 后缀后，被认为匹配数据库中具有相同名称的表。

属性被认为匹配数据库中具有相同名称的列。

比方说我们有这样一个行定义：

```
public class CustomerRow : Row
{
    public string StreetAddress
    {
        get { return Fields.StreetAddress[this]; }
        set { Fields.StreetAddress[this] = value; }
    }
}
```

如果我们写一个从 *CustomerRow* 中选择 *StreetAddress* 字段的查询，它就会产生下面语句：

```
SELECT
T0.StreetAddress AS [StreetAddress]
FROM Customer T0
```

根据约定，*CustomerRow* 匹配 *Customer* 表。类似地，*StreetAddress* 属性匹配名为 *StreetAddress* 的列。

*T0* 是一个特殊的别名，由 Serenity 行分配给主表。

由于 *StreetAddress* 列属于主表(*Customer*)，它具有选择表达式 *T0.StreetAddress*，并且列别名为 *[StreetAddress]*。

默认情况下使用属性名称作为列别名。

## SqlSettings.AutoQuotedIdentifiers 标识

在一些数据库系统，标识符是大小写敏感的。

例如在 Postgres 中，如果创建一个带引号的标识符列 "StreetAddress"，当查找该列时，你必须使用引号。即使你这样写 SELECT StreetAddress ... (相同的大小写)，它也不会工作。

你必须使用这种形式 SELECT "StreetAddress"。

因此，Postgres 用户通常喜欢用小写字母的标识符。但 FluentMigrator 总是向标识符添加引号，所以我们需要一种变通方法来向标识符添加方括号/引号。

默认情况下，Serenity 不会向列和表名添加方括号/引号，但是它有兼容的设置。

如果 `SqISettings.AutoQuotedIdentifiers` 标识设置为 `true`，上面的查询将会变为：

```
SELECT
T0.[StreetAddress] AS [StreetAddress]
FROM [Customer] T0
```

Serenity 为了向后兼容，此标识设置默认为 `false`，但 Serene 1.8.6+ 在应用程序启动时把该标识设置为 `true`。

如果我们使用 Postgres 方言，将输出：

```
SELECT
T0."StreetAddress" AS "StreetAddress"
FROM "Customer" T0
```

## Column 特性

[命名空间: `Serenity.Data.Mapping`] - [程序集: `Serenity.Data`]

使用 `Column` 特性，可以把属性映射为数据库的一些其它列名称。

```
public class CustomerRow : Row
{
    [Column("street_address")]
    public string StreetAddress
    {
        get { return Fields.StreetAddress[this]; }
        set { Fields.StreetAddress[this] = value; }
    }
}
```

现在查询将变为：

```
SELECT
T0.street_address AS [StreetAddress]
FROM Customer T0
```

也可以手动添加方括号：

```
public class CustomerRow : Row
{
    [Column("[street_address]")]
    public string StreetAddress
    {
        get { return Fields.StreetAddress[this]; }
        set { Fields.StreetAddress[this] = value; }
    }
}
```

```
SELECT
T0.[street_address] AS [StreetAddress]
FROM Customer T0
```

如果 `SqlSettings.AutoQuotedIdentifiers` 为 `true`，将自动添加方括号。

如果需要使用多个数据库类型，请使用 `SqlServer` 特定的方括号( `[]` )。这些方括号在运行查询之前，将转换为方言特定的引号（双引号，引号等）。

但是，如果只针对一种类型的数据库，你可能更喜欢使用特定于该数据库类型的引号。

## TableName 特性

[命名空间: `Serenity.Data.Mapping`] - [程序集: `Serenity.Data`]

如果数据库的表名称不同于行(row)的类名，请使用此特性：

```
[TableName("TheCustomers")]
public class CustomerRow : Row
{
    public string StreetAddress
    {
        get { return Fields.StreetAddress[this]; }
        set { Fields.StreetAddress[this] = value; }
    }
}
```

```
SELECT
T0.StreetAddress AS [StreetAddress]
FROM TheCustomers T0
```

也可以使用方括号或引号：

```
[TableName("[My Customers]")]
public class CustomerRow : Row
{
    public string StreetAddress
    {
        get { return Fields.StreetAddress[this]; }
        set { Fields.StreetAddress[this] = value; }
    }
}
```

```
SELECT
T0.StreetAddress AS [StreetAddress]
FROM [My Customers] T0
```

此外，更喜欢兼容数据库的方括号。

## Expression 特性

[命名空间: *Serenity.Data.Mapping*] - [程序集: *Serenity.Data*]

此特性用于指定一个非基本 (non-basic) 字段的表达式，例如，一个实际上不存在于数据库中的字段。

此类字段可以有几种类型。

其中一个例子是：含计算表达式 `(T0.[Firstname] + ' ' + T0.[Lastname])` 的 `Fullname` 字段。如：

```

public class CustomerRow : Row
{
    public string Firstname
    {
        get { return Fields.Firstname[this]; }
        set { Fields.Firstname[this] = value; }
    }

    public string Lastname
    {
        get { return Fields.Lastname[this]; }
        set { Fields.Lastname[this] = value; }
    }

    [Expression("(T0.[Firstname] + ' ' + T0.[Lastname]))"]
    public string Fullname
    {
        get { return Fields.Fullname[this]; }
        set { Fields.Fullname[this] = value; }
    }
}

```

小心 "+" 运算符，因为在这里它是 Sql Server 特定的运算符。如果想要使用多种类型的数据库，你应该以这种方式写表达式：

```
CONCAT(T0.[Firstname], CONCAT(' ', T0.[Lastname]))
```

**Firstname** 和 **Lastname** 是表字段（表的实际字段），但即使没有 **Expression** 特性，它们也有基本的隐式定义的表达式：**T0.Firstname** 和 **T0.Lastname**（在 Serenity 查询中，主表指定使用 **T0** 作为别名）。

在此文档中，当我们谈论一个 表字段 时，是指数据库表中实际对应的列字段。

视图字段(*View Field*) 是指具有计算表达式的字段，或另一张表的字段（如 SQL 视图中来自联接的字段）。

Fullname 表达式中使用 **T0** 别名引用字段。

在没有该前缀的情况下也可能工作。但最好使用前缀。当你开始添加联接时，可能有多个字段具有相同名称而导致列不明确的错误。

## ForeignKey 特性

[命名空间: *Serenity.Data.Mapping*] - [程序集: *Serenity.Data*]

该特性用于指定外键列，并添加主表和相关主字段信息。

```
public class CustomerRow : Row
{
    [ForeignKey("Countries", "Id")]
    public string CountryId
    {
        get { return Fields.Firstname[this]; }
        set { Fields.Firstname[this] = value; }
    }
}
```

我们这里指定 *Customer* 表的 *CountryId* 字段有一个外键是 *Countries* 表的 *Id* 字段。

外键并不需要在数据库中存在。Serenity 不会检查它。

Serenity 可以利用这样的元信息，即使它不会影响生成单独的查询。

当我们与即将介绍的下一个特性一起使用时，ForeignKey 将更有意义。

## LeftJoin 特性

当查询数据库时，由于关联关系，往往会展开许多联接。大多数联接是 LEFT 或 INNER 连接。

使用 Serenity 实体，通常会使用 LEFT JOIN。

数据库管理员更喜欢定义视图，使它更容易查询多个表的组合，并避免一次次编写这些联接。

Serenity 实体可以像 SQL 视图那样使用，所以你可以获得实体中其他表的列，并对其进行查询，就好像它们是一个大的组合表。

```
public class CustomerRow : Row
{
    [ForeignKey("Cities", "Id"), LeftJoin("c")]
    public Int32? CityId
    {
        get { return Fields.CityId[this]; }
        set { Fields.CityId[this] = value; }
    }

    [Expression("c.[Name]")]
    public string CityName
    {
        get { return Fields.CityName[this]; }
        set { Fields.CityName[this] = value; }
    }
}
```

我们这里指定 **Cities** 表在关联时应分配别名 **c**，并且其关联类型应该是 **LEFT JOIN**。关联的 **ON** 条件表达式根据 **ForeignKey** 特性确定为 **c.[Id] == T0.CountryId**。

首选 **LEFT JOIN**，因为即使 **Customers** 没有 **CityId** 设置，它也允许检索左 (**left**) 表的所有记录。

**CityName** 是一个视图字段（不是 **Customer** 表的字段），它有一个 **c.Name** 表达式。它表明 **CityName** 来自 **Cities** 表的 **Name** 字段。

现在，如果想要选择所有客户的的城市名称，查询文本将变为：

```
SELECT
    c.Name AS [CityName]
FROM Customer T0
LEFT JOIN Cities c ON (c.[Id] = T0.CityId)
```

如果我们 **Customer** 表没有 **CountryId** 字段，但我们想通过 **city** 表中的 **CountryId** 字段得到城市所属国家的名称，该怎么做呢？

```

public class CustomerRow : Row
{
    [ForeignKey("Cities", "Id"), LeftJoin("c")]
    public Int32? CityId
    {
        get { return Fields.CityId[this]; }
        set { Fields.CityId[this] = value; }
    }

    [Expression("c.[Name]")]
    public string CityName
    {
        get { return Fields.CityName[this]; }
        set { Fields.CityName[this] = value; }
    }

    [Expression("c.[CountryId]"), ForeignKey("Countries", "Id"),
    LeftJoin("o")]
    public Int32? CountryId
    {
        get { return Fields.CountryId[this]; }
        set { Fields.CountryId[this] = value; }
    }

    [Expression("o.[Name]")]
    public string CountryName
    {
        get { return Fields.CountryName[this]; }
        set { Fields.CountryName[this] = value; }
    }
}

```

这次我们在 Cities 表的 CountryId 字段使用 LEFT JOIN。指定 o 作为 Countries 表的别名，并从该表获得国家名称字段。

只要不是保留的关键字，你可以给联接分配任何表别名，并且实体中联接的别名应该是唯一的。Sergen 生成像 jCountry 这样的别名，但你可以用更短和更自然的方式重命名它们。

让我们从所有客户中选择 CityName 和 CountryName 字段：

```
SELECT
    c.[Name] AS [CityName],
    o.[Name] AS [CountryName]
FROM Customer T0
LEFT JOIN Cities c ON (c.[Id] = T0.CityId)
LEFT JOIN Countries o ON (o.[Id] = c.[CountryId])
```

我们将在 FluentSQL 章节看到如何生成这类查询。

到目前为止，我们已经在属性中同时使用 `LeftJoin` 和 `ForeignKey` 特性。

也可以在实体类中附加 `LeftJoin` 特性。这在主实体没有相应字段的关联中非常有用。

例如，假设有一张 `CustomerDetails` 扩展表，存储一些客户的额外信息（一对一关系）。`CustomerDetails` 表有一个主键 `CustomerId`，它实际上是 `Customer` 表 `Id` 字段的外键。

```
[LeftJoin("cd", "CustomerDetails", "cd.[CustomerId] = T0.[Id]")]
public class CustomerRow : Row
{
    [Identity, PrimaryKey]
    public Int32? Id
    {
        get { return Fields.Id[this]; }
        set { Fields.Id[this] = value; }
    }

    [Expression("cd.[DeliveryAddress]")]
    public string DeliveryAddress
    {
        get { return Fields.DeliveryAddress[this]; }
        set { Fields.DeliveryAddress[this] = value; }
    }
}
```

当选择 `DeliveryAddress` 时，在这里它看起来像：

```
SELECT  
cd.[DeliveryAddress] AS [DeliveryAddress]  
FROM Customer T0  
LEFT JOIN CustomerDetails cd ON (cd.[CustomerId] = T0.[Id])
```

# FieldFlags 枚举

[命名空间: *Serenity.Data.Mapping*] - [程序集: *Serenity.Data*]

Serenity 有一组字段标识控制字段行为。

```
public enum FieldFlags
{
    None = 0,
    Insertable = 1,
    Updatable = 2,
    NotNull = 4,
    PrimaryKey = 8,
    AutoIncrement = 16,
    Foreign = 32,
    Calculated = 64,
    Reflective = 128,
    ClientSide = 256,
    Trim = 512,
    TrimToEmpty = 512 + 1024,
    DenyFiltering = 2048,
    Unique = 4096,
    Default = Insertable | Updatable | Trim,
    Required = Default | NotNull,
    Identity = PrimaryKey | AutoIncrement | NotNull
}
```

普通表字段有 *Insertable*、*Updatable* 和 *Trim* 默认设置，对于 *Default* 组合标识。

## Insertable 标识

*Insertable* 标识控制字段在新增记录状态下是否可编辑。默认情况下，所有普通字段都被认为是可插入的。

某些字段可能不是在数据库表中插入的，如，标识(identity) 列不应该有该标识设置。

当字段没有该标识，在新增记录状态的表单下不能编辑该字段。在服务端的仓库层同样有效。

有时候，内部字段可能在 SQL INSERT 声明中完全有效，但不应该在表单中编辑。一个例子是 `InsertedByUserId` 应在服务层面上设置，而不是由终端用户设置。如果我们让终端用户可以在表单中编辑它，这将是一个安全漏洞。此类字段也不应该具有 `Insertable` 标识设置。

这意味着字段标识不需要与数据库表的设置相匹配。

## Insertable 特性

要取消字段的 `Insertable` 标识，可以使用 `[Insertable(false)]` 特性：

```
[Insertable(false)]
public string MyField
{
    get { return Fields.MyField[this]; }
    set { Fields.MyField[this] = value; }
}
```

使用 `Insertable(true)` 启用标识符功能。

非可插入字段在表单中是不隐藏的。它们只是处于只读状态。如果你想隐藏它们，请使用 `[HideOnInsert]` 特性 (Serenity 1.9.8+)，或者使用 `form.MyField.GetGridField().Toggle(IsNew)` 重写对话框中 `UpdateInterface` 方法。

