Moqui 应用开发指南

轻松地开发完整的企业级应用

（美）David E. Jones 著

EricChang（张振宇） 译

Version 1.0 - First Edition

Based on Moqui Framework version 1.4.1 and Mantle Business Artifacts version 0.5.2.
These open source projects are public domain licensed and are available for download through http://www.moqui.org.

A special thanks to the sponsors who helped make this book what it is, keep the price low on the printed book, and make the PDF download version available for free.

For permission to use any part of this work, please send an email to the author at dej@dejc.com. For more details about the author see his web site at http://www.dejc.com.

Designed for a full color 8x10" bound book. With this format the PDF version can also be printed on A4 or US Letter paper.

For the CreateSpace print edition:
ISBN-13: 978-0692267059"
ISBN-10: 0692267050"

前言

我不是一名专家级的架构开发者，而是和你们大家一样，只是一名职业的应用程序开发人员（*老戴好谦虚哈 @_@！*）。我的职业生涯一直围绕着构建和定制化各种不同领域组织的应用系统，并帮助他们进行生产流程管理以及信息自动化管理。

像任何一个工匠一样，一套很好的工具对于应用程序开发人员来说是十分必要的。而我自从 1999 年涉足商业领域开始，就在寻觅这套最好的开发工具集。但是在当时，企业级 Java 应用还未到成熟期，市场上也存在很多不同的工具和技术实现在不断验证和巩固一系列的规范和标准。

这当中存在一个问题：构建一个大规模的 ERP 系统，需要大量的硬件资源支撑才可以运行良好，同时，大厂商的这种企业级应用服务都绑定各自不完备的规范实现，即使有配套的工具和实现技术，对于开发来说也是很难使用的，开发人员备受折磨。并且这些应用在初始化完成之后，客制化和维护也十分困难和昂贵。总之一句话，蛋疼。

各种参照大部分标准的开源替代项目逐渐在市场上浮现出来，并与商业级的产品继续角逐。然而，这只能有助于减少授权的费用，但是对于开发和产品性能而言，帮助并不是很有效。

当然，这里还有很大的改善空间。在 2001 年，我发起了一个开源项目 OFBIZ（The Open For Business Project），它实现了一组自动化信息系统基础的各种行为操作。这意味着其中电子商务、ERP、CRM、MRP（*制造资源计划：Manufacturing Resources Planning*）等都是梳理过并可用的系统模块。基于我在企业级 Java 应用的经验和独特的想法、设计模式，我设计了一套十分特别的工具集能让人们进行开发工作。这套工具集简化了组织数据的对象映射、逻辑封装以及系统内部使用支持面向服务的设计模式，面向服务逐步变成系统间内部交互的一种规范。

不仅仅是技术开发工具，一个好的应用开发人员还需要一个灵活可扩展、便于理解的数据模型能够定义应用开发中的数据结构和保持数据的一致性。幸运的是，在 2001 年我早期规划阶段，正好大概开始 OFBiz 前两个月，Len Silverston 出版了《数据模型资源手册，卷一》和《数据模型资源手册，卷二》。这是 Silverston 对于之前一本同名书（Silverston, Inmon, and Graziano ， 1997 出版）的高度回顾和总结阐述。

数据模型的理念和模式在上面这两卷书中有介绍，这也是 OFBiz 的数据模型的规范标准。这些数据模型作为规范实现，在从一个简单的电子商务应用到全功能特性的 ERP 和 CRM 系统中有很优雅的表现。并且，作为很多开源项目的基础以及扩展上，数据模型在千万个用户系统中被广泛使用。

OFBiz 项目多年来在各种领域中的广泛使用，框架已经扩展积累了很多高度抽象的业务构件。在保持框架稳定的前提下，改进的想法，一些扩展机制还有来源于外部竞争对手的比较等都现实存在。很多的想法都被需要包含在 OFBiz 中，但是随着开源项目的发展特别是社区用户和贡献者的分解，OFBiz 逐步变的基础部分很难去改变。

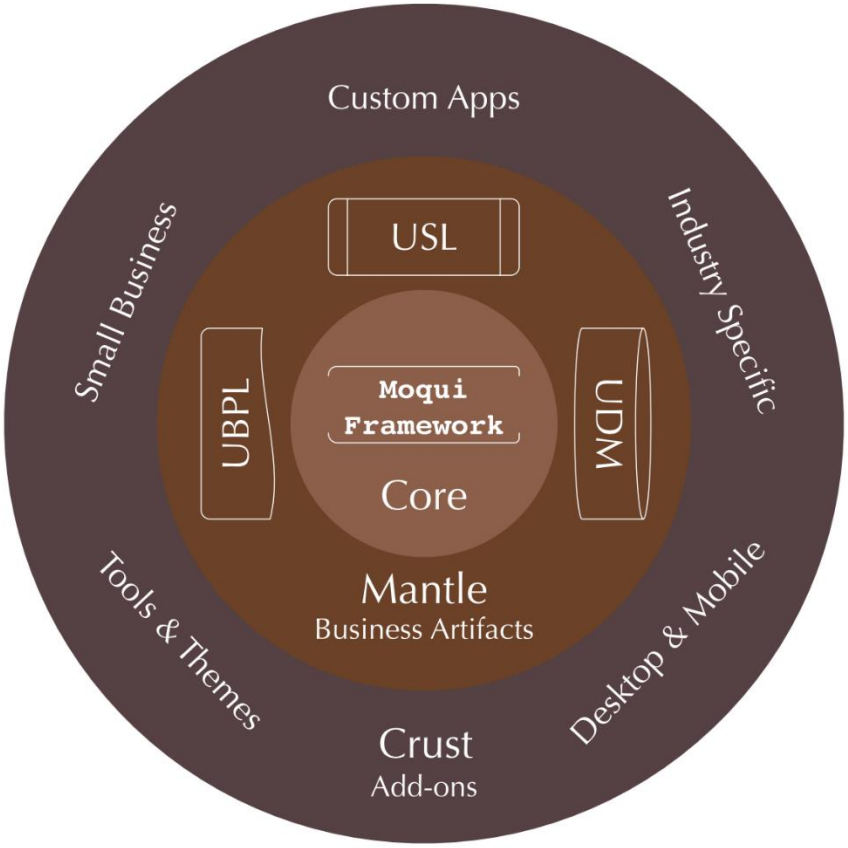
多年来，我对于框架本身有一堆很伟大的并且很关键的改进和扩展列表。随着这些列表逐渐增多，我意识到也许我需要寻求另外一种方式和新的最好工具集去开展下一个阶段，以便解决这些问题。最终结果导致了 Moqui 这个独立框架的诞生。同时，“地幔”业务构件能提供一种通用的开源、内部交互的应用生态系统的基础组成，它支持商业化的产品和开源一般性的业务构件彼此隔离。