## Updatable 标识

该标识就像 `Insertable` 标识，但是控制表单记录的编辑状态和服务端的更新操作。默认情况下，所有普通字段都被认为是可更新的。

## Updatable 特性

要取消字段的 `Updatable` 标识，可以使用 `[Updatable(false)]` 特性：

```
[Updatable(false)]
public string MyField
{
    get { return Fields.MyField[this];
    set { Fields.MyField[this] = value;
}
```

使用 *Updatable(true)* 启用标识符功能。

非可插入字段在表单中是不隐藏的。它们只是处于只读状态。如果你想隐藏它们，请使用 [HideOnInsert] 特性 (Serenity 1.9.8+)，或者使用 *form.MyField.GetGridField().Toggle(IsNew)* 重写对话框中 *UpdateInterface* 方法。

## Trim 标识

该标识只对字符串类型的字段有效，它控制着值在保存之前是否被修剪。所有字符串默认包含该标识。

当字段值是空字符串或全是空格，它将被修剪为 null。

## TrimToEmpty 标识

如果你更喜欢把字符串字段修剪为空的字符串，而不是 null，则使用此标识。

当字段值是 null 或全是空格，它将被修剪为空字符串。

## SetFieldFlags 特性

此属性可用于在字段中包含或排除一组标识。第一个参数是必需的，表示包含的标识；第二个是可选参数，表示排除的标识。

要启用字段中的 TrimToEmpty 标识，我们可以这样使用：

```
[SetFieldFlags(FieldFlags.TrimToEmpty)]
public string MyField
{
    get { return Fields.MyField[this];
    set { Fields.MyField[this] = value;
}
```

要取消 Trim 标识，则：

```
[SetFieldFlags(FieldFlags.None, FieldFlags.TrimToEmpty)]
public string MyField
{
    get { return Fields.MyField[this];
    set { Fields.MyField[this] = value;
}
```

要包含 TrimToEmpty 和 Updatable，并移除 Insertable，则：

```
[SetFieldFlags(
    FieldFlags.Updatable | FieldFlags.TrimToEmpty,
    FieldFlags.Insertable)]
public string MyField
{
    get { return Fields.MyField[this];
    set { Fields.MyField[this] = value;
}
```

Insertable 和 Updatable 特性是 SetFieldFlags 特性的子类。

## NotNull 标识

使用该标识设置字段为不可空。默认情况下，该标识使用 NotNull 特性设置字段在数据库为不可空字段。

当字段是不可空的，它在表单对应的标签中有一个红色的星号，并要求必须输入值。

## NotNullable 特性

在字段中使用 `NotNull` 特性启用非空限制，移除该特性表示取消该限制。

即使字段在数据库中不是可空的，也可以使用 `[Required(false)]` 让字段在表单中变为非必填字段。它不会清除 `NotNull` 标识。

## Required 标识

这是 `Default` 和 `NotNullable` 标识的组合。

它与表单中控制验证的 `[Required]` 特性没有关系。

## PrimaryKey 标识和 PrimaryKey 特性

为表中的主键字段设置该标识。

列表和检索请求处理程序的 `Key` 列选择模式选择主键字段。

使用 `[PrimaryKey]` 特性启用该标识。

## AutoIncrement 标识和 AutoIncrement 特性

设置此字段在服务器端是自动递增，例如，标识列或使用生成器（generator）的列。

## Identity 标识和 Identity 特性

这是 `PrimaryKey`、`AutoIncrement` 和 `NotNull` 标识的组合，常用于标识列。

## Foreign 标识

该标识设置通过联接其他表得到的外来视图字段。

它自动为字段设置表达式，该表达式包含 `T0` 以外的表别名。

例如，如果一个字段有像 [Expression("jCountry.CountryName")] 这样的特性，它将有此标识。

该标识与 ForeignKey 特性没有关系。

## Calculated 标识

如果字段的表达式有涉及多个字段或一些数学运算，它将有此标识。

也可以为在 SQL 服务器端计算的字段设置此标识。

## ClientSide 标识和 ClientSide 特性

对应于 Serenity 实体未映射的字段。它们没有在数据库表中对应的字段。

这些类型的字段可以用于临时计算、存储及客户端和服务层上的传输。

## Reflective 标识

是用于未映射字段的一种高级形式，这些字段没有存储在行中，但反映了另一个不同形式的字段值。例如，绝对值显示为整数的字段也可以是负整数。

应该只在极少数情况下，对这类未映射的字段使用该标识。

## DenyFiltering 标识

如果设置该标识，则表示拒绝对敏感字段进行过滤操作。对于像 PasswordHash 这样的机密字段很有用，不应该允许由客户端选择或过滤这些机密字段。

## Unique 标识和 Unique 特性

当字段有该标识，在数据库中与现有值进行检查以确保其值必须是唯一的。

你可以使用 Unique 特性启用该标识，并确定是否应在服务级别检查此约束（在数据库级别检查以避免出现神秘约束的错误）。



## 流式 SQL (Fluent SQL)

Serenity 包含一系列 SELECT、INSERT、UPDATE 和 DELETE 语句的查询生成器。

这些生成器可以被简单字符串或 Serenity 实体 (row) 系统使用。

它们的输出可通过像 Dapper (已集成到 Serenity) 这样的微 ORM 或 Serenity 扩展直接执行。

# SqlQuery 对象

[命名空间: *Serenity.Data*] - [程序集: *Serenity.Data*]

SqlQuery 通过一个流式接口编写动态 SQL SELECT 查询。

## 优点

SqlQuery 比手写 SQL 有如下优势：

- 使用 Visual Studio 的智能感知功能编写 SQL；
- 最小开销的流式接口；
- 由于在编译时而不是运行时检查查询语法，所以可减少语法错误；
- 像 Select、Where、Order By 这样的子句可以按任何顺序使用。当查询转换为字符串时，它们被放到正确的位置。类似地，这些子句可以使用多次并且在转换为字符串时被合并。因此你可以根据输入参数有条件地构建 SQL。
- 不再搞砸参数和参数名称。所有使用的值转换为自动命名参数。如果需要，你也可以使用手动命名的参数；
- 可以生成一个特殊的查询，可以在非原生支持分页的服务器类型（如 SQL Server 2000）中执行分页；
- 通过方言系统，可以针对特定的服务器类型和版本进行查询；
- 如果与 Serenity 实体（可以像 Dapper 那样使用的微型 ORM）一起使用，有助于从 DataReader 零反射加载查询结果。它也支持自动左/右联接。

## 如何使用这里的示例

我推荐使用 LinqPad 执行这里给出的示例。

你应该添加 *Serenity.Core*、*Serenity.Data* 和 *Serenity.Data.Entity* 的 NuGet 程序包引用。

另一种做法是从一个 Serene 的应用程序的 *bin* 或程序包目录中找到并直接引用这些 DLL。

请确保在查询属性 (Query Properties) 对话框中的导入额外命名空间 (Additional Namespace Imports) 中添加 *Serenity* 和 *Serenity.Data*。

## 一个简单的 **Select** 查询示例

```
void Main()
{
    var query = new SqlQuery();
    query.Select("Firstname");
    query.Select("Surname");
    query.From("People");
    query.OrderBy("Age");

    Console.WriteLine(query.ToString());
}
```

这将输出结果：

```
SELECT
Firstname,
Surname
FROM People
ORDER BY Age
```

在程序的第一行，我们调用 *SqlQuery* 的唯一无参构造函数。如果此时调用 *ToString()*，将输出：

```
SELECT FROM
```

*SqlQuery* 不会进行任何语法验证。它只是通过调用其方法转换成你自己构建的查询。即使你没有选择任何字段或调用 *From* 方法，它也将生成基本的 SELECT FROM 语句。

*SqlQuery* 不能生成空查询。

然后，我们调用 `Select` 方法，并向其传递 `"FirstName"` 字符串参数。现在我们的查询将变为：

```
SELECT Firstname FROM
```

当执行 `Select("Surname")` 声明时，`SqlQuery` 在上一检索字段（`Firstname`）和当前检索字段之间添加逗号：

```
SELECT Firstname, Surname FROM
```

在执行 `From` 和 `OrderBy` 方法之后，最终输出：

```
SELECT Firstname, Surname FROM People ORDER BY Age
```

## 方法调用顺序和它的影响

在前面的示例，即使我们重新调整 `From`、`OrderBy` 和 `Select` 行的顺序，输出的结果也不会发生变化。只有修改 `Select` 声明的顺序才会改变输出结果。

```
void Main()
{
    var query = new SqlQuery();
    query.From("People");
    query.OrderBy("Age");
    query.Select("Surname");
    query.Select("Firstname");

    Console.WriteLine(query.ToString());
}
```

但是，只有 `SELECT` 语句内部的列顺序会发生改变：

```

SELECT
Surname,
Firstname
FROM People
ORDER BY Age

```

你可以按任意顺序使用 `Select`、`From`、`OrderBy`、`GroupBy` 方法，并且可混合使用（如：先调用 `Select`，然后调用 `OrderBy`，之后再次调用 `Select` ……）。

建议把 `FROM` 放在查询的开头，特别是与 `Serenity` 实体一起使用时，因为这样做有助于自动联接、决定数据库方言等。

## 方法链

每行使用 `query.` 开头显得冗长且可读性比较差。几乎所有的 `SqlQuery` 方法是链式的，并把查询本身作为结果返回。

我们可以像下面这样重写该查询：

```

void Main()
{
    var query = new SqlQuery()
        .From("People")
        .Select("Firstname")
        .Select("Surname")
        .OrderBy("Age");

    Console.WriteLine(query.ToString());
}

```

该功能类似于 `jQuery` 和 `LINQ` 可枚举方法链。

我们甚至可以去掉查询变量：

```
void Main()
{
    Console.WriteLine(
        new SqlQuery()
            .From("People")
            .Select("Firstname")
            .Select("Surname")
            .OrderBy("Age")
            .ToString());
}
```

强烈建议把每个方法放在它自己的行中，且为了可读性和一致性，请使用合适的缩进。

## Select Method

```
public SqlQuery Select(string expression)
```

在前面的示例中，我们使用上述的 **Select** 重载方法（它大约有 11 个重载方法）。

**Expression** 参数可以是简单的字段名称或像 "FirstName + ' ' + LastName" 这样的表达式。

每当调用此方法，设置的表达式以逗号间隔添加到 SELECT 语句的查询结果。

还有一个 **SelectMany** 方法，可以在一个调用中选择多个字段：

```
public SqlQuery SelectMany(params string[] expressions)
```

例如：

```

void Main()
{
    var query = new SqlQuery()
        .From("People")
        .SelectMany("Firstname", "Surname", "Age", "Gender")
        .ToString();

    Console.WriteLine(query.ToString());
}

```

```

SELECT
Firstname,
Surname,
Age,
Gender
FROM People

```

我个人更喜欢通过多次调用 `Select` 方法来实现该目的。

你可能会想：为什么 `SelectMany` 并不是 `Select` 的另一重载？这是因为 `Select` 有一个更为常用的重载，可以选择含别名的列：

```
public SqlQuery Select(string expression, string alias)
```

```

void Main()
{
    var query = new SqlQuery()
        .Select("(Firstname + ' ' + Surname)", "Fullname")
        .From("People")
        .ToString();

    Console.WriteLine(query.ToString());
}

```

```
SELECT
    (Firstname + ' ' + Surname) AS [Fullname]
FROM People
```

## From 方法

```
public SqlQuery From(string table)
```

SqlQuery.From 方法至少应被调用一次（通常一次）。

建议在查询中首先调用该方法。

当第二次调用该方法，表名称将以逗号间隔添加到 FROM 声明。因此，它将变为 CROSS JOIN：

```
void Main()
{
    var query = new SqlQuery()
        .From("People")
        .From("City")
        .From("Country")
        .Select("Firstname")
        .Select("Surname")
        .OrderBy("Age");

    Console.WriteLine(query.ToString());
}
```

```
SELECT
    Firstname,
    Surname
FROM People, City, Country
ORDER BY Age
```

## 在 **SqlQuery** 中使用别名对象

当引用表的数量增加时，通常使用表别名，此时我们的查询变得更长：

```
void Main()
{
    var query = new SqlQuery()
        .From("Person p")
        .From("City c")
        .From("Country o")
        .Select("p.Firstname")
        .Select("p.Surname")
        .Select("c.Name", "CityName")
        .Select("o.Name", "CountryName")
        .OrderBy("p.Age")
        .ToString();

    Console.WriteLine(query.ToString());
}
```

```
SELECT
p.Firstname,
p.Surname,
c.Name AS [CityName],
o.Name AS [CountryName]
FROM Person p, City c, Country o
ORDER BY p.Age
```

虽然它可以这样工作，但可以更好地把 `p`、`c` 和 `o` 定义为 `Alias` 对象。

```
var p = new Alias("Person", "p");
```

`Alias` 对象为表指定简称。它有索引和操作运算符的重载来生成访问 SQL 成员的表达式，如 `p.Surname`。

```
void Main()
{
    var p = new Alias("Person", "p");
    Console.WriteLine(p + "Surname"); // + operator overload
    Console.WriteLine(p["Firstname"]); // through indexer
}
```

```
p.Surname
p.Firstname
```

不幸的是，不能重载 C# 的成员访问运算符 (.)，因此，我们不得不使用 (+)。  
一种替代方法是使用 dynamic，但是它的表现并不佳。

让我们使用 Alias 对象修改查询：

```
void Main()
{
    var p = new Alias("Person", "p");
    var c = new Alias("City", "c");
    var o = new Alias("Country", "o");

    var query = new SqlQuery()
        .From(p)
        .From(c)
        .From(o)
        .Select(p + "Firstname")
        .Select(p + "Surname")
        .Select(c + "Name", "CityName")
        .Select(o + "Name", "CountryName")
        .OrderBy(p + "Age")
        .ToString();

    Console.WriteLine(query.ToString());
}
```

```

SELECT
p.Firstname,
p.Surname,
c.Name AS [CityName],
o.Name AS [CountryName]
FROM Person p, City c, Country o
ORDER BY p.Age