本书将帮助您开始使用 Moqui 框架，并提供您花费数以月计开发卓越的应用系统的一个参考。

一、 Moqui 介绍

什么是 Moqui 生态系统？

Moqui 生态系统是一组以一个共同的框架和一系列通用的业务构件为中心的软件包。核心的软件包（在下图中的 Core 内核和 Mantle 地幔中）被拆分为不同的开源项目工程，这种方式可以保证维持其既定目标各自发展，并且能够关注于这些工程的管理和依赖处理，同时保持工程的清洁性。这些软件包按照适中的社区化管理方式，就像 Linux 的 Kernel 内核一样。



这个生态系统的目标是为了能提供一组内部可交互并且能够与商业软件竞争的构件（在 Crust 外壳 中提供的插件机制）。这些构件和机制都基于一个共同的框架并具有很高的客制化的灵活性和简易性，同时，系统提供的一系列通用业务构件（数据模型和逻辑服务）使得其更加完整和完备。

这个生态系统包含：

- Moqui **框架**：简易高效、安全灵活的开发支持
- **地幔业务构件**：一组通用普适的、可作为您各种场景业务系统的基础业务构件集合，包含：
  - **UBPL：通用业务过程/流程库**
  - **UDM ：通用的数据模型**
  - **USL ：通用业务服务库**

- **地壳（插件）**：主题皮肤、综合的工具集、不同行业的应用支持、大公司规模支持、业务领域支持等

本书的关注核心是 Moqui 框架，但是最后一章将对地幔业务构件进行总结。

什么是 Moqui 开发框架？

Moqui 开发框架是一个全功能的，企业级应用开发框架，基于 Groovy 和 Java 语言。这个开发框架包含了一系列的工具用于开发界面、服务、实体以及诸如声明式的构件安全、多租户支持等这种高级功能特性。

这个框架十分适合于开发各种简单的 web 应用站点（如 moqui.org）以及小规模的复合 ERP 系统。构建于 Moqui 框架基础上的应用系统十分容易部署在各种高扩展性的基础软件上（例如 Java Servlet 容器或者应用服务器），并支持传统的关系型数据库以及很多现代的非关系型 NoSQL 数据库上。

Moqui 开发框架基于 OFBiz（目前 Apache 的顶级项目 OFBiz，参见 http://ofbiz.apache.org）十多年来的项目实施经验以及原作者的设计和开发。很多的设想和方法论，包括纯粹的关系数据层（并非传统的对象关系映射）以及面向服务的逻辑层，这些主干核心设计 Moqui 都继承自 OFBiz，并且被重新精炼和组织定义。

由于采用了更干净的设计、更简洁的实现，同时使用了更多 2001 年做 OFBiz 时未使用的很多更好的第三方类库，Moqui 框架的核心代码只有 OFBiz 核心框架代码的 15%左右，并提供了更多有意义的功能和更多的高级工具。

最终，Moqui 框架会在构建系统时，自动的帮您把控住一大部分很重要的核心关键部分的实现。

Moqui 的基本概念

应用构件/工件

Moqui 开发框架的工具集都是围绕着构件进行组织的，同时这些构件允许你进行创建来体现应用系统的通用部分。在 Moqui 框架中，构件指的是你作为开发人员创建的各种 XML 文件甚至是脚本或者其他代码。框架中支持如下的几种类型的构件：

- **实体 entities** ：贯穿于整个业务系统中的关系数据模型（直接使用模型，无需复杂的对象关系映射）
- **界面 screens 和 表单 forms** ：用于基于 web 的应用界面或者其他用户接口（通用方式是基本构件描述存放在 XML 文件中，或者用户指定扩展存放于数据库中）
- **界面转换 screen transitions** ：用于配置页面到页面的流转以及设置页面跳转时，业务处理过程的必要输入
- **服务 services** ：远程调用的方式运行内部逻辑交互或者曝露外部的服务
- **ECA（事件-条件-行为 event-condition-action）规则** ：用于类似实体、服务操作以及 email 信息接收等系统级事件触发

下面给出一个表格来展示应用中通用部分以及构件之间的联系：

界面 screen	XML 定义的界面（渲染成各种类型的文本或者用于生成其他 UI 界面；支持 html、xml、xsl-fo、csv 以及简单文本转换，开箱即用）
表单 form	XML 定义的表单（在界面内定义；各种开箱即用的组件 widgets，并可以自定义组件或者自定义扩展已有组件）
准备展示的数据	界面动作 actions（在界面内定义，可以调用外部逻辑）
页面流	带条件定义和默认响应的界面转换（在源界面内定义转换，响应目标配置为目标界面或者外部页面资源）
处理过程输入	转换动作 actions（不仅可以支持一个简单的服务处理表单的公共验证等，也可以定义页面内部的动作或者外部逻辑调用）
菜单	按照子界面的层级配置定义遍历，自动生成，也可以明确配置定义菜单
内部服务	XML 的服务配置方式来定义各种内部或者外部服务操作的实现
XML 和 JSON 的 RPC 服务	内部服务配置 allow-remote=true 参数，即可以通过公共的服务接口被调用，接口使用的是规范的 List 和 Map 数据接口进行映射
RESTful web 服务	通过简单的转换定义来获取内部服务，支持 URL 路径、表单、JSON 请求以及 JSON 或者 XML 响应
远程服务调用	代理方式来定义内部服务，可以采用自动的 XML-RPC、JSON-RPC 或者其他的映射方式，也可以使用支持 RESTful 的简单工具或者其他服务类型
发送邮件	在 EmailTemplate 中单独配置下标题，界面能直接渲染成 html、简单文本的邮件格式
接收邮件	定义一个 email 的 ECA 规则用于调用一个内部服务去处理 email
使用脚本、模板和 JCR 内容	通过资源的门面接口去访问、执行或者渲染资源

执行上下文

执行上下文是 Moqui API 的应用核心接口部分。运行时会独立的创建一个上下文实例去执行边缘的构件，如界面或者服务。执行上下文，或者简写为“ec”，拥有各种门面的接口去曝露各种框架的功能和工具。

上下文同时也维护着一个上下文的 map 用来呈现每个构件运行的可变空间。这个上下文 map 是一个 map 的堆栈，每个构件执行在一个干净的 map 中，并入栈，一旦构件结束运行即出栈。每当读取这个 map 栈都将从头开始向下遍历直到找到符合条件的 map 条目。当写这个 map 栈时，总是在栈的顶部去执行写操作（除非明确的指定参照根 map，例如在栈底的 map）。

通过这种方式，每个构件无需担心其他构件的影响，但是构件仍然具有很容易的去访问父构件的数据（通过调用构件链的方式调用或者包含方式，能够向下找到当前的构件）。由于上下文是为了每个边缘构件的执行去创建的，所有它拥有构件运行的详实信息，如：什么时候启用，包含的用户，构件携带的信息等等。

构件栈

每个构件在运行时或者包含、调用其他构件时，这个构件都会即时的被压入栈中。这个栈始终维持着当前活动的构件的轨迹，同时构件的执行信息也会被添加到一个构件使用情况的追踪历史列表中。

当构件入栈时，每个构件都需要进行鉴权，同时构件关联的安全信息也会被追溯记录。通过这种方式，权限设计就变得很简单了，包含的构件、被调用的构件都能够继承相应的权限。这种权限继承的机制带来的好处是你只需配置并控制直接能访问的关键界面或者服务就可以了*（注：这意味着内部调用的其他界面或者逻辑服务无需关心鉴权，会直接继承调用者的权限）*。

管中窥豹，可见一斑

在使用 Moqui 框架进行开发时，你将会经常用到高度抽象的构件，如 XML 定义文件。这种设计方式是为了一方面支持很多通用的需求，同时另一方面也是为了能灵活的在任何点上都可以向下落到低层级的工具上，如模板、脚本等。在某些点上，你也许希望了解到框架内部到底在做什么，或者当你遇到了一个问题时，如果你明确知道框架内部的处理机制你能更容易定位处理问题。

然而服务和实体定义通过代码进行控制；其他的构件比如 XML 动作以及 XML 界面和表单是通过 FreeMarker 模板的宏定义去转换处理成其他文本方式去渲染的。XML 动作（Actions）定义会被转化为简单的 Groovy 脚本，然后被容器编译成 class 文件，并进行缓存和执行。XML 界面和表单中显示的部分（widget 组件）同样会通过模板被转化为特定的输出类型（html，

xml，xsl-fo,csv,text 等等）。

通过这种形式，你能够很容易的看到通过模板生成的输出结果，并且通过简单的配置，你甚至可以制定你自己修改的模板或者扩展开箱即用的功能点。

开发过程

Moqui 开发框架为了促进开发实现，采用了一些自然的概念，映射源自于设计阶段的元素，比如：界面概览和线框，页面流图，数据声明以及自动的业务过程描述。这里的每个设计阶段的构件都能够被转化为 Moqui 工具集中特定的实现构件。

这种设计的构件通常是最好的，特别是在基于明确定义和组织特定的活动，例如必须支持干系人和系统之间互相影响之类的需求上。这些需求应当是清晰的，并且能按照设计进行分解以便辅助驱动设计的决策，同时，必须确保系统中所有关键的方面都被认真考虑到并能覆盖设计。

通过这种方式，实现构件能够引用设计的内容，顺延的，设计能够引用需求的内容。伴随着自然的映射设计时的构件，就能很简单的去开发实现构件并能同时分配和验证它们。

按照这种架构设计去实现构件的话，通常首先要求创建构件的这个顺序就无关紧要了。不同的人员甚至可以并行的去工作，比如定义实体和创作界面。

对于基于 web 的应用系统来说，特别是那些需要个性化艺术设计的公开应用，静态构件类似如图片或者 CSS 样式等都需要独立于界面 XML，以单独的文件进行存储。同时，这些静态构件需要按照子界面的目录结构规范去组织文件结构，并且最外层目录需要与界面名相同。在子界面的层级下，不同界面的资源很自然的可以共享。

真实按照界面和表单 XML 生成的 HTML 页面能够被个性化定制，只需要对应每个 XML 元素的生成输出去重写或者添加 FreeMarker 的宏定义即可。当有需要时，个性化的 HTML 页面也是可以被包含到界面中的。框架允许这种半可视化的方式定制生成带样式和脚本的 HTML 页面，或者需要时直接通过自定义的 HTML 页面去满足需求。

Web 前端设计师能够查看真实的通过独立的 CSS 或者其他静态资源自动生成的 HTML 页面进行前端开发工作。当需要个性化定制时，前端开发人员可以先制作出 HTML 原型页面，然后开发人员可以通过制作模板或者定制参数匹配宏定义将定制部分融入到系统中。

另外一种比较好的做法是，雇佣较多的高级前端工程师去完成整个系统的客制化前端 HTML、样式及脚本等，前端使用某个格式的 JSON 数据通过 HTTP 协议与后端的服务接口进行交互*（注：这也是较流行的单页面或者 RESTful 风格的架构）*。这种方式也比较适合于手机或者桌面终端应用通过 web services 与服务器端进行交互。Web services 你可以使用框架内置的 JSON-RPC、XML-RPC 或者其他的服务自动映射工具，也可以使用自定义的包装服务去调用内部的服务以支持各种 web 服务应用架构。

不管怎样，对于一个给定的项目来说，你的团队应该是有分工的，各管一块。框架允许你将一个项目按照设计分割成多个构件，这样就很容易的去分配工作了，大家也可以很方便的在定义好的接口基础上并行进行开发工作。

开发工具

在需求和设计阶段，你需要一组很好用的工具来方便用户、领域专家、需求分析人员、设计人员以及开发人员进行沟通交流。这些工具应当具有以下特性：

- 层次化的文档结构
- 文档间或者文档内有链接可以关联（通常是链接显示的是被关联的文档标题）
- 文档需要支持图片等附件形式
- 每个文档需要有完整的修订版本历史记录
- 每个文档需要有线框的评论内容
- 文档更新能够自动邮件通知
- 统一库管理，能够在线访问，便于交流

现实中有很多这类的工具可供选择，但是大部分的都不满足上述所有的要求，所以沟通上因此会产生诸多不便。一个很好的商业化的选择是 Atlassian 公司的 Confluence。相对很多面向大规模组织使用的产品，Atlassian 提供了一种小团队能够负担的起的本地部署方案。同样开源里面也有很多的选择，包括基于 Moqui 和 Mantle 的 wiki 就用在了 HiveMind 的项目管理上。