```

如上所示，结果是相同的，但代码有点长。那么使用别名有什么优点呢？

如果我们有一个含字段名称的常量列表：

```

void Main()
{
    const string Firstname = "Firstname";
    const string Surname = "Surname";
    const string Name = "Name";
    const string Age = "Age";

    var p = new Alias("Person", "p");
    var c = new Alias("City", "c");
    var o = new Alias("Country", "o");
    var query = new SqlQuery()
        .From(p)
        .From(c)
        .From(o)
        .Select(p + Firstname)
        .Select(p + Surname)
        .Select(c + Name, "CityName")
        .Select(o + Name, "CountryName")
        .OrderBy(p + Age)
        .ToString();

    Console.WriteLine(query.ToString());
}

```

我们就可以利用智能感知功能和编译时检查。

显然，它不是很符合逻辑并且难以定义每个查询的字段名称。应该在一个集中的位置或实体声明中定义别名。

让我们使用 **Alias** 创建一个人员的简单 ORM：

```
public class PeopleAlias : Alias
{
    public PeopleAlias(string alias)
        : base("People", alias) { }

    public string ID { get { return this["ID"]; } }
    public string Firstname { get { return this["Firstname"]; } }
}
public string Surname { get { return this["Surname"]; } }
public string Age { get { return this["Age"]; } }
}

public class CityAlias : Alias
{
    public CityAlias(string alias)
        : base("City", alias) { }

    public string ID { get { return this["ID"]; } }
    public string CountryID { get { return this["CountryID"]; } }
    public new string Name { get { return this["Name"]; } }
}

public class CountryAlias : Alias
{
    public CountryAlias(string alias)
        : base("Country", alias) { }

    public string ID { get { return this["ID"]; } }
    public new string Name { get { return this["Name"]; } }
}

void Main()
{
    var p = new PeopleAlias("p");
    var c = new CityAlias("c");
```

```

var o = new CountryAlias("o");
var query = new SqlQuery()
    .From(p)
    .From(c)
    .From(o)
    .Select(p.Firstname)
    .Select(p.Surname)
    .Select(c.Name, "CityName")
    .Select(o.Name, "CountryName")
    .OrderBy(p.Age)
    .ToString();

Console.WriteLine(query.ToString());
}

```

现在我们有一组含字段名称和可以在所有查询中重用的表别名类。

这只是一个解释别名的示例，我并不推荐写这样的类。实体(Entities) 提供了更多的功能。

在上面的示例，我们使用包含 *Alias* 参数的 *SqlQuery.From* 重载：

```
public SqlQuery From(Alias alias)
```

当调用此方法时，表名称和它的别名被添加到查询的 *FROM* 子句。

## OrderBy 方法

```
public SqlQuery OrderBy(string expression, bool desc = false)
```

*OrderBy* 也可以在调用时含字段名称或表达式（如，*Select*）。

如果你指定可选参数 *desc* 为 *true*，将在字段名称或表达式附加关键词 *DESC*。

默认情况下，*OrderBy* 附加指定的表达式到 *ORDER BY* 语句末尾。但有时你可能想在起始处插入表达式/字段。

## SqlQuery 对象

例如，有一些预定义顺序的查询，但如果用户在网格中对列进行排序，列名称应被插入到索引为 0 的位置。

```
public SqlQuery OrderByFirst(string expression, bool desc = false)
```

```
void Main()
{
    var query = new SqlQuery()
        .Select("Firstname")
        .Select("Surname")
        .From("Person")
        .OrderBy("PersonID");

    query.OrderByFirst("Age");

    Console.WriteLine(query.ToString());
}
```

```
SELECT
    Firstname,
    Surname
FROM Person
ORDER BY Age, PersonID
```

## Distinct 方法

```
public SqlQuery Distinct(bool distinct)
```

使用此方法在 SELECT 语句预置 DISTINCT 关键字。

```
void Main()
{
    var query = new SqlQuery()
        .Select("Firstname")
        .Select("Surname")
        .From("Person")
        .OrderBy("PersonID")
        .Distinct(true);

    Console.WriteLine(query.ToString());
}
```

```
SELECT DISTINCT
Firstname,
Surname
FROM Person
ORDER BY PersonID
```

## GroupBy 方法

```
public SqlQuery GroupBy(string expression)
```

GroupBy 的行为类似于 OrderBy，但没有 GroupByFirst 变体。

```
SELECT
Firstname,
Lastname,
Count(*)
FROM Person
GROUP BY Firstname, LastName
```

```

SELECT
    Firstname,
    Lastname,
    Count(*)
FROM Person
GROUP BY Firstname, LastName

```

## Having 方法

```
public SqlQuery Having(string expression)
```

Having 可以和 GroupBy（尽管它并不检查 GroupBy）一起使用，并且在表达式末尾追加 HAVING 语句。

```

void Main()
{
    var query = new SqlQuery()
        .From("Person")
        .Select("Firstname")
        .Select("Lastname")
        .Select("Count(*)")
        .GroupBy("Firstname")
        .GroupBy("LastName")
        .Having("Count(*) > 5");

    Console.WriteLine(query.ToString());
}

```

```

SELECT
    Firstname,
    Lastname,
    Count(*)
FROM Person
GROUP BY Firstname, LastName
HAVING Count(*) > 5

```

## 分页操作(**SKIP / TAKE / TOP / LIMIT**)

```
public SqlQuery Skip(int skipRows)
```

```
public SqlQuery Take(int rowCount)
```

SqlQuery 有类似于 LINQ 的 Take 和 Skip 的分页方法。

数据库类型决定映射的 SQL 关键字。

由于 SqlServer 2012 之前的版本没有等效的 SKIP 方法，若要使用 SKIP 方法，你的查询应该至少有一个 ORDER BY 语句，因为需要使用 ROW\_NUMBER()。如果你使用 SqlServer 2012+ 的方言，就没有该要求。

```
void Main()
{
    var query = new SqlQuery()
        .From("Person")
        .Select("Firstname")
        .Select("Lastname")
        .Select("Count(*)")
        .OrderBy("PersonId")
        .Skip(100)
        .Take(50);

    Console.WriteLine(query.ToString());
}
```

```
SELECT
    Firstname,
    Lastname,
    Count(*)
FROM Person
ORDER BY PersonId OFFSET 100 ROWS FETCH NEXT 50 ROWS ONLY
```

在该示例中，默认使用 SQLServer2012 方言。

## 支持数据库方言

在我们的分页示例中，SqlQuery 使用与 Sql Server 2012 兼容的语法。

通过 Dialect 方法，可以更改 SqlQuery 的目标服务器类型：

```
public SqlQuery Dialect(ISqlDialect dialect)
```

这些是支持的方言类型列表：

```
FirebirdDialect  
PostgresDialect  
SqliteDialect  
SqlServer2000Dialect  
SqlServer2005Dialect  
SqlServer2012Dialect
```

如果我们想在 Sql Server 2005 中查询：

```
void Main()  
{  
    var query = new SqlQuery()  
        .Dialect(SqlServer2005Dialect.Instance)  
        .From("Person")  
        .Select("Firstname")  
        .Select("Lastname")  
        .Select("Count(*)")  
        .OrderBy("PersonId")  
        .Skip(100)  
        .Take(50);  
  
    Console.WriteLine(query.ToString());  
}
```

```
SELECT * FROM (
    SELECT TOP 150
        Firstname,
        Lastname,
        Count(*), ROW_NUMBER() OVER (ORDER BY PersonId) AS __num__
    FROM Person) __results__ WHERE __num__ > 100
```

若使用 `SqliteDialect.Instance`，将输出：

```
SELECT
    Firstname,
    Lastname,
    Count(*)
FROM Person
ORDER BY PersonId LIMIT 50 OFFSET 100
```

如果在应用程序中只使用一种类型的数据库，你可以通过设置默认方言以避免每次开始查询时都要选择方言：

```
SqlSettings.DefaultDialect = SqliteDialect.Instance;
```

在应用程序的启动方法（如 `global.asax.cs`）中添加上述代码。

# Criteria 对象

当为 SELECT、UPDATE 或 DELETE 创建动态 SQL 时，可能需要编写复杂的 WHERE 子句。

也可以使用拼接字符串构建这些语句，但避免语法错误是很繁琐的事且容易遭到 SQL 注入攻击。

使用参数化可以解决 SQL 注入问题，但为了添加参数需要过多的手动工作。

幸运的是，Serenity 有一个条件系统（criteria system），可以帮助你用类似 LINQ 表达式树的方式构建参数化的查询。

Serenity criterias 是通过 C# 的运算符（utilitizing operator）重载特性来实现的，而不像 LINQ 使用表达式树。

让我们首先在 where 子句写一个基本的 SQL 字符串：

```
new SqlQuery()
    .From("MyTable")
    .Select("Name")
    .Where("Month > 5 AND Year < 2015 AND Name LIKE N'%a%'")
```

使用 criteria 对象实现相同的声明：

```
new SqlQuery()
    .From("MyTable")
    .Select("Name")
    .Where(
        new Criteria("Month") > 5 &
        new Criteria("Year") < 4 &
        new Criteria("Name").Contains("a"))
```

这看起来有点长，但它使用了参数：

```

SELECT
    Name
FROM
    MyTable
WHERE
    Month > @p1 AND
    Year < @p2 AND
    Name LIKE N'%a%'

```

如果你有一个实体，你可以在智能提示的帮助下写该语句：

```

var m = MyTableRow.Fields;
new SqlQuery()
    .From(m)
    .Select(m.Name)
    .Where(
        m.Month > 5 &
        m.Year < 4 &
        m.Name.Contains("a"))

```

我们这里没有使用 `new Criteria()`，因为 `Field` 对象已经有构建条件的重载操作。

## BaseCriteria 对象

`BaseCriteria` 是所有条件 (`criteria`) 对象类型的基类。

它重载了几个 C# 操作运算符，包括 `>`、`<`、`&`、`|`，它们可以用在 C# 表达式中，以构建复杂的条件。

`BaseCriteria` 自身没有构造函数，所以你需要创建一个从它派生的对象。`Criteria` 可能是最常使用的一个子类。

## Criteria 对象

`Criteria` 是一个简单对象，包含 SQL 表达式的字符串，通常该字符串是一个字段名称。

```
new Criteria("MyField")
```

它也可以包含一个 SQL 表达式（尽管不建议使用这种方式）：

```
new Criteria("a + b")
```

系统不会检查该参数的语法，因此构建的条件中可能含有无效的表达式。

```
new Criteria("Some invalid expression()///'^')")
```

## AND (&) 操作

可以使用 C# 的 `&` 运算符对两个条件对象进行与 (AND) 运算：

```
new Criteria("Field1 > 5") &
new Criteria("Field2 < 4")
`
```

请注意，我们这里不是使用短路运算符 `&&`。

它使用运算符 (AND) 创建新的条件对象 (BinaryCriteria)，并引用这两个条件对象。它并不修改原始的条件对象。

BinaryCriteria 类似于表达式树的二元表达式 (BinaryExpression)。

它的 SQL 输出将是：

```
Field1 > 5 AND Field2 < 4
```

也可以使用 C# 的 `&=` 运算符：

```
BaseCriteria c = new Criteria("Field1 > 5");
c &= new Criteria("Field2 < 4")
```

BaseCriteria 是所有条件对象类型的基类。如果在第一行使用 `Criteria c = ...`，第二行将得到编译时错误，因为 `&` 运算符返回 BinaryCriteria 对象，而不是返回 Criteria 对象。

## OR (|) 操作

类似于 AND 操作，但它使用 OR。

```
new Criteria("Field1 > 5") |
new Criteria("Field2 < 4")
`
```

```
Field1 > 5 OR Field2 < 4
```

## 括号操作 (~)

当使用多个 AND/OR 子句，你可能想使用括号。

```
new Criteria("Field1 > 5") &
(new Criteria("Field2 > 7") | new Criteria("Field2 < 3"))
```

但是这不可以与条件对象一起工作，因为上述的条件将输出：

```
Field1 > 5 AND Field2 > 7 OR Field2 < 3
```

这里的信息适用于 Serenity 1.9.8 之前的版本。在该版本之后，Serenity 在所有二元条件（AND、OR 等）周围添加括号，即使你没有使用括号。

所以，如果你想在某些地方显式地使用括号，你只能使用 ~。

我们的括号被怎么处理？让我们试着添加更多的括号。

```
new Criteria("Field1 > 5") &
((((new Criteria("Field2 > 7") | new Criteria("Field2 < 3")))))
```

一直还是输出：

```
Field1 > 5 AND Field2 > 7 OR Field2 < 3
```

C# 没有提供重载括号的方法，它只被用来决定运算的顺序，因此，Serenity criteria 不能确定你是否在使用括号。

我们必须使用特殊的运算符： ~ （实际上它是 C# 的补码）。

```
new Criteria("Field1 > 5") &  
~(new Criteria("Field2 > 7") | new Criteria("Field2 < 3"))
```

现在 SQL 看起来像我们之前所希望的：

```
Field1 > 5 AND (Field2 > 7 OR Field2 < 3)
```

由于 Serenity 1.9.8+ 自动向二元条件添加括号，上面的表达式事实上将变为：

```
(Field1 > 5) AND (((Field2 > 7) OR (Field2 < 3)))
```

## 比较运算符 (>, >=, <, <=, ==, !=)

我们重载了大多数 C# 比较运算符，所以你可以在条件中使用它们。

```
new Criteria("Field1") == new Criteria("1") &  
new Criteria("Field2") != new Criteria("2") &  
new Criteria("Field3") > new Criteria("3") &  
new Criteria("Field4") >= new Criteria("4") &  
new Criteria("Field5") < new Criteria("5") &  
new Criteria("Field6") <= new Criteria("6")
```

```
Field1 == 1 AND  
Field2 <> 2 AND  
Field3 > 3 AND  
Field4 >= 4 AND  
Field5 < 5 AND  
Field6 <= 6
```