注意下，内容沟通工具通常会与你的代码库隔离，尽管如果你把内容管理工具和你的代码库合并，那么参与你的项目的人会比较方便的使用它。因为 Moqui 框架本身支持渲染 wiki 页面和传递二进制的附件，所以你可能会认为这是个 Moqui 的组件。但这里最大的问题是，除非 Moqui 有个很完善好用的 wiki 应用能够很方便的更新内容，否则这将很难去吸引技术较弱的人来参与 Moqui 的项目。

对于实际的代码库也有很多的选择，但更多的会依赖于个人或者团队的倾向。Moqui 本身是存放在 GitHub 上的，并且 GitHub 上的私有库的价格是能接受的（特别是些小的项目库）。如果你正好也使用 GitHub，从 jonesde 的 moqui 库中切分支，在自己的私有库中维护自己的运行时目录，同时定期更新基础的主程序的代码即可。

对于项目开发来说，即使你不使用 GitHub，那么使用一个本地或者远程的 git 代码库也是个很好的管理代码的方式。当然，如果你倾向于使用 Subversion 或者 Mercurial，也是无可厚非的选择。

对于真正的开发过程来说，你也许需要一个编辑器或者开发环境支持以下类型的文件：

- XML（支持自动完成、验证、显示注释等）

- Groovy（groovy 脚本语法特性支持以及 XML 文件中的 groovy 脚本语法特性支持）
- HTML，CSS 和 JavaScript 脚本
- FreeMarker（FTL）
- Java（可选的）

我个人倾向使用的是 JetBrains 的 IDE 工具 IntelliJ IDEA。社区免费版对于 XML 和 Groovy 支持性非常好。对于 HTML，CSS，JavaScript 以及 FreeMarker 支持你需要购买商业版，否则 IDE 只是作为简单文本编辑器支持。我就是使用了商业版完成了大部分 Moqui 框架的实现，包括复杂的 FreeMarker 宏定义模板。在购买了个人授权使用商业版后，我开发起来真的很爽，但是社区版本身就已经很强大了。

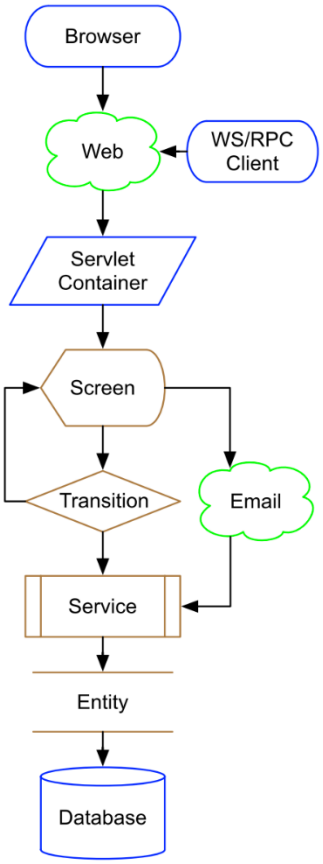
其他流行的 Java IDE 比如 Eclipse 和 NetBeans 同样也是很好的开发选择，它们提供了构建和插件功能去支持以上这些类型的文件。我个人喜欢开发工具有自动完成和其他高级的开发辅助功能，但是你如果喜欢简单的文本编辑器，那么选择自己喜欢的就好了。

Moqui 开发框架基于 Gradle（1.0 及以上版本）构建。虽然包括 IntelliJ IDEA 在内的 IDE 都提供了相当好的简化命令任务的用户界面操作支持，但是我个人更喜欢命令行的方式去构建运行项目。

纵览

Web 浏览器请求

一个 web 浏览器请求经由 Servlet 容器会传递到框架层面（默认实现是内嵌了 Winstone Servlet 容器，同样也支持 Tomcat 以及其他的 Java Servlet 规范容器）。Servlet 容器通过 web.xml 文件按照标准的方式在服务器上找到安装的 MoquiServlet，并查找到相应的请求路径。MoquiServlet 很简单，仅仅设置了一个执行上下文，然后渲染请求的界面。



Web 应用的界面渲染源自于配置的“根”界面，然后希望获取到的目标界面都通过子界面路径的方式向下查找。除了目标界面路径，可能也同时会有一个转换名称用于跳转到目标界面。

转换用于处理界面跳转的输入数据（而不是准备展现的数据），区别于界面上的动作，动作通常是为了准备呈现的数据（不是处理跳转的输入数据）。如果存在一个转换名称，那么这个转换对应的服务和动作就会运行，同时会反馈一个转换指定的响应（基于条件约定而不管是否存在错误），然后通常这个响应会在跳转到另一个界面进行反馈。

当一个服务被调用（通常来源于转换或者界面动作），服务门面会按照服务定义去验证并清空输入的 map，然后调用执行定义好的内部或者外部脚本，Java 方法，自动或隐式的实体操作或者远程服务。

与数据库交互的实体操作，只能通过服务进行写操作，同时任何时候都可以通过界面的动作去进行读操作（转换、界面动作、服务脚本/方法等都可以支持读操作）。

Web 服务调用

Web 服务请求通常伴随着一个浏览器端的表单提交请求发起，并受控于界面跳转。请求过来的数据受控于跳转的动作，同时，action 动作控制的响应比较特殊，需要以 XML，JSON 等格式进行返回。跳转/转换的默认响应需要被设置为“none”类型，这样就不会去请求渲染界面或者重定向到某个界面了。

邮件的接收、发送

接收邮件通过 pollEmailServer 服务（配置使用 email 邮件服务的实体）进行邮件的 ECA 规则来控制。这些规则里含有解析邮件以及格式化成有效的 maps 格式的信息。如果规则条件满足，就会执行规则定义的动作行为。规则能够支持你想要做的任何事情，比如在某些地方保存某些信息，在目录中添加一个用于回顾的队列，生成一个自动触发的响应等。

发送邮件最简单的方式就是调用 sendEmailTemplate 服务。这个服务通过 emailTemplateId 查找到 EmailTemplate 记录，就能获取到这个待发送邮件的配置信息，包括标题、发送地址、要展示的 XML 界面、附件等各种其他操作。这意味着可以适用于各种邮件场景，特别是通知公告类的信息，以及系统管理的自动回复客户的服务信息等。