## 内联值（**Inline Values**）

当比较运算符的一侧是条件对象，而另一侧是整数、字符串、日期、guid 等值，这些值将被转换为条件参数。

```
new Criteria("Field1") == 1 &  
new Criteria("Field2") != "ABC" &  
new Criteria("Field3") > DateTime.Now &  
new Criteria("Field4") >= Guid.NewGuid() &  
new Criteria("Field5") < 5L
```

```
Field1 == @p1 AND  
Field2 <> @p2 AND  
Field3 > @p3 AND  
Field4 >= @p4 AND  
Field5 < @p5
```

当含有该条件的查询发送到 SQL，这些参数将具有相应的值。

自动参数编号默认从 1 开始，但最后一个编号存储在查询中，被条件使用，所以编号可能会变化。

让我们在查询中使用该条件：

```

new SqlQuery()
    .From("MyTable")
    .Select("Field999")
    .Where(new Criteria("FirstOne") >= 999)
    .Where(new Criteria("SecondOne") >= 999)
    .Where(
        new Criteria("Field1") == 1 &
        new Criteria("Field2") != "ABC" &
        new Criteria("Field3") > DateTime.Now &
        new Criteria("Field4") >= Guid.NewGuid() &
        new Criteria("Field5") < 5L
    )
)

```

```

SELECT
    Field999
FROM
    MyTable
WHERE
    FirstOne >= @p1 AND -- @p1 = 999
    SecondOne >= @p2 AND -- @p2 = 999
    Field1 == @p3 AND -- @p3 = 1
    Field2 <> @p4 AND -- @p4 = N'ABC'
    Field3 > @p5 AND -- @p5 = '2016-01-31T01:16:23'
    Field4 >= @p6 AND -- @p6 = '23123-DEFCD-....'
    Field5 < @p7 -- @p7 = 5

```

这里是与之前列出表达式有相同的条件，参数编号从 3 而不是 1 开始。因为 2 之前的编号被用于其他 WHERE 子句。

所以参数编号使用查询作为上下文。你不应该做出参数名称将是什么的假设。

## ParamCriteria 和显式参数名

如果想要使用一些显式命名的参数，你可以使用 ParamCriteria：

```
new SqlQuery()
    .From("SomeTable")
    .Select("SomeField")
    .Where(new Criteria("SomeField") <= new ParamCriteria("@mypa
ram"))
    .Where(new Criteria("SomeOtherField") == new ParamCriteria(""
@myparam"))
    .SetParam("@myparam", 5);
```

我们这里使用 `SqlQuery` 的 `SetParam` 扩展方法设置参数的值。

也可以事先声明此参数，然后重用它：

```
var myParam = new ParamCriteria("@myparam");

new SqlQuery()
    .From("SomeTable")
    .Select("SomeField")
    .Where(new Criteria("SomeField") <= myParam)
    .Where(new Criteria("SomeOtherField") == myParam)
    .SetParam(myParam.Name, 5);
```

## ConstantCriteria

如果你不想使用参数化的查询，可以使用 `ConstantCriteria` 对象存储值。它们将不会转换为自动参数。

```
new SqlQuery()
    .From("MyTable")
    .Select("MyField")
    .Where(
        new Criteria("Field1") == new ConstantCriteria(1) &
        new Criteria("Field2") != new ConstantCriteria("ABC")
    )
```

```

SELECT
    MyField
FROM
    MyTable
WHERE
    FirstOne >= 1
    SecondOne >= N'ABC'

```

## Null 比较

在 SQL 中，使用像 `==`, `!=` 的运算符比较 `NULL` 值，将返回 `NULL`。对于这样的比较，应使用 `IS NULL` 或 `IS NOT NULL`。

Criteria 对象没有重载防止 `null` (或 `object`) 的比较，所以如果你尝试写下面的表达式，可能会得到错误：

```

new Criteria("a") == null; // what is type of null?

int b? = null;
new Criteria("c") == b; // no overload for nullable types

```

这些表达式都可以使用 `IsNull` 和 `Nullable.Value` 方法编写：

```

new Criteria("a").IsNull();
new Criteria("a").IsNotNull();
int? b = 5;
new Criteria("c") == b.Value;

```

如果你迫切希望这样写：`Field = NULL`，可以这样做：

```

new Criteria("Field") == new Criteria("NULL")

```

## LIKE 操作

Criteria 有 *Like*, *NotLike*, *StartsWith*, *EndsWith*, *Contains*, *NotContains* 方法，以帮助使用 LIKE 操作。

```
new Criteria("a").Like("__C%") &
new Criteria("b").NotLike("D%") &
new Criteria("c").StartsWith("S") &
new Criteria("d").EndsWith("X") &
new Criteria("e").Contains("This") &
new Criteria("f").NotContains("That")
```

```
a LIKE @p1 AND -- @p1 = N'__C%'
b NOT LIKE @p2 AND -- @p2 = N'D%'
c LIKE @p3 AND -- @p3 = 'S%'
d LIKE @p4 AND -- @p4 = N'%X'
e LIKE @p5 AND -- @p5 = N'%This%'
f NOT LIKE @p6 -- @p6 = N'%That%'
```

## IN 和 NOT IN 操作

在内联数组中使用 IN 或 NOT IN：

```
new Criteria("A").In(1, 2, 3, 4, 5)
```

```
A IN (@p1, @p2, @p3, @p4, @p5)
-- @p1 = 1, @p2 = 2, @p3 = 3, @p4 = 4, @p5 = 5
```

```
new Criteria("A").NotIn(1, 2, 3, 4, 5)
```

```
A NOT IN (@p1, @p2, @p3, @p4, @p5)
-- @p1 = 1, @p2 = 2, @p3 = 3, @p4 = 4, @p5 = 5
```

也可以向 IN 方法传递任何可枚举的参数：

```
IEnumerable<int> x = new int[] { 1, 3, 5, 7, 9 };
new Criteria("A").In(x);
```

```
A IN (1, 3, 5, 7, 9)
-- @p1 = 1, @p2 = 3, @p3 = 5, @p4 = 7, @p5 = 9
```

也可以使用子查询：

```
var query = new SqlQuery()
    .From("MyTable")
    .Select("MyField");

query.Where("SomeID").In(
    query.SubQuery()
        .From("SomeTable")
        .Select("SomeID")
        .Where(new Criteria("Balance") < 0));
```

```
SELECT
    MyField
FROM
    MyTable
WHERE
    SomeID IN (
        SELECT
            SomeID
        FROM
            SomeTable
        WHERE
            Balance < @p1 -- @p1 = 0
    )
```

## NOT 操作

使用 C# 的 !(not) 运算符表示 NOT 运算操作：

```
!(new Criteria("a") >= 5)
```

```
NOT (a >= @p1) -- @p1 = 5
```

## Field 对象的用法

到目前为止，我们已经使用 Criteria 对象构造函数来构建条件。Field 对象也有类似的重载，所以它们可以相互替代。

例如，以 Northwind 的 Order、Detail 和 Customer 行 (row) 为例：

```
var o = OrderRow.Fields.As("o");
var od = OrderDetailRow.Fields.As("od");
var c = CustomerRow.Fields.As("c");
var query = new SqlQuery()
    .From(o)
    .Select(o.CustomerID);

query.Where(
    o.CustomerCountry == "France" &
    o.ShippingState == 1 &
    o.CustomerID.In(
        query.SubQuery()
            .From(c)
            .Select(c.CustomerID)
            .Where(c.Region == "North")) &
    new Criteria(
        query.SubQuery()
            .From(od)
            .Select(Sql.Sum(od.LineTotal.Expression))
            .Where(od.OrderID == o.OrderID)) >= 1000);
```

它将输出：

```
SELECT
    o.CustomerID AS [CustomerID]
FROM
    Orders o
LEFT JOIN
    Customers o_c ON (o_c.CustomerID = o.CustomerID)
WHERE
    o_c.[Country] = @p2
    AND (CASE WHEN
        o.[ShippedDate] IS NULL THEN 0
        ELSE 1
    END) = @p3
    AND o.CustomerID IN (
        SELECT
            c.CustomerID AS [CustomerID]
        FROM
            Customers c
        WHERE
            c.Region = @p1)
    AND (SELECT
        SUM((od.[UnitPrice] * od.[Quantity] - od.[Discount]))
    FROM
        [Order Details] od
    WHERE
        od.OrderID = o.OrderID) >= @p4
```

# 连接和事务

Serenity 使用简单的 ADO.NET 数据访问对象，像 `SqlConnection`、`DbCommand` 等。

它提供了一些基本的助手（`helpers`）来创建连接、添加参数、执行查询等。

## SqlConnections 类

[命名空间: `Serenity.Data`, 程序集: `Serenity.Data`]

该类包含创建连接的静态函数，并在数据库中以不可见的方式控制它。

### SqlConnections.NewByKey 方法

```
public static IDbConnection NewByKey(string connectionKey)
```

该方法用于获取新的 `IDbConnection` 连接字符串，连接字符串定义在应用程序配置文件（`app.config` 或 `web.config`）。

```
using (var connection = SqlConnections.NewByKey("Default"))
{
    // ...
}
```

尽量在 `using` 块中使用数据库连接。

它读取 `web.config` 中 `key` 为 "Default" 的连接字符串，并使用同样在连接字符串中指定的 `ProviderName` 信息创建一个新的连接字符串。例如，如果 `ProviderName` 是 "System.Data.SqlClient"，则创建一个新的 `SqlConnection` 对象。

通常不需要显式打开连接，因为在需要它们的时候将自动打开连接（只要你使用 Serenity 扩展）。

### SqlConnections.NewFor< TClass > 方法

如果你不想记忆连接字符串的键，而是想重用行 (row) 的信息（表单的 *ConnectionKey* 特性），你可能会更喜欢该变体。

查看 Row 类的顶部，你可能会发现由 Sergen 生成的 ConnectionKey 特性：

```
[ConnectionKey("Northwind")]
public sealed class CustomerRow : Row, IIdRow, INameRow
{}
```

当要查询客户时，为了不使用硬编码 "Northwind"，你可以重用来自 CustomerRow 的信息：

```
using (var connection = SqlConnections.NewFor<CustomerRow>())
{
    return connection.List<CustomerRow>();
```

此方法等效于 *SqlConnections.NewByKey("Northwind")*。

我们没有在这里打开连接，因为 List 扩展方法会自动打开它。

用此方法的类不一定非得是 Row，任何类包含 ConnectionKey 特性都可以工作，即管大部分情况会是 row 类。

## SqlConnections.New 方法

```
public static IDbConnection New(string connectionString, string
providerName)
```

有时，你可能想创建配置文件中不存在的连接。

```
using (var connection = SqlConnections.New(
    "Data Source=(localdb)\v11.0; Initial Catalog=Northwind;
    Integrated Security=true", "System.Data.SqlClient"))
{
    // ...
}
```

在这里我们需要指定连接字符串和提供者的名称（如 "System.Data.SqlClient"）。

你可能会问自己“为什么使用这种方法而不是简单地使用 `new SqlConnection()`？”请参阅下一关于这些优势的主题。

## WrappedConnection

到目前为止，我们看到的所有方法都返回 `IDbConnection` 对象。你可能以为返回的是 `SqlConnection`、`FirebirdConnection` 等，但这是不完全正确的。

你所接收到的 `IDbConnection` 对象是 Serenity 特定的 `WrappedConnection` 对象，实际上包含了基本的 `SqlConnection` 或 `FirebirdConnection` 等。

这样做有助于 Serenity 提供一些功能，如自动打开连接、方言支持、默认事务、单元工作模式、可测试的重载连接等。

你可以不需要注意这些细节，而直接使用返回 `IDbConnection` 实例工作，它们会像基础连接那样工作，但你应该优先使用 `SqlConnections` 方法来创建连接，否则可能会丢失一些列出的功能。

## UnitOfWork 和 IUnitOfWork

`UnitOfWork` 是包含事务引用的简单对象。它有两个我们可以附加的额外事件：

比方说我们正在创建任务，当这些任务成功保存到数据库时，应该发送几封电子邮件。

如果我们很急并在事务提交之前发送电子邮件，可能会在事务失败的情况下为不存在的任务发送电子邮件。所以我们应该只在事务被提交成功时发送电子邮件，例如，在 `OnCommit` 事件发。

你可能会说先提交事务然后再发送电子邮件，但是，如果我们调用的创建任务服务只是更大操作中的一个步骤，所以我们不控制事务而在所有步骤成功后提交。

另一个场景是关于上传文件。这次我们要更新一些文件（File）实体，并且上传的新文件将替换旧文件。如果我们同样很急并且在事务执行完成之前(而最后事务执行失败了)删除旧的文件，我们最终得到实际不存在磁盘中的旧文件的文件实体。所以，我们其实应该在 `OnCommit` 事件中删除文件并替换新的文件，且在 `OnRollback` 事件中删除上传的文件。

```
void SomeBatchOperation()
{
    using (var connection = SqlConnections.NewByKey("Default"))
    using (var uow = new UnitOfWork(connection))
    {
        // here we are in a transaction context
        // create several tasks in transaction
        CreateATask(new TaskRow { ... });
        CreateATask(new TaskRow { ... });
        //...

        // commit the transaction
        // if any exception occurs here or at prior
        // lines transaction will rollback
        // and no e-mails will be sent
        uow.Commit();
    }
}

void CreateATask(IUnitOfWork uow, TaskRow task)
{
    // insert task using connection wrapped inside IUnitOfWork
    // this will automatically run in transaction context
    uow.Connection.Insert(task);

    uow.OnCommit += () => {
        // send e-mail for this task now, this method will only
        // be called if transaction commits successfully
    };

    uow.OnRollback += () => {
        // optional, do something else if it fails
    };
}
```