二、 运行 Moqui

下载 Moqui 以及必备软件

Moqui 框架默认的配置软件只需要 Java 的 JDK 6 及以后版本（推荐 JDK 7）。源码构建编译框架需要 Gradle 1.6 及以后版本。

你可以从 SourceForge 下载 Moqui 框架：https://sourceforge.net/projects/moqui/files/

选择最新版本的目录，然后任意选择二进制或者源码发布包。框架的二进制发布包以“moqui-<版本号>.zip”命名，源码包以“moqui-<版本号>-src.zip”命名。

Moqui 框架的最新源码可以通过 GitHub 进行下载和在线预览，地址为：
https://github.com/jonesde/moqui

同样的业务地幔构件的 GitHub 最新代码地址为：
https://github.com/jonesde/mantle

虽然你可以单独的从 GitHub 上下载 Mantle 地幔构件，但是在 SourceForge 上也有一个内嵌 Moqui 框架环境的 Mantle 整包。

运行时目录以及 Moqui 的 XML 配置文件

Moqui 框架部署运行主要有三个核心部分：

- 可执行的 WAR 包文件（详细见下）
- 运行时目录
- Moqui 配置文件（XML 格式）

不管你怎么使用这个可执行的 WAR 文件，你必须拥有一个运行时的目录，同时你也许会重写 Moqui 默认配置文件（MoquiDefaultConf.xml 文件）里的配置信息，例如在运行时目录或者 conf 目录中的 MoquiProductionConf.xml 文件里面重写。

运行时目录主要放置的是你想要加载的组件、应用系统的入口文件（界面入口文件）以及配置文件。同时，框架会在这个目录下存放日志文件，Derby 数据库文件（如果你使用 Derby 数据库的话）等。也许你最终是要在自己的代码库里面创建自己的运行时目录，你可以把默认的运行时目录作为你的工程的起始点。

运行的时候，需要指定下如下两个属性：

moqui.runtime	指定运行时目录（如果存在 runtime 的子目录，默认配置为“./runtime”;如果不存在，则配置为“.”）
moqui.conf	指定 Moqui 的运行时配置文件（类似上面配置，使用 URL 或者路径方式配置）

这里有两种方式去指定这两个属性：

- 通过编译路径下的 MoquiInit.properties 文件
- 在命令行通过系统属性去指定（使用 java -D 参数）

可执行的 WAR 文件

耶，是的：就是一个可以执行的 WAR 文件。你主要做的就是下面这些事情（通过简单的命令去按需展示和修改）：

加载数据	$ java -jar moqui-<version>.war -load
运行内置的 web 服务	$ java -jar moqui-<version>.war
WAR 包形式部署（Tomcat 等）	$ cp moqui-<version>.war ../tomcat/webapps
显示配置和帮助信息	$ java -jar moqui-<version>.war -help

当运行数据加载器时（使用 -load 参数），以下的附加参数都是可用的：

-types=<type>[,<type>]	加载指定的数据类型，适配 entity-facade-xml.@type 属性（可以是任意的数据，通常为：种子数据，种子初始化数据，样例数据等）
-location=<location>	加载指定路径下的单个数据文件
-timeout=<seconds>	每个文件加载事务的超时时间，默认为 600 秒（10 分钟）
-dummy-fks	使用虚拟外键，规避引用完整性错误
-use-try-insert	尝试插入、更新操作来替代优先检查数据记录的异常校验
-tenantId=<tenantId>	加载指定租户 ID 的数据

注意下，如果命令中没有-types 或者 -location 参数，将会加载所有已知类型的数据文件。

上面显示的运行时配置 moqui.runtime 和 moqui.conf 属性的例子来源于编译路径下的 MoquiInit.properties 文件。通过命令行来指定这些参数的例子如下：

$ java –D moqui.conf=conf/MoquiStagingConf.xml -jar moqui-<version>.war

注意：moqui.conf 参数的路径关联的是 moqui.runtime 的目录，或者换句话说，指定的是相应运行时目录下的配置文件。

当运行内嵌的 web 服务（除了使用-load 和-help 参数）使用的是 Winstone Servlet 容器。完整的 Winstone 可用参数列表，请参见： http://winstone.sourceforge.net/#commandLine

为了方便使用，此处罗列下常用的 Winstone 参数：

--httpPort	设置 http 的监听端口。-1 表示禁用，默认为 8080
--httpListenAddress	设置 http 的监听地址。默认为所有接口
--httpsPort	设置 https 的监听端口。-1 表示禁用，默认禁用
--ajp13Port	设置 ajp13 的监听端口。-1 表示禁用，默认为 8009
--controlPort	设置关闭/控制的端口。-1 表示禁用，默认禁用

在 WAR 文件中内嵌运行时目录

Moqui 框架能够运行加载一个外部的运行时目录（独立于 WAR 文件之外），也支持运行时目录打包在 WAR 文件内部的方式。内嵌的方式特别适用于类似 Amazon ElasticBeanstalk 这种 WAR 部署方式的容器。创建一个内嵌运行时目录的 WAR 文件步骤如下：

1. 在运行时目录下添加必须的组件以及其他的资源文件
2. 按照需要调整 ${moqui.home}/MoquiInit.properties 文件
3. 必要时调整 Moqui 的配置文件（runtime/conf/Moqui*Conf.xml）
4. 基于 moqui.war 文件、你的运行时目录的所有文件以及 MoquiInit.properties 配置文件，打一个完整 WAR 包的方式为以下任意一种：
   a) $ gradle addRuntime
   b) $ ant add-runtime
5. 拷贝创建好的 WAR 文件（moqui-plus-runtime.war）到部署的目标位置
6. 运行服务（或者重启/刷新去自动更新运行的 WAR）

最终的 WAR 文件的根目录下就是运行时目录（和 WEB-INF 目录同级），并且所有的 JAR 文件都在 WEB-INF/lib 目录下。

编译 Moqui 框架

Moqui 框架使用自动编译工具 Gradle（http://www.gradle.org）去编译源码。虽然有各种自定义的能自动处理常用的构建任务的工具，但最常用的还是 Gradle 自带的构建编译任务。框架内部还有一个 Ant 的构建编译文件，但是这个文件内只是一些常用任务，不包含编译源码的任务。

编译 JAR，WAR 文件	**$ gradle build**	
加载所有的数据	**$ gradle load**	$ ant load
运行 WAR 内的服务	**$ gradle run**	$ ant run
清除 JAR，WAR 文件	**$ gradle clean**	
清空所有的编译文件和运行时创建的文件（日志、数据库文件等）	**$ gradle cleanAll**	