# 服务终结点

在 Serenity 中，服务终结点是 ASP.NET MVC 控制器的一个子类。

这是 Northwind 的 OrderEndpoint 摘录：

```
namespace Serene.Northwind.Endpoints
{
    [RoutePrefix("Services/Northwind/Order"), Route("{action}")]
    [ConnectionKey("Northwind"), ServiceAuthorize(Northwind.PermissionKeys.General)]
    public class OrderController : ServiceEndpoint
    {
        [HttpPost]
        public SaveResponse Create(IUnitOfWork uow, SaveRequest<MyRow> request)
        {
            return new MyRepository().Create(uow, request);
        }

        public ListResponse<MyRow> List(IDbConnection connection,
            ListRequest request)
        {
            return new MyRepository().List(connection, request);
        }
    }
}
```

## 控制器的命名和命名空间

即使文件名为 *OrderEndpoint.cs*，但我们的类名称却是 *OrderController*。这是由于 ASP.NET MVC 的限制（我找不到之所以这样的逻辑）：所有控制器必须以 *Controller* 后缀结尾。

如果控制器类名称不以此后缀结束，行为 (*action*) 将根本不能工作。所以要非常小心。

这个错误我也犯了几次并花了我个小时。

类的命名空间 (*Serene.Northwind.Endpoints*) 就不是那么重要了，但我们通常把终结点放在 *MyProject.Module.Endpoints* 命名空间下，以保持一致性。

*OrderController* 继承自 *ServiceEndpoint*（也应该），从而为该 MVC 控制器提供不太常用的功能，我们很快就会介绍这些功能。

## 路由特性

```
[RoutePrefix("Services/Northwind/Order"), Route("{action}")]
```

上面的路由特性属于 ASP.NET 路由特性，为服务终结点配置基本地址。我们的行为将使用 "mysite.com/Services/Northwind/Order" 获得。

请避免经典 ASP.NET MVC 路由，它把所有路由放在 *ApplicationStart* 方法用 *routes.AddRoute* 等配置。这样真的很难管理。

*Serenity* 的所有服务终结点默认使用 */Services/Module/Entity* 寻址方案。即使你依然能够使用另一套寻址方案，但建议保持一致性并遵守基本约定。

## ConnectionKey 特性

该特性指定在创建连接时，应该使用应用程序配置文件 (如 *web.config*) 中的那个连接键。

让我们看看如何以及何时使用它来自动创建连接：

```
public ListResponse<MyRow> List(IDbConnection connection, ListRequest request)
{
    return new MyRepository().List(connection, request);
}
```

在这里我们看到该操作需要一个 *IDbConnection* 参数，但不能从客户端发送 *IDbConnection* 到 MVC 的行为 (action)。所以由谁来创建该连接？

还记得我们的控制器是继承 `ServiceEndpoint` 吗？因此 `ServiceEndpoint` 可以知道我们的行为需要连接，它会检查控制器类的 `[ConnectionKey]` 特性来确定连接键，然后使用 `SqlConnections.NewByKey()` 创建一个连接，并用此连接执行我们的行为，当行为执行结束时，关闭连接。

可以从行为中删除此连接参数并手动创建它：

```
public ListResponse<MyRow> List(ListRequest request)
{
    using (var connection = SqlConnections.NewByKey("Northwind"))
    {
        return new MyRepository().List(connection, request);
    }
}
```

实际上 `ServiceEndpoint` 在幕后为我们创建连接。

为什么不使用此功能，而让平台自动处理这个细节呢？其中一个原因是：你可能需要打开一个未列在配置文件中的自定义连接，或根据某些条件打开一个动态连接。

我们有另一种方法，它需要 `IUnitOfWork`（事务）而不是 `IDbConnection` 参数：

```
public SaveResponse Create(IUnitOfWork uow, SaveRequest<MyRow> request)
{
    return new MyRepository().Create(uow, request);
}
```

这里的情况是类似的。`ServiceEndpoint` 创建另一个连接，但这次它(`IUnitOfWork`)在该连接启动一个事务来调用我们的行为方法，并在返回时自动提交事务。如果执行失败，将回滚。

这是同一件事的手动版：

```

public SaveResponse Create(SaveRequest<MyRow> request)
{
    using (var connection = SqlConnections.NewByKey("Northwind"))
    {
        using (var uow = new UnitOfWork(connection))
        {
            var result = new MyRepository().Create(uow, request);
            uow.Commit();
            return result;
        }
    }
}

```

因此，ServiceEndpoint 用 1 行代码处理 8 行代码的逻辑。

## 何时使用 **IUnitOfWork / IDbConnection**

按照惯例，修改一些状态（创建、更新等）的 Serenity 操作方法应该在内部使用事务，因此需要使用 **IUnitOfWork** 参数，而那些只读操作（列表、检索）应该使用 **IDbConnection**。

如果服务方法含 **IUnitOfWork** 参数，则表明你的方法将修改一些数据。

## 关于 **[HttpPost]** 特性

你可能已经注意到创建(Create)，更新(Update)，删除(Delete)等方法具有此特性，而列表(List)、检索(Retrieve)等方法不包含该特性。

此特性限制创建、更新、删除操作只能使用 HTTP POST，而不允许它们由 HTTP GET 调用。

因为这些方法修改状态，例如从 DB 插入、更新、删除一些记录，所以它们不应该被无意中调用，并且它们的结果不应该被允许缓存。

这也带来一些安全隐患。GET 方法的行为可能会受到一些攻击。

列表、检索不会修改任何状态，因此它们允许使用 GET 调用，如：在浏览器的地址栏调用。

即使列表、检索可以通过 GET 调用，Serenity 总是使用 HTTP POST 调用服务（如 Q.CallService），并启用缓存，以避免出现意外的结果。

## ServiceAuthorize 特性

我们的控制器类有 ServiceAuthorize 特性：

```
ServiceAuthorize(Northwind.PermissionKeys.General)
```

该特性类似于 ASP.NET MVC 的 [Authorize] 特性，但它只检查用户是否已登录，若没有登录则抛出异常。

如果使用时不带参数（如 [ServiceAuthorize()]），此属性也会检查该用户是否已登录。

当你把访问许可键(permission key) 字符串传递给它时，它会检查该用户是否已登录且具有该权限。

```
ServiceAuthorize("SomePermission")
```

如果用户未被授予 "SomePermission"，则阻止他执行任何终结点的方法。

[PageAuthorize] 特性也类似，但你可能更喜欢在服务终结点使用 [ServiceAuthorize] 特性，因为它的错误处理更适合服务。

虽然 [PageAuthorize] 将用户重定向到登录页面，但如果用户没有权限，ServiceAuthorize 返回一个更适合的未授权的服务错误。

也可以在行为中使用 [ServiceAuthorize] 特性，而不只在控制器中使用该特性：

```
[ServiceAuthorize("SomePermissionThatIsRequiredForCreate")]
public SaveResponse Create(SaveRequest<MyRow> request)
```

## 关于 Request 和 Response 对象

除了特殊处理的 IUnitOfWork 和 IDbConnection 参数，所有 Serenity 服务行为都是单个请求参数并返回单个结果。

```
public SaveResponse Create(IUnitOfWork uow, SaveRequest<MyRow> request)
```

让我们从返回结果开始。如果你有使用 ASP.NET MVC 的背景，你会知道控制器不能返回任意对象。它们必须返回一个派生自 *ActionResult* 的对象。

但是我们的 *SaveResponse* 派生自 *ServiceResponse*，它只是一个普通的对象：

```
public class SaveResponse : ServiceResponse
{
    public object EntityId;
}

public class ServiceResponse
{
    public ServiceError Error { get; set; }
}
```

这怎么可能？还是 *ServiceEndpoint* 在幕后处理这些细节。它把 *SaveResponse* 转换为指定行为返回 JSON 数据的结果。

只要响应对象是从 *ServiceResponse* 派生并且是可序列化成 JSON，我们就不必担心这些细节。

我们的请求对象也是一个普通的类，派生自基本的 *ServiceRequest* 类：

```
public class SaveRequest<TEntity> : ServiceRequest, ISaveRequest
{
    public object EntityId { get; set; }
    public TEntity Entity { get; set; }
}

public class ServiceRequest
{}
```

*ServiceEndpoint* 的 HTTP 请求内容通常是 JSON，若要将其反序列化为我们的请求参数，需要使用特殊的 MVC 操作过滤器(JsonFilter)。

如果想使用一些自定义的操作，你的方法也应该遵循这一理念，如：只有一个请求（派生自 `ServiceRequest`）并返回一个响应（派生自 `ServiceResponse`）。

让我们添加一个服务方法，让其返回所有订单总数大于某一数量的订单：

```
public class MyOrderCountRequest : ServiceRequest
{
    public decimal MinAmount { get; set; }
}

public class MyOrderCountResponse : ServiceResponse
{
    public int Count { get; set; }
}

public class OrderController : ServiceEndpoint
{
    public MyOrderCountResponse MyOrderCount(IDbConnection connection,
                                              MyOrderCountRequest request)
    {
        var fld = OrderRow.Fields;
        return new MyOrderCountResponse
        {
            Count = connection.Count<OrderRow>(fld.TotalAmount >
= request.MinAmount);
        };
    }
}
```

请遵循这种模式并不要尝试向操作方法添加更多的参数。`Serenity` 遵循只有一个请求对象的基本消息模式，以便后继的扩展可添加更多的属性。

不要这样做（被称为 RPC 风格。RPC, Remote procedure call：远程过程调用）：

```
public class OrderController : ServiceEndpoint
{
    public decimal MyOrderCount(IDbConnection connection,
        decimal minAmount, decimal maxAmount, ....)
    {
        // ...
    }
}
```

更佳的做法是（基于消息服务）：

```
public class MyOrderCountRequest : ServiceRequest
{
    public decimal MinAmount { get; set; }
    public decimal MaxAmount { get; set; }
}

public class OrderController : ServiceEndpoint
{
    public MyOrderCountResponse MyOrderCount(IDbConnection connection,
        MyOrderCountRequest request)
    {
        // ...
    }
}
```

这可以避免记忆参数的顺序，使你的请求对象更具扩展性而又不破坏向后的兼容性，还有更多你可能后来才会注意到的优点。

## 为什么 **Endpoint** 方法几乎都是空的？

我们通常将实际工作委托给仓储层：

```
public ListResponse<MyRow> List( IDbConnection connection, ListRequest request)
{
    return new MyRepository().List(connection, request);
}
```

记住，ServiceEndpoint 直接依赖于 ASP.NET MVC。这意味着你在 ServiceEndpoint 写的任何代码将依赖于 ASP.NET MVC，因此需要 web 环境。

你可能不能重用任何写在这里的代码，比如重用一个桌面应用程序的代码。否则就不能将此代码独立为一个 DLL，它并不具有对 WEB 库引用的任何代码。

但如果你真的没有这样的需求，你可以删除所有的仓储及所有在终结点内部编写的代码。

有些人可能认为：实体、仓储、业务规则、终结点等都应该在自己独立的程序集中。从理论上及某些情况下，这可能是有效的，但有些（或大部分）的用户不需要这么多的程序集，且可能落入 YAGNI（YAGNI, you aren't gonna need it : 你不会需要它）类别。

# 列表请求处理器 (ListRequestHandler)

这是处理来自客户端列表请求的基类，如，从网格列表的请求。

让我们首先介绍该类是在何时及如何处理列表请求的：

1. 首先必须从客户端触发列表请求。可能的情形有：
  - a) 打开含网格的列表页面。在创建网格对象之后，基于当前的可见列、初始排序、过滤器等建立了一个 `ListRequest` 对象，并将其提交到服务器端。
  - b) 用户点击列头排序、点击分页按钮或者刷新按钮时触发与情形 A 同样的事件。
  - c) 手动使用 `XYZService.List` 方法调用列表服务。
2. MVC `XYZController` (在 `XYZEndpoint.cs` 文件中) 的服务请求 (AJAX) 到达服务器，请求参数从 JSON 反序列化为 `ListRequest` 对象。
3. `XYZEndpoint` 调用 `XYZRepository.List` 方法，并以检索到的 `ListRequest` 对象作为其参数。
4. `XYZRepository.List` 方法创建一个 `ListRequestHandler` (`XYZRepository.MyListHandler`) 的子类并使用 `ListRequest` 作为参数调用其 `Process` 方法。
5. `ListRequestHandler.Process` 方法根据 `ListRequest`、实体类型 (Row) 的元数据及其他信息构建动态 SQL 查询语句，并执行它。
6. `ListRequestHandler.Process` 返回 `ListResponse`，它包含要返回行的 Entities 成员。
7. `XYZEndpoint` 接收该 `ListResponse`，并从 action 中返回它。
8. `ListResponse` 被序列化成 JSON 发送回客户端。
9. Grid 接收实体，更新其显示的行及其他像分页状态的部件。

我们将在另一章中介绍如何生成网格和提交列表请求。现在，让我们集中于 `ListRequestHandler`。

## 列表请求对象

我们应该先看看 `ListRequest` 对象有哪些成员：

```

public class ListRequest : ServiceRequest, IIncludeExcludeColu
mns
{
    public int Skip { get; set; }
    public int Take { get; set; }
    public SortBy[] Sort { get; set; }
    public string ContainsText { get; set; }
    public string ContainsField { get; set; }
    public Dictionary<string, object> EqualityFilter { get; se
t; }

    [JsonConverter(typeof(JsonSafeCriteriaConverter))]
    public BaseCriteria Criteria { get; set; }
    public bool IncludeDeleted { get; set; }
    public bool ExcludeTotalCount { get; set; }
    public ColumnSelection ColumnSelection { get; set; }

    [JsonConverter(typeof(JsonStringHashSetConverter))]
    public HashSet<string> IncludeColumns { get; set; }
    [JsonConverter(typeof(JsonStringHashSetConverter))]
    public HashSet<string> ExcludeColumns { get; set; }
}