注意在 Gradle 里面，加载（load）和运行（run）任务都依赖于构建（build）任务。由于这种依赖存在，最简单的方式去在一个流行的数据库上创建一个新的开发环境可以通过执行这个命令： $ gradle load run

它将会创建 war 文件，运行数据加载器，然后启动服务。停止服务只需要键入**<ctrl-c>**（或者你自定义的快捷方式）。

数据库配置

数据库（或者数据源）的安装配置已经在 Moqui 框架的 XML 配置文件的 moqui-

conf.entity-facade.datasource 元素中完成了。这里有个元素用来为每个实体分组使用，并且 datasource.**group-name** 属性与 entity.**group-name** 属性相对应匹配。默认的 Moqui 框架有四个实体分组：transactional，analytical，nosql 以及 tenantcommon。如果你仅在数据源 datasource 中配置了 transactional 分组，那么其将应用于其他的分组之上。只有一种情况例外：如果你想要开发多租户的应用，那么你必须也要在数据源 datasource 中定义 tenantcommon 分组。

下面是默认提供的 Apache Derby 数据库的配置：

```
<datasource group-name="transactional"    database-conf-name="derby"
    schema-name="MOQUI">
    <inline-jdbc pool-minsize="5" pool-maxsize="50">
        <xa-properties databaseName="${moqui.runtime}/db/derby/MoquiDEFAULT"
            createDatabase="create"/>
    </inline-jdbc>
</datasource>
```

database-conf-name 元素指定了某种数据库配置并且需匹配 database-list.database.**name** 元素定义的那些已支持的数据库。数据库配置指定了类似使用的 SQL 标准类型，SQL 语法操作以及 JDBC 驱动等细节信息。

本例通过使用 xa-properties 元素去指定在 JDBC 驱动中使用 XA（事务处理方面）接口。每个 JDBC 驱动都可以指定配置这个属性元素。在 MoquiDefaultConf.xml 文件中包含了一些可以参考的例子，但完整的操作清单请查阅 JDBC 驱动的文档。

下面是一个使用非 XA 接口配置 MySQL 数据库的例子：

```
<datasource   group-name="transactional"   database-conf-name="mysql"   schema-name="">
    <inline-jdbc jdbc-uri="jdbc:mysql://127.0.0.1:3306/MoquiDEFAULT?autoReconnect=
        true&amp;useUnicode=true&amp;characterEncoding=UTF-8"
          jdbc-username="moqui"   jdbc-password="moqui"
          pool-minsize="2"   pool-maxsize="50"/>
</datasource>
```

对于非 XA 接口的配置来说，各种 jdbc-*开头的属性,而不是子元素存在于 inline-jdbc 元素节点中。本例中展示了主要的必须项：JDBC URI，用户名和密码。在运行时目录的 Moqui XML 配置文件（例如 MoquiProductionConf.xml 文件）中，你可以在 entity-facade 元素节点下添加 database 元素节点来使用其他的数据库配置。

本书由 Ant Websystems 进行赞助（http://www.antwebsystems.com/）

三、 框架工具和配置

下面的内容是一个关于 Moqui 框架中各种工具，以及 Moqui XML 配置文件中相应配置元素的概要总结。默认设置存于 MoquiDefaultConf.xml 文件里，它被包含在 Moqui 框架发布的二进制可执行 WAR 文件中。这是个非常好的文件，可以查看它去看一下那些可用的设置以及它们默认的配置内容。如果你下载的是 Moqui 框架的二进制发布包，你可以在线查看这个文件（注意一下这里来源于 GitHub 上的主分支，并可能和你下载的文件有些许的差异）：
https://github.com/moqui/moqui/blob/master/framework/src/main/resources/MoquiDefaultConf.xml

这个文件中任何的设置都可以被运行时目录中指定的 Moqui XML 配置文件所覆盖（并通常在 runtime 目录下的 conf 目录中）。这两个文件在任何设置被使用之前将被合并，并且运行时文件会覆盖默认文件中的配置。因此，改变配置的一个简单途径就是从默认配置文件中拷贝粘贴到运行时的配置文件中，然后就能得到期望的变化了。

执行上下文和 Web 门面

执行上下文是 Moqui 框架 API 中的核心对象。这个对象在单个服务器交互（如一个 web 界面请求或远程服务调用）的上下文中维持状态。通过 ExecutionContext 对象，你可以访问一堆的“门面”，它们用于访问框架各个不同部分的功能。下面会有这些门面的细节。

执行上下文追踪的主要状态是个变量空间，或“上下文环境”，它用于界面，动作，服务，脚本甚至实体以及其他的操作。这个上下文是个哈希表或键值对的 map。它通过 ContextStack 类实现并支持使用 **push**()和 **pop**()方法（它们将所有的上下文转化为一个 map 的堆栈）去保护变量空间。由于不同的构件要执行，它们自动的在写入前 **push**()将上下文入栈，然后在构件完成前 **pop**()弹出上下文恢复其状态。写入上下文总是将数值放到栈的顶部，但是读取已命名的数值则是从顶部开始查找栈的每一层，这样更深层的字段也是可访问的。

有些场景下，比如调用一个服务，我们想要一个干净的上下文去更好的隔离构件和调用它的构件。对于这种情况我们使用 **pushContext**()方法去获得一个干净的上下文，然后在构件运行之后使用 **popContext**()方法去恢复到原始的上下文。

上下文是构件可能执行的任何变量空间。在界面中当 XML 动作执行时，结果返回到本地上下文中。甚至服务内嵌的 Groovy 脚本以及界面动作共享一个变量空间，所以在上下文中变量是公开存在的，便于随后的构件使用。