```

## ListRequest.Skip 和 ListRequest.Take 参数

这些参数用于分页，它们类似于 LINQ 中的 Skip 和 Page 扩展。

这里有一个需要指出的小区别。如果你使用 Take(0)，LINQ 无记录返回，而 Serenity 将返回所有的记录。调用 LIST 服务并请求 0 条记录是毫无意义的。

所以，SKIP 和 TAKE 的默认值为 0，并且它们将忽略 0 / undefined。

```

// returns all customers as Skip and Take are 0 by default
CustomerService.List(new ListRequest
{
}, response => {});

```

如果你有页面大小为 50 的网格列表，切换到第 4 页，将跳过前 200 条记录，并选取 50 条记录。

```
// returns customers between row numbers 201 and 250 (in some default order)
CustomerService.List(new ListRequest
{
    Skip = 200,
    Take = 50
}, response => {});
```

这些参数根据 SQL 方言转换为有关的 SQL 分页语句。

## ListRequest.Sort 参数

此参数接受数组并返回排序后的结果。排序是由生成的 SQL 执行。

SortBy 参数希望接收一个 SortBy 对象的列表：

```
[JsonConverter(typeof(JsonSortByConverter))]
public class SortBy
{
    public SortBy()
    {

    }

    public SortBy(string field)
    {
        Field = field;
    }

    public SortBy(string field, bool descending)
    {
        Field = field;
        Descending = descending;
    }

    public string Field { get; set; }
    public bool Descending { get; set; }
}
```

当需要调用服务端 `XYZRepository` 的 `List` 方法先对国家排序，然后按城市倒序，你可能会这样做：

```
new CustomerRepository().List(connection, new ListRequest
{
    SortBy = new[] {
        new SortBy("Country"),
        new SortBy("City", descending: true)
    }
});
```

`SortBy` 类有一个自定义的 `JsonConverter`，因此当在客户端构建一个列表请求，你应该使用一个简单的字符串数组：

```
// CustomerEndpoint and thus CustomerRepository is accessed from
// client side (YourProject.Script) through CustomerService class
// static methods
// which is generated by ServiceContracts.tt
CustomerService.List(connection, new ListRequest
{
    SortBy = new[] { "Country", "City DESC" }
}, response => {});
```

这是由于 `ListRequest` 类定义在客户端，具有略微不同的结构：

```
[Imported, Serializable, PreserveMemberCase]
public class ListRequest : ServiceRequest
{
    public int Skip { get; set; }
    public int Take { get; set; }
    public string[] Sort { get; set; }
    // ...
}
```

这里使用的列名称应与字段的属性名称对应。不允许使用表达式。下面的做法是不可行的！

```
CustomerService.List(connection, new ListRequest
{
    SortBy = new[] { "t0.FirstName + ' ' + t0.LastName" }
}, response => {});
```

## ListRequest.ContainsText 和 ListRequest.ContainsField 参数

这是网格左上角搜索输入框的快速搜索功能所使用的参数。

当只指定 ContainsText 而 ContainsField 为空时，对所有含 [QuickSearch] 特性的字段执行搜索。

可以定义一些特定的字段列表，以便通过重写 GetQuickSearchField() 方法对客户端网格执行搜索。所以当在快速搜索输入框中选择这些字段，则只执行对所选列的搜索。

如果将 ContainsField 设置为没有快速搜索特性的字段名称，出于安全目的，系统将引发异常。

像往常一样，搜索使用动态 SQL 的 LIKE 语句完成。

```
CustomerService.List(connection, new ListRequest
{
    ContainsText = "the",
    ContainsField = "CompanyName"
}, response => {});
```

```
SELECT ... FROM Customers t0 WHERE t0.CompanyName LIKE '%the%'
```

如果 ContainsText 为 null 或空字符串，将忽略该值。

## ListRequest.EqualityFilter 参数

EqualityFilter 是一个字典，允许按某些字段进行快速相等筛选，用于网格上面的下拉列表快速过滤器（用 AddEqualityFilter 帮助类定义）。

```
CustomerService.List(connection, new ListRequest
{
    EqualityFilter = new JsDictionary<string, object> {
        { "Country", "Germany" }
    }
}, response => {});
```

```
SELECT * FROM Customers t0 WHERE t0.Country = "Germany"
```

再次提醒：你应该使用属性名称作为相等字段键（equality field keys），而不能使用表达式。Serenity 不允许客户端有任何随心所欲的 SQL 表达式，以防止 SQL 注入。

请注意，类似于 ContainsText，它将忽略 null 值和空字符串值，因此不能在 EqualityFilter 使用空值或 null 值进行筛选，这样的请求将返回的所有记录：

```
CustomerService.List(connection, new ListRequest
{
    EqualityFilter = new JsDictionary<string, object> {
        { "Country", "" }, // won't work, empty string is ignored

        { "City", null }, // won't work, null is ignored
    }
}, response => {});
```

如果你试图用空的国家条件筛选客户，请使用的 Criteria 参数。

## ListRequest.Criteria

此参数接受条件对象，类似于我们在流式 SQL 章节谈到的服务端 Criteria 对象。唯一不同的是，由于这些条件对象是发送自客户端，因此必须验证其不能包含任何随心所欲的 SQL 表达式。

下面的服务请求将返回国家和城市都为空的客户：

```
CustomerService.List(connection, new ListRequest
{
    Criteria = new Criteria("Country") == "" ||
        new Criteria("City").IsNull()
}, response => {});
```

你可以设置生成 ListRequest 的 Criteria 参数，它将在 XYZGrid.cs 像下面这样提交：

```
protected override bool OnViewSubmit()
{
    // only continue if base class didn't cancel request
    if (!base.OnViewSubmit())
        return false;

    // view object is the data source for grid (SlickRemoteView)
    // this is an EntityGrid so view.Params is a ListRequest
    var request = (ListRequest)view.Params;

    // we use " &= " here because otherwise we might clear
    // filter set by an edit filter dialog if any.

    request.Criteria &=
        new Criteria(ProductRow.Fields.UnitsInStock) > 10 &
        new Criteria(ProductRow.Fields.CategoryName) != "Condiments" &
        new Criteria(ProductRow.Fields.Discontinued) == 0;

    return true;
}
```

还可以在网格上类似地设置 ListRequest 的其他参数。

## ListRequest.IncludeDeleted

此参数只对实现了 `IIsActiveDeletedRow` 接口的行才有用。如果有这样接口的行，列表处理器默认只返回没有被删除的行 (`IsActive != -1`)。这些行并没有被实际删除，而是被标记为已删除。

如果此参数为 `True`，列表处理器将不检索 `IsActive` 列而返回所有的行。

一些网格对这样的行在右上角有一个小橡皮擦图标来切换该标识，从而可显示或隐藏已删除的记录（默认）。

## ListRequest.ColumnSelection 参数

Serenity 力图只从 SQL 服务器为实体加载必需的列，以使 SQL Server <-> WEB 服务器之间保持尽可能低的网络通信量。

`ListRequest` 有一个 `ColumnSelection` 参数使你可以控制从 SQL 加载的列集合。

`ColumnSelection` 枚举有如下的值定义：

```
public enum ColumnSelection
{
    List = 0,
    KeyOnly = 1,
    Details = 2,
}
```

默认情况下，网格列表从 "ColumnSelection.List" 模式（可以更改）的列表服务中请求记录。因此，其列表请求看起来像这样：

```
new ListRequest
{
    ColumnSelection = ColumnSelection.List
}
```

在 `ColumnSelection.List` 模式中，`ListRequestHandler` 返回表字段，因此，这些字段是实际属于该表的字段，而不是来自关联表的视图字段。

有一个例外：表达式字段只包含表字段的引用，如 `(t0.FirstName + ' ' + t0.LastName)`。`ListRequestHandler` 同样加载这些字段。

*ColumnSelection.KeyOnly* 只包含 ID / 主键 字段。

*ColumnSelection.Details* 包含所有字段，包括视图字段，除非该字段被显式排除或被标记为 "sensitive" (如，密码字段)。

对话框在 Details 模式加载编辑记录，因此它们也包含视图字段。

## ListRequest.IncludeColumns 参数

我们告诉网格在 *List* 模式下请求记录，因此加载只表字段，那么它如何显示来自其它表的列呢？

网格将可见列的列表发送到列表服务的 *IncludeColumns*，所以即使它们是视图字段，也在选择 (*selection*) 中包含这些列。

在内存网格 (memory grids) 中不能这样做。因为它们不会直接调用服务，你必须在视图字段添加 [MinSelectLevel(SelectLevel.List)] 特性，这样才能在内存详细网格 (memory detail grids) 加载。

如果你有显示供应商名称 (*SupplierName*) 的产品网格列表，它实际的 ListRequest 看起来像这样：

```
new ListRequest
{
    ColumnSelection = ColumnSelection.List,
    IncludeColumns = new List<string> {
        "ProductID",
        "ProductName",
        "SupplierName",
        "..."
    }
}
```

因此，这些额外的视图字段也包含在选择 (*selection*)。

如果你有一个网格列表，出于性能考虑你应该只能加载可见的列，且它的 ColumnSelection 级别重写为 KeyOnly。请注意，非可见表字段不会出现在客户端行 (row) 中。

## ListRequest.ExcludeColumns 参数

IncludeColumns 的相反功能是 ExcludeColumns。比方说在网格列表的行中有一个永远也不会显示的类型为 nvarchar(max) 的 Notes 字段。为了降低网络流量，可以选择不在产品网格中加载此字段：

```
new ListRequest
{
    ColumnSelection = ColumnSelection.List,
    IncludeColumns = new List<string> {
        "ProductID",
        "ProductName",
        "SupplierName",
        "...",
    },
    ExcludeColumns = new List<string> {
        "Notes"
    }
}
```

OnViewSubmit 是设置此参数（及一些其他参数）的最佳场所：

```
protected override bool OnViewSubmit()
{
    if (!base.OnViewSubmit())
        return false;

    var request = (ListRequest)view.Params;
    request.ExcludeColumns = new List<string> { "Notes" }
    return true;
}
```

## 在服务端控制加载

你可能想要从 *ColumnSelection.List* 排除一些像 Notes 这样的字段，而不是显式地在网格中排除它们。使用 MinSelectLevel 特性可以实现此目的：

```
[MinSelectLevel(SelectLevel.Details)]
public String Note
{
    get { return Fields.Note[this]; }
    set { Fields.Note[this] = value; }
}
```

这是加载字段时控制不同 ColumnSelection 级别的 SelectLevel 枚举：

```
public enum SelectLevel
{
    Default = 0,
    Always = 1,
    Lookup = 2,
    List = 3,
    Details = 4,
    Explicit = 5,
    Never = 6
}
```

*SelectLevel.Default*：默认值，对应于表字段是 *SelectLevel.List*，视图字段是 *SelectLevel.Details*。

默认情况下，表字段的选择级别是 *SelectLevel.List*，而视图字段是 *SelectLevel.Details*。

*SelectLevel.Always*：表示此字段可被任何列选择模式选择，包括使用 *ExcludeColumns* 显式排除的字段。

*SelectLevel.Lookup* 已经被废弃，请避免使用。检索列由 *[LookupInclude]* 特性决定。

*SelectLevel.List*：表示在 *ColumnSelection.List* 和 *ColumnSelection.Details* 模式或被 *IncludeColumns* 参数显式包含时选择此字段。

*SelectLevel.Details*：表示在 *ColumnSelection.Details* 模式或被 *IncludeColumns* 参数显式包含时选择此字段。

*SelectLevel.Explicit*：表示此字段不应该在任何模式下被选择，除非它显式包含在 `IncludeColumns` 参数。在网格或编辑对话框使用此字段，是没有意义的。

*SelectLevel.Never*：表示永远不会加载此字段！用于不应该发送到客户端的字段（如，密码哈希值）。

# 部件(Widgets)

Serenity 脚本 UI 层的组件类（控制）基于类似于 *jQuery UI* 的部件工厂系统，但使用 C# 重新设计。

你可以在这里找到有关 *jQuery UI* 部件系统的更多详细信息：

<http://learn.jquery.com/jquery-ui/widget-factory/>

<http://msdn.microsoft.com/en-us/library/hh404085.aspx>

部件(Widget)，是附加到 HTML 元素并扩展其行为的对象。

例如，当 IntegerEditor 部件附加到 INPUT 元素时，IntegerEditor 部件让该元素输入数字更加容易并验证输入的数字是否是正确的整数。

类似地，当一个工具栏部件附加到 DIV 元素时，该 DIV 元素便变成了一个带有工具按钮的工具栏（在本例中，DIV 仅作为占位符）。

# ScriptContext 类

C# 不支持全局方法，因此 jQuery 的 `$` 函数在 Saltarelle 中不能像在 Javascript 那样使用。

在 Javascript 中的表达式 `$('#SomeElementId')` 对应于 Saltarelle 的 C# 代码 `jQuery.Select("#SomeElementId")`。

一种替代方法，可以使用 `ScriptContext`：

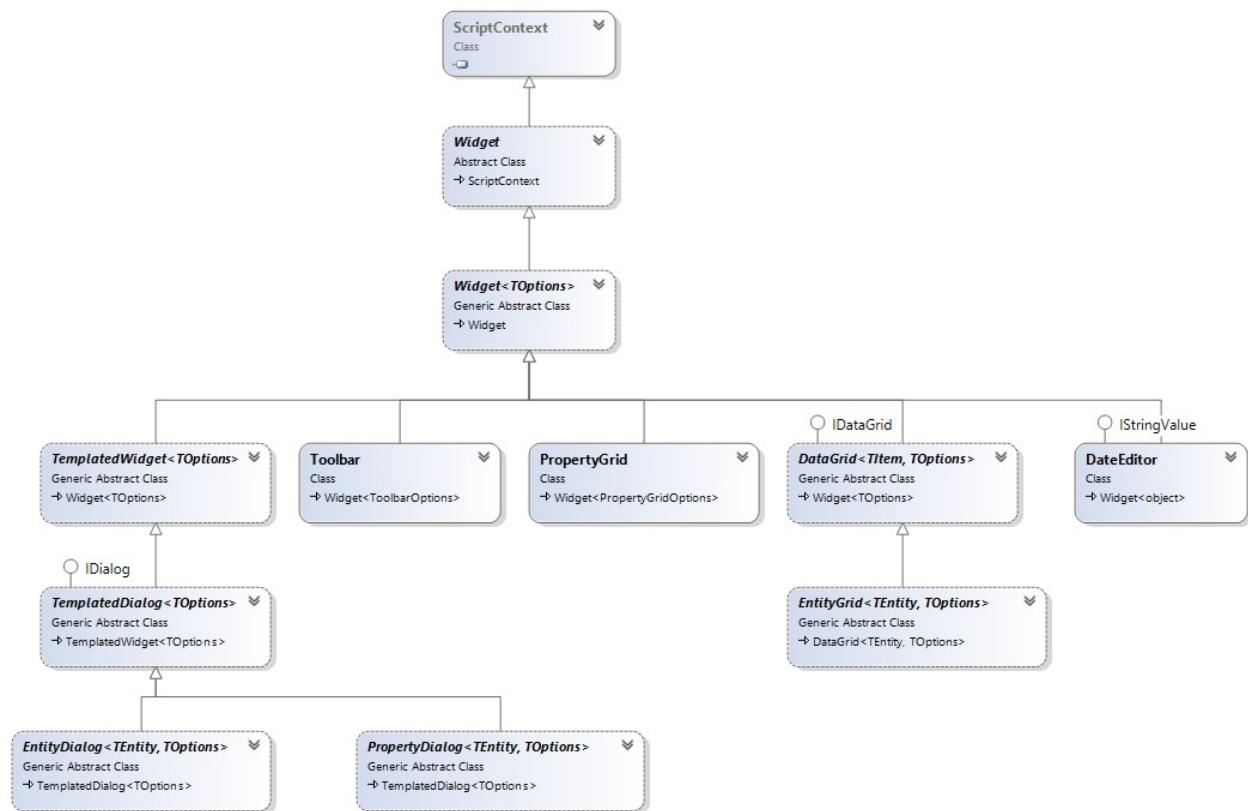
```
public class ScriptContext
{
    [InlineCode("${{p}}")]
    protected static jQueryObject J(object p);
    [InlineCode("${{p}}, {{context}}")]
    protected static jQueryObject J(object p, object context);
}
```

由于 `$` 在 C# 中不是有效的方法名称，可以选择 `J` 代替。在 `ScriptContext` 子类中，`jQuery.Select()` 函数可以简单地使用 `J()` 调用。

```
public class SampleClass : ScriptContext
{
    public void SomeMethod()
    {
        J("#SomeElementId").AddClass("abc");
    }
}
```

# Widget 类

## 部件的类图



## 一个简单的部件示例

让我们来构建一个小部件，让 DIV 在每次单击后增加其字体的大小：

```
namespace MySamples
{
    public class MyCoolWidget : Widget
    {
        private int fontSize = 10;

        public MyCoolWidget(jQueryObject div)
            : base(div)
        {
            div.Click(e => {
                fontSize++;
                this.Element.Css("font-size", fontSize + "pt");
            });
        }
    }
}
```

```
<div id="SomeDiv">Sample Text</div>
```

我们可以在 HTML 元素上创建该小部件，如：

```
var div = jQuery.Select("#SomeDiv");
new MyCoolWidget(div);
```

## Widget 类的成员

```

public abstract class Widget : ScriptContext
{
    private static int NextWidgetNumber = 0;

    protected Widget(jQueryObject element);
    public virtual void Destroy();

    protected virtual void OnInit();
    protected virtual void AddCssClass();

    public jQueryObject Element { get; }
    public string WidgetName { get; }
    public string UniqueName { get; }
}

```

## Widget.Element 属性

继承自 Widget 的类，可以通过 Element 属性得到它们创建的元素。

```
public jQueryObject Element { get; }
```

此属性的类型为 jQueryObject，并返回该元素，以在创建部件时使用。在我们的示例中。在点击事件中使用 this.Element 引用容器 DIV 元素。

## HTML 元素和部件的 CSS 类

当在 HTML 元素上创建部件时，它对元素进行一些修改。

首先，根据部件的类型，HTML 元素得到一个 CSS 类。

在我们示例中，.s-MyCoolWidget 样式类被添加到 ID 为 #SomeDiv 的 DIV。

因此，在创建部件之后，DIV 看起来类似于：

```
<div id="SomeDiv" class="s-MyCoolWidget">Sample Text</div>
```

此 CSS 类是由部件类名前面添加 `s-` 前缀组成（可以通过重写 `Widget.AddCssClass` 方法改变该前缀）。

## 使用部件 **CSS** 类对 **HTML** 元素进行样式化

部件 CSS 类可以用于部件创建的 HTML 元素。

```
.s-MyCoolWidget {
    background-color: red;
}
```

## 使用 **jQuery.Data** 函数从 **HTML** 元素获得部件

随着添加 CSS 类，部件的另一个信息也被添加到 HTML 元素，即使它不是明显的标签。可以通过在 Chrome 控制台输入如下代码查看此信息：

```
> $('#SomeDiv').data()
> Object { MySamples_MyCoolWidget: $MySamples_MyCoolWidget }
```

因此，可以使用 `$.data` 函数获得一个附加到 HTML 元素的部件。在 C# 中，可以写成：

```
var myWidget = (MyCoolWidget)(J("#SomeDiv").GetDataValue('MySamples_MyCoolWidget'));
```

## WidgetExtensions.GetWidget 扩展方法

上面那行代码看起来有点长且复杂，可以用 Serenity 的快捷方式替代：

```
var myWidget = J("#SomeDiv").GetWidget<MyCoolWidget>();
```

如果 HTML 元素存在部件，上述代码则返回部件，否则将抛出异常：

```
Element has no widget of type 'MySamples_MyCoolWidget'!
```

## WidgetExtensions.TryGetWidget 扩展方法

TryGetWidget 可以用于检查是否存在部件，如果不存在，则返回 null：

```
var myWidget = $('#SomeDiv').TryGetWidget<MyCoolWidget>();
```

## 在 HTML 元素创建多个部件

在同一类中只可以附加一个部件到 HTML 元素。

试图在一个元素上创建同一类(class) 的辅助部件，将引发以下错误：

```
The element already has widget 'MySamples_MyCoolWidget'.
```

只要部件的行为不相互影响，可以向单个元素附加任意数量来自不同类(class) 的部件。

## Widget.UniqueName 属性

每个部件实例自动获得一个唯一的名称（如 MySamples\_MyCoolWidget3），可以通过 this.UniqueName 属性访问该名称。

把这个唯一名称作为 HTML 元素和其它由部件自身产生的子元素的 ID 前缀非常有用。

它也可以用在事件类 \$.bind 和 '\$.unbind'，附加 / 移除事件不会影响其它可能附加到元素的事件：

```
jQuery("body").Bind("click." + this.UniqueName, delegate { ... });
...
jQuery("body").Unbind("click." + this.UniqueName);
```

## Widget.Destroy 方法

有时可能需要释放附加的部件，而不是删除 HTML 元素自身。

Widget 类为该目的提供了 `Destroy` 方法。

在 `Destroy` 方法的默认实现中，由部件自身指定事件（通过使用 `UniqueName` 事件类）进行清理并从 HTML 元素中移除 CSS 类 (`.s-WidgetClass`)。

自定义的部件类可能需要重写 `Destroy` 方法，以撤消更改 HTML 元素和释放资源（虽然不需要移除以前使用 `UniqueName` 类附加的事件）。

当从 DOM 中删除含部件的 HTML 元素时，会自动调用 `Destroy` 方法。也可以手动调用该方法。

如果销毁(`destory`) 操作不能被正确执行，在某些浏览器中就可能会出现内存泄漏。

# Widget < TOptions > 泛型类

如果部件需要一些额外的初始化选项，可以从 `Widget< TOptions >` 类派生。

在类方法中可以通过受保护的字段 `options` 访问传递给构造函数的选项。

```
public abstract class Widget< TOptions > : Widget
    where TOptions: class, new()
{
    protected Widget(jQueryObject element, TOptions opt = null)
    { ... }
    protected readonly TOptions options;
}
```

# TemplatedWidget 类

部件在其构造函数或其它方法生成复杂的 HTML 标签，因此可能导致类具有很多难以维护的意大利面条式的代码。此外，由于标签就定在程序代码中，可能难以自定义输出。

```
public class MyComplexWidget : Widget
{
    public MyComplexWidget(jQueryObject div)
        : base(div)
    {
        var toolbar = J("<div>")
            .Attribute("id", this.UniqueName + "_MyToolbar")
            .AppendTo(div);

        var table = J("<table>")
            .AddClass("myTable")
            .Attribute("id", this.UniqueName + "_MyTable")
            .AppendTo(div);

        var header = J("<thead/>").AppendTo(table);
        var body = J("<tbody/>").AppendTo(table);
        ...
        ...
        ...
    }
}
```

通过使用 HTML 模板，可以避免这种问题。例如，我们可以向 HTML 页面添加下列模板：

```

<script id="Template_MyComplexWidget" type="text/html">
<div id="~_MyToolbar">
</div>
<table id="~_MyTable">
    <thead><tr><th>Name</th><th>Surname</th>...</tr></thead>
    <tbody>...</tbody>
</table>
</script>

```

在这里使用了 `SCRIPT` 标签，但通过指定其类型为 `"text/html"`，浏览器不会把它当成真正的脚本来执行。

让我们通过使用 `TemplatedWidget` 重写之前的意大利面条式的代码块：

```

public class MyComplexWidget : TemplatedWidget
{
    public MyComplexWidget(jQueryObject div)
        : base(div)
    {
    }
}

```

当在一个 `HTML` 元素上创建该小部件，其内容如下：

```

<div id="SampleElement">
</div>

```

你最终将得到这样的 `HTML` 标签：

```

<div id="SampleElement">
    <div id="MySamples_MyComplexWidget1_MyToolbar">
    </div>
    <table id="MySamples_MyComplexWidget1_MyTable">
        <thead><tr><th>Name</th><th>Surname</th>...</tr></thead>
        <tbody>...</tbody>
    </table>
</div>

```

TemplatedWidget 自动查找类的模板，并将它应用于 HTML 元素。

## 生成 TemplatdWidget 的 ID

如果你仔细观察，在模板中我们为子元素指定 ID：`~_MyToolbar` 和 `~_MyTable`。

但是当该模板应用到 HTML 元素时，结果标签使用的 ID 却是 `MySamples_MyComplexWidget1_MyToolbar` 和

`MySamples_MyComplexWidget1_MyTable`。

TemplatedWidget 使用部件的 唯一名称 和下划线("") (`this.idPrefix` 包含合并的前缀) 替换 `~` 前缀。

使用该策略，即使同一部件模板在页面中用于多个 HTML 元素，其 ID 也不会互相冲突，因为它们都有唯一的 ID。

## TemplatedWidget.ByID 方法

TemplateWidget 向生成的标签追加一个唯一的名称，部件模板中的 ID 属性不能用于访问创建部件后的元素。

部件的唯一名称和下划线应该放在查找元素的模板中的原始 ID 属性前面：

```
public class MyComplexWidget : TemplatedWidget
{
    public MyComplexWidget(jQueryObject div)
        : base(div)
    {
        J(this.uniqueName + "_" + "Toolbar").AddClass("some-class");
    }
}
```

也可以使用 TemplatdWidget 的 ByID 方法替代：

```

public class MyComplexWidget
{
    public MyComplexWidget(jQueryObject div)
        : base(div)
    {
        ByID("Toolbar").AddClass("some-class");
    }
}

```

## TemplatedWidget.GetTemplateName 方法

在最近的示例中， MyComplexWidget 自动查找其模板。

TemplatedWidget 根据约定找到其模板（根据编码约定）。它在类名称之前插入 Template\_ 前缀，搜索含此 ID 属性 ( Template\_MyComplexWidget ) 的 SCRIPT 元素，并使用其 HTML 内容作为模板。

如果我们想使用另一个像下面的 ID :

```

<script id="TheMyComplexWidgetTemplate" type="text/html">
    ...
</script>

```

在浏览器控制台中将看到下面的错误：

```

Can't locate template for widget 'MyComplexWidget' with name 'Template_MyComplexWidget'!