你将在 Moqui 所基于的代码（使用 Groovy 语法）中看到一些常用的表达式。包括：

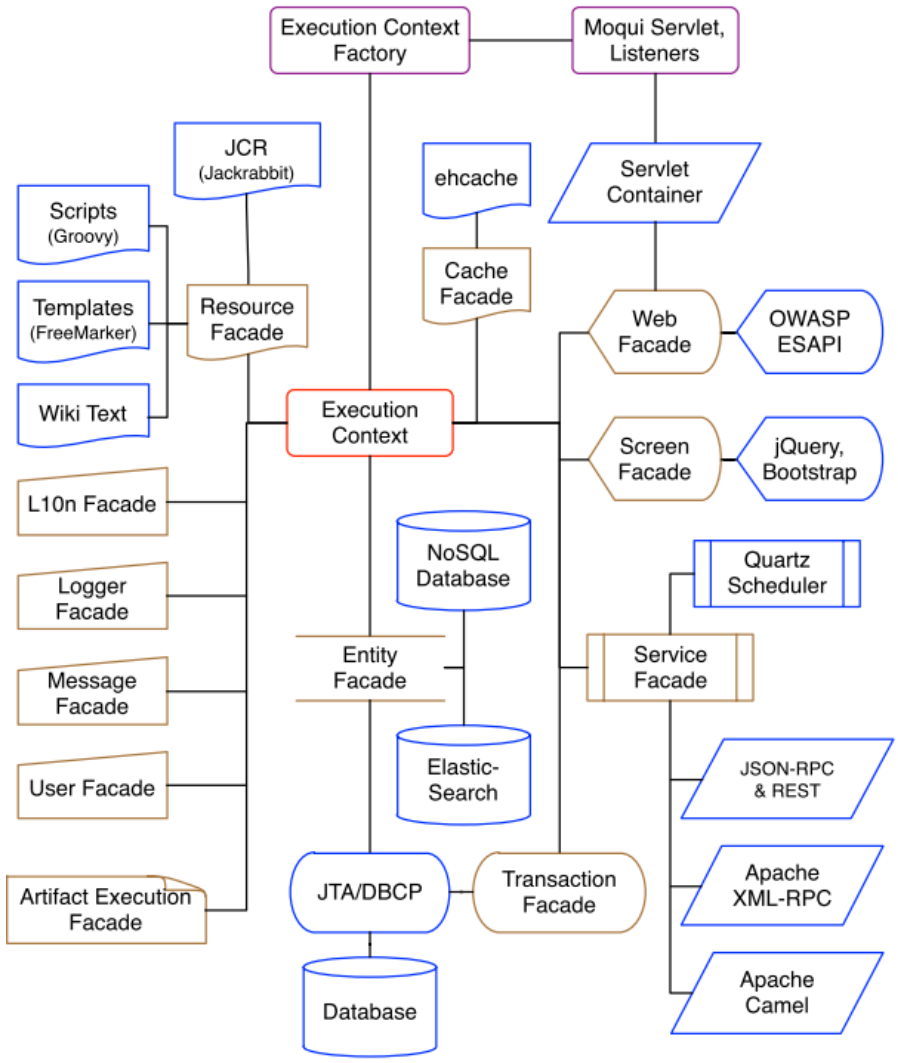
- ec.context：指的是当前变量上下文
- ec.context.**exampleId**：指的是上下文中的“exampleId”字段
- ec.context.**exampleId** = "foo"：设置 exampleId 字段的值为“foo”
- **exampleId** = "foo"：在内联的脚本中你同样可以这样做

对于作为 web 请求（HttpServletRequest）的一部分，一个创建的 ExecutionContext 实例有一个特殊的门面叫做 web 门面。这个门面用于访问上下文的 servlet 环境信息，包括请求，响应，会话以及应用（ServletContext）。它同样用于访问 servlet 环境中各种这些部分的状态（属性），包括请求参数，请求属性，会话属性和应用属性。

Web 参数

请求参数"map"（ec.web.**requestParameters**）是一个特别的 map，它包含了源于 URL 参数字符串的参数，内联 URL 参数（使用"/~name=value/"格式），以及源自提交参数的多个部分（当适用时）。这里还有个特殊的参数 map（ec.web.**parameters**），它按照以下的顺序（之后的覆盖之前的）将所有其他的 map 组合在一起：请求参数，应用属性，会话属性和请求属性。参数 map 类似于上下文是一个请求属性 map 的栈，所以如果你要对其进行写入，数值将会被放在栈的顶部。

为了安全的因素，请求参数 map 使用 OWASP ESAPI 类库进行规范化转换以及过滤。它和服务门面校验一起有助于防止跨站脚本攻击（XSS）和注入攻击。



工厂，Servlet&监听器

执行上下文实例通过执行上下文工厂（Execution Context Factory）进行创建。当有需要的时候可以直接通过编码的方式直接去创建，但通常都是通过 Moqui 框架所运行的容器去创建。

运行 Moqui 框架最常见的方式就是作为一个 web 应用去运行，不管是通过部署一个 WAR 包在一个 servlet 容器或应用服务器中，还是通过使用内嵌的 Winstone Servlet 容器去运行可执行的 WAR 文件。这两种情境下 Moqui 的根 web 应用都会被加载，并且 WEB-INF 目录下的 web.xml 文件会告诉 servlet 容器去加载 MoquiServlet，MoquiSessionListener 和 MoquiContextListener。这些都是框架中包含的默认类，并且如果你想要改变 ExecutionContextFactory 和 ExecutionContext 的生命周期，你可以明确的创建你自己的类。

MoquiContextListener 在 **contextInitialized**()事件中创建了 ExecutionContextFactory 对象，并在 **contextDestroyed**()事件中销毁它。MoquiServlet 的工厂为每个请求在 **doGet**()和 **doPost**()方法中创建了 ExecutionContext 对象，同时在每个请求的最后，使用相同的方法去销毁这个对象。

资源和缓存门面

资源门面用于访问和执行资源，例如脚本，模板和内容。缓存门面用于缓存上的常规操作，并作为一个缓存 Cache 接口的实现去获得一个缓存的引用。与支持的基本 get/put/remove 等操作一起，你可以获得每个缓存的统计以及修改缓存的属性，例如超时，大小限制以及清理算法。默认的缓存门面实现就是 ehcache 的一个封装，除了在 Moqui XML 配置文件中对缓存门面进行配置，你还可以使用 ehcache.xml 文件进行额外的配置选项。

资源门面使用缓存门面按照文件源的位置（通过 **getLocationText**()方法）去缓存纯文本，通过路径位置去编译 Groovy 和 XML 动作脚本（使用 runScriptInCurrentContext 里的方法），以及同样通过路径位置去编译 FreeMarker（FTL）模板（通过 **renderTemplateInCurrentContext**()方法）。

这里同样有缓存用于贯穿在界面 XML 和表单定义中的分散的小型 Groovy 表达式，并且缓存以表达式实际的文本作为键而不是表达式来源的位置（通过 **evaluateCondition**()，**evaluateContextField**()和 **evaluateStringExpand**()方法）。