```

我们可以修改模板 ID 或要求部件使用自定义 ID :

```
public class MyComplexWidget
{
    protected override string GetTemplateName()
    {
        return "TheMyComplexWidgetTemplate";
    }
}
```

## TemplatedWidget.GetTemplate 方法

`GetTemplate` 方法可以被重写，以从另一个资源提供或手动指定模板：

```
public class MyComplexWidget
{
    protected override string GetTemplate()
    {
        return $('#TheMyComplexWidgetTemplate').GetHtml();
    }
}
```

## Q.GetTemplate 方法和服务端模板

默认调用 `GetTemplateName` 方法并搜索具有该 ID 的 `SCRIPT` 元素来实现 `TemplatedWidget.GetTemplate` 方法。

如果没有找到这样的 `SCRIPT` 元素，可使用同一 ID 调用 `Q.GetTemplate` 方法获取模板。

如果也不返回结果，则会抛出错误。

`Q.GetTemplate` 可访问定义在服务端的模板。这些模板从 `~/Views/Template`、`~/Modules` 或它们的子文件夹中含 `.template.cshtml` 扩展的文件编译而来。

例如，我们在服务端为 `MyComplexWidget` 创建一个模板文件 `~/Views/Template/SomeFolder/MyComplexWidget.template.cshtml`，其内容如下：

```
<div id="~_MyToolbar">
</div>
<table id="~_MyTable">
    <thead><tr><th>Name</th><th>Surname</th>...</tr></thead>
    <tbody>...</tbody>
</table>
```

模板文件名和扩展名是很重要的，而其文件夹会被完全忽略。

使用该策略就不需要将部件模板插入到页面标签中。

此外，由于这种服务端的模板在首次使用时加载（延迟加载），并在浏览器和服务端中缓存，页面标签没有被可能永远不会在特定网页使用的部件模板污染。因此，服务端模板青睐内嵌 **SCRIPT** 模板。

## TemplatedDialog 类

TemplatedWidget 的子类 TemplatdDialog 使用 jQuery UI 对话框创建页面内的模态对话框。

与其他小部件类型不同，TemplatedDialog 创建自己的 HTML 元素，并把自身附加到该新建的元素。

## 网格列表（**Grids**）

# 格式化器类型

## URLFormatter

此格式化器让你可以在网格列显示 URL 链接。

它有如下可选参数：

选项名称	描述
UrlFormat	<p>URL 的格式。如，在 "<a href="http://www.site.com/{0}">http://www.site.com/{0}</a>" 中 {0} 被替换为 UrlProperty 的值。</p> <p>如果没有指定，连接将依然显示 UrlProperty 的值。</p> <p>如果 URL 格式以 "~/" 开头，它将解析为应用程序根。例如，如果格式为 "~/upload/{0}"，且你的应用程序运行在 "localhost:3045/mysite"，生成的 URL 为 "/mysite/upload/xyz.png"。</p>
UrlProperty	<p>该属性用于确定链接的 URL。</p> <p>如果没有指定，列的名称被该格式化器替换。</p> <p>如果 UrlProperty 的值以 "~/" 开头，将像 UrlFormat 一样解析。</p>
DisplayFormat	<p>链接的文本显示格式。例如，在 "单击以打开 {0}" 中 {0} 替换为 DisplayProperty 的值。</p> <p>如果未指定格式，链接的值将为 DisplayProperty。</p>
DisplayProperty	<p>该属性用于确定连接文本。</p> <p>如果没有指定，列的名称被该格式化器替换。</p>
Target	链接的目标。使用 "_blank" 表示在新选项卡中打开链接。

# 持久化设置

Serenity 2.1.5 引入保存如下信息的网格列表设置：

- 可见列和显示顺序
- 列宽
- 排序的列
- 高级过滤器（由右下角的编辑过滤器链接创建）
- 快速过滤器（撰写本文档时，尚未提供该功能）
- 包含已删除的状态切换

默认情况下，网格列表不会自动持久化任何东西。

因此，如果你隐藏某些列并离开订单页面，当你再次返回该页面时，你就会看到那些隐藏的列再次成为可见列。

你需要开启所有网格列表的持久化设置，或设置单独记住它们的设置。

## 默认情况下开启持久性

DataGridView 有名为 `DefaultPersistanceStorage` 的静态配置参数。此参数控制默认情况下的网格列表设置自动保存的位置。它最初为空。

在 `ScriptInitialization.ts`，你可能像下面这样默认开启所有网格列表的持久化：

```
namespace Serene.ScriptInitialization {
    Q.Config.responsiveDialogs = true;
    Q.Config.rootNamespaces.push('Serene');

    Serenity.DataGrid.defaultPersistanceStorage = window.session
    Storage;
}
```

这将设置保存到浏览器的会话(session)。当任何浏览器窗口保持打开状态时，将数据保存到一个键/值字典。当用户关闭所有浏览器窗口时，所有的设置都会丢失。

另一种选择是使用浏览器本地存储。在浏览器重新启动之间将保留设置。

```
Serenity.DataGrid.defaultPersistanceStorage = window.localStorage;
```

使用这两个选项的任意一个，在重新加载页面之间，网格列表会开始记住它们的设置。

## 处理由多个用户共同使用的浏览器

`SessionStorage` 和 `localStorage` 是浏览器范围，因此，如果浏览器由多个用户共同使用，他们会得到同一组设置。

如果一个用户更改某些设置并注销，而后另一个用户登录，第二个用户将以第一个用户的设置开始使用系统（除非你在注销时清除 `localStorage`）。

如果你认为这是应用程序的一个问题，可以尝试编写自定义提供程序：

```
namespace Serene {
    export class UserLocalStorage implements Serenity.SettingsStorage {
        getItem(key: string): string {
            return window.localStorage.getItem(
                Authorization.userDefinition.Username + ":" + key);
        }

        setItem(key: string, value: string): void {
            window.localStorage.setItem(
                Authorization.userDefinition.Username + ":" + key,
                value);
        }
    }

    //...
    Serenity.DataGrid.defaultPersistanceStorage = new UserLocalStorage();
}
```

请注意，上述代码不能提供任何的安全保障。它只是让用户有单独的设置。

## 持久化存储每个网格列表类型的设置

要开启持久化，或更改某一特定网格列表的存储目标，可重写 `getPersistanceStorage` 方法：

```
namespace Serene.Northwind {
    //...
    export class OrderGrid extends Serenity.EntityGrid<OrderRow,
any> {
        //...

        protected getPersistenceStorage(): Serenity.SettingsStorage {
            return window.localStorage;
        }
    }
}
```

你也可以在该方法中通过返回 `null` 关闭网格列表类的持久化。

## 确定保存的设置类型

默认情况下，在开始时注意到的所有设置（如可见列、宽度、过滤器等）都被保存。你也可以选择不持久化/还原特定的设置。这些都可在 `getPersistenceFlags` 方法控制：

```
namespace Serene.Northwind {
    //...
    export class OrderGrid extends Serenity.EntityGrid<OrderRow,
any> {
        //...

        protected getPersistenceFlags(): GridPersistenceFlags {
            return {
                columnWidths: false // dont persist column width
            };
        }
    }
}
```

这里是一组完整的标识：

```
interface GridPersistenceFlags {
    columnWidths?: boolean;
    columnVisibility?: boolean;
    sortColumns?: boolean;
    filterItems?: boolean;
    quickFilters?: boolean;
    includeDeleted?: boolean;
}
```

## 何时保存/还原设置

当你改变网格列表的如下设置时，会自动保存设置：

- 使用列选取器对话框选择可见的列
- 手动调整列的大小
- 编辑高级过滤器
- 拖动列、改变列位置
- 更改排序的列

只在创建网格列表后，在加载第一页时还原设置。

## 持久化设置到数据库（**UserPreferences** 表）

Serene 2.1.5 带有 **UserPreferences** 表，可以作为持久化存储设置。若要使用该表存储设置，你只需要将网格列表的 **defaultPersistanceStorage** 设置为存储类型（类似于其他存储类型）。

```
/// <reference path="../Common/UserPreference/UserPreferenceStorage.ts" />

Serenity.DataGrid.defaultPersistanceStorage = new Common.UserPreferenceStorage();
```

别忘了添加引用语句，否则你会有运行时错误，因为 TypeScript 有加载顺序问题。

或者：

```
namespace Serene.Northwind {
    //...
    export class OrderGrid extends Serenity.EntityGrid<OrderRow,
any> {
    //...

    protected getPersistenceStorage(): Serenity.SettingsStorage {
        return new Common.UserPreferenceStorage();
    }
}
```

## 手动保存/还原设置

如果需要手动保存/还原设置，可以使用以下方法：

```
protected GetCurrentSettings(GridPersistanceFlags? flags): PersistedGridSettings;
protected RestoreSettings(PersistedGridSettings settings, GridPersistanceFlags? flags): void;
```

这些都是 DataGrid 的受保护方法，所以只能从子类调用。

# 代码生成器 (Sergen)

Sergen 有一些额外的配置选项，你可以通过解决方案目录下的配置文件 (Serenity.CodeGenerator.config) 设置。

这是所有的配置选项：

```
public class GeneratorConfig
{
    public List<Connection> Connections { get; set; }
    public string KDiff3Path { get; set; }
    public string TFPPath { get; set; }
    public string TSCPPath { get; set; }
    public bool TFSIntegration { get; set; }
    public string WebProjectFile { get; set; }
    public string ScriptProjectFile { get; set; }
    public string RootNamespace { get; set; }
    public List<BaseRowClass> BaseRowClasses { get; set; }
    public List<string> RemoveForeignFields { get; set; }
    public bool GenerateSSImports { get; set; }
    public bool GenerateSSTypings { get; set; }
    public bool GenerateTSCode { get; set; }
}
```

Connections、RootNamespace、WebProjectFile、ScriptProjectFile、GenerateSSImports、GenerateSSTypings 和 GenerateTSCode 都可在用户界面中设置，所以我们将重点放在其他配置选项。

## KDiff3 路径

Sergen 在有需要时会尝试启动 KDiff3 把更改合并到现有文件。这也会发生在当再次尝试为实体生成代码时，Sergen 将执行 KDiff3，而不是重写目标文件。

Sergen 在其默认路径 C:\Program Files\Kdiff3 查找 KDiff3，但是如果你把 Kdiff3 安装在其它位置，可以使用该配置选项重写这个路径。

## TFSIntegration 和 TFPath

对于使用 TFS 的用户，Sergen 提供此配置选项让其可以检出现有文件及添加新项目到源代码控制。如果你的项目是在 TFS，则把 TFSIntegration 设置为 true；如果 tf.exe 不在其默认位置 C:\Program Files\Visual Studio\x.y\Common7\ide\，则设置 TFPath。

```
{  
    // ...  
    "TFSIntegration": true,  
    "TFPath": "C:\Program Files\....\tf.exe"  
}
```

## RemoveForeignFields

默认情况下，Sergen 会检查表的外键，并生成一个含所有外键表字段的行类。

有时，你可能有一些外键表字段，例如一些像 InsertUserId、UpdateDate 等对另一行没有用的日志字段。

你也可以在生成代码后手动删除它们，但使用此配置选项可能更容易，因为只需在字符串数组中列出想要在生成行中删除的字段即可：

```
{  
    // ...  
    "RemoveForeignFields": ["InsertUserId", "UpdateUserId",  
                           "InsertDate", "UpdateDate"]  
}
```

请注意，这并不会从表的行类删除这些字段，它只从外键联接中删除这些视图字段。

## BaseRowClasses

如果你使用一些基行类（base row class），例如，类似 Serene 中的 LoggingRow 类。你可能想让 Sergen 生成继承自这些基行类（base row class）的类。

若要实现此目标，需要列出基类和其字段。

```
{  
    // ...  
    "BaseRowClasses": [{  
        "ClassName": "Serene.Administration.LoggingRow",  
        "Fields": ["InsertUserId", "UpdateUserId",  
                  "InsertDate", "UpdateDate"]  
    }]  
}
```

如果 Sergen 确定表已经有数组 "Fields" 列出的所有字段，它将把 "ClassName" 设置为基类，而并不会在行中显式生成这些字段，因为它们已经在基行类（base row class）中定义。

也可以定义多个基行类（base row class）。如果行字段匹配多个基类，Sergen 将选择匹配字段最多的基行类。

# 使用的工具和库

Serenity 平台使用了一些很有用的开源工具和库，列出如下（按字母顺序排列）：

此列表可能看起来有点长，但一个 Serenity 的应用程序并没有依赖所有的库。

其中一些库只在 Serenity 平台自身的发展过程中才用到，而有一些是可选功能的依赖项。

我们尽量使用开源库，因为它们的优质可以避免重新造轮子。

## **Autonumeric**

(<https://github.com/BobKnothe/autoNumeric>)

**BlockUI** (<https://github.com/malsup/blockui/>)

**Bootstrap** (<https://github.com/twbs/bootstrap>)

**Cake Build** (<https://github.com/cake-build/cake>)

**Cecil** (<https://github.com/jbevain/cecil>)

## **Clean-CSS [Node]**

(<https://github.com/jakubpawlowicz/clean-css>)

**Colorbox** (<https://github.com/jackmoore/colorbox>)

## **Dapper**

(<https://github.com/StackExchange/dapper-dot-net>)

**DialogExtend** (<https://github.com/ROMB/jquery-dialogextend>)

**jLayout** (<https://github.com/bramstein/jlayout>)

**Json.NET**

(<https://github.com/JamesNK/Newtonsoft.Json>)

**JSON2**

(<https://github.com/douglascrockford/JSON-js>)

**JSRender**

(<https://github.com/BorisMoore/jsrender>)

**jQuery** (<https://github.com/jquery/jquery>)

**jQuery Cookie** (<https://github.com/carhartl/jquery-cookie>)

**jQuery Validation**

(<https://github.com/jzaefferer/jquery-validation>)

**jQuery UI** (<https://github.com/jquery/jquery-ui>)

**jQuery.event.drag**

(<http://threehubmedia.com/code/event/drag>)

**Less.JS (Node)** (<https://github.com/less/less.js>)

**Linq.js** (<http://linqjs.codeplex.com/>)

- metisMenu**  
(<https://github.com/onokumus/metisMenu>)
- Munq** (<https://munq.codeplex.com/>)
- NodeJS** (<https://github.com/joyent/node>)
- Pace** (<https://github.com/HubSpot/pace>)
- PhantomJS** (<https://github.com/ariya/phantomjs>)
- RazorGenerator**  
(<https://razorgenerator.codeplex.com/>)
- RSVP** (<https://github.com/tildeio/rsvp.js/>)
- Saltarelle Compiler** (<https://github.com/erik-kallen/SaltarelleCompiler>)
- Select2** (<https://github.com/ivaynberg/select2>)
- SlickGrid** (<https://github.com/mleibman/SlickGrid>)
- Toastr** (<https://github.com/CodeSeven/toastr>)
- UglifyJS2 (Node)**  
(<https://github.com/mishoo/UglifyJS2>)
- XUnit** (<https://github.com/xunit/xunit>)