为了更通用的资源访问，**getLocationReference**()方法返回一个 ResourceReference 接口的实现。它可用于读取资源内容（包括文件和目录），并获得关于内容/MIME 类型，最后修改时间及其所在的位置等信息。框架剩余的部分使用这些资源引用以一种通用的扩展的方式去访问资源。ResourceReference 接口可以按需进行实现，并且已存在的默认实现包含了后面的协议/体制：http，https，file，ftp，jar，classpath，component 以及 content（JCR，例如 Apache Jackrabbit）。

界面门面

界面门面的 API 迷惑性的简单，主要就是充当了 ScreenRender 接口实现的工厂。通过 ScreenRender 接口你可以在各种上下文中渲染界面，最常见的就是在服务中不依赖于 Servlet 容器进行渲染，或使用便利的 ScreenRender.**render**(request, response)方法在响应一个 HttpServletRequest 请求时进行呈现。

通常当进行渲染时，你将指定根界面的位置来渲染根界面，同时可选的指定一个子界面路径去渲染子界面（如果根界面有子界面，并且使用子界面来替代默认条目的界面）。对于 web 请求来说，子界面的路径就是简单的"pathInfo"请求（URL 路径中 web 应用/servlet 安装位置后面残余的部分）。

界面定义

界面门面真正的“魔法”是在界面定义 XML 文件中。每个界面定义都可以指定 web 设置，参数，响应转换，子界面，展现前动作，展现时动作以及部件。部件包括子界面菜单/激活/面板，截面，容器，容器面板，指定渲染模式的内容（例如 html，xml，csv，text，xsl-fo 等等）以及表单。

这里有两种表单类型：单个表单和列表表单。它俩都有各种布局选项并支持各种各样的字段类型。虽然界面表单主要是定义在界面 XML 中，但是它们同样也可以使用 DbForm 和关联实体去为不同的用户组进行扩展。

注意下基于服务（使用 **auto-fields-service** 元素）的表单非常重要的一点是，各种客户端的校验基于服务中与表单字段相关的校验定义，将会被自动的添加。

界面/表单渲染模板

ScreenRender 的输出，通过为界面和表单定义的 XML 元素而运行的宏定义模板进行创建。如果模板是通过 ScreenRender.**macroTemplate**()方法指定的，那么它将会被使用，否则模板的呈现将决定于渲染模式 **renderMode**，以及 Moqui XML 配置文件中的 screen-facade.screen-text-output 元素配置。你可以创建自己的模板去覆盖默认模板，或简单的完全忽略它们，同时在 Moqui XML 配置文件中配置它们来获得你想要的输出。这里有个这样的模板例子位于 runtime/template/screen-macro/ScreenHtmlMacros.ftl 文件中，同时需覆盖的配置文件为 runtime/conf/development/MoquiDevConf.xml。

默认的 HTML 界面和表单模板使用了 jQuery Core 和 UI 作为动态的客户端交互。其他的 JS 类库可以像上面描述的那样用于修改界面 HTML 宏定义，以及指定期待的 JavaScript 和 CSS 文件去改变主题数据（默认配置在 runtime/component/webroot/data/WebrootThemeData.xml 文件中）。

服务门面

服务门面通过大量的同步的，异步的，计划的以及特殊的（TX 提交/回滚）服务调用接口去调用服务。每个接口都有不同的方法去构建你想要调用的信息，同时它们都有方法去处理服务的名称和参数。

当一个服务被调用时，调用方无需了解它是如何执行的或服务位于何处。服务定义抽象出服务的语义定义，因此那些细节是服务实现的部分，并不是服务调用的部分。

服务命名

服务命名由 3 个部分构成：路径，动词和名词。当引用一个服务时，它们的组合形式为："**${path}.${verb}#${noun}**"，这里井号符号是可选的，但可用于确保动词和名词是完全匹配的。路径应当是一个 Java 包风格的路径，例如 org.moqui.impl.UserServices 指的就是文件 classpath://service/org/moqui/impl/UserServices.xml。虽然指定路径的方式有点不方便，但是它更易于去组织服务，基于服务的调用去找到定义，并提升性能以及进行缓存（自从框架可

以按需进行懒加载服务定义)。

服务定义文件将会基于框架的位置模式路径被查找到："classpath://service/$1"和"component://.*/service/$1"，这里的$1 就是路径，'.'会被转化为'/'，同时在路径的最后加上".xml"。

服务名称的动词（必须项）和名词（可选项）部分分隔开来是为了更好的描述服务做什么和服务执行在哪个实体上。当服务执行在一个指定的实体上，名词就是那个实体的名称。

服务门面唯一的基于实体定义支持 CrUD 操作。使用这些实体隐式服务需要使用由一个无路径的，一个增加、修改或删除的名词，一个哈希（#）标识，以及实体的名字所组成的服务名称。例如更新一条 UserAccount 的记录使用的服务名称为 **update#UserAccount**。当定义实体自动 entity-auto 服务时，名词必须为实体的名称，同时服务门面将会使用出入参及实体定义去决定做什么（对于新增操作的主/次要序列 IDs 值最有帮助，等等)。

上面段落中例子的完整服务名称如下所示：

org.moqui.impl.UserServices.**update#UserAccount**

参数清理，转换和校验

当调用服务时你可以传递任何你想要的参数，同时服务调用者将会清理基于服务定义的参数（移除未知参数，改变类型等），并在将参数放到上下文中以供服务运行之前，基于服务定义中的校验规则进行参数校验。当服务运行时，参数将会和继承的上下文记录值一起存在 ec.context 的 map 中，同时参数将会在上下文参数的一个 map 中，与其它上下文参数隔离的去访问。

有一个重要的验证是通过服务定义中的 parameter.**allow-html** 元素去进行配置的。默认不允许任何 HTML，但你可以为服务参数使用这个属性去允许任何 HTML 或仅仅安全的 HTML。安全的 HTML 取决于使用的 OWASP ESAPI 以及 Antisamy 类库，同时在 antisamy-esapi.xml 文件中配置经过深思熟虑的安全策略。

Quartz 调度

服务门面使用 Quartz 调度去进行异步的以及排程的服务调用。Moqui XML 配置文件中有一些可用的选项用于调用服务，但是 Quartz 本身的配置使用 quartz.properties 文件（这里有个默认的在 framework/src/main/resources/目录中，并可以在这个 classpath 上覆盖它)。

Web 服务

对于 web services 服务来说，服务门面使用 Apache XML-RPC 用于传入和呼出的 XML-RPC 服务调用，同时自定义代码使用 Moqui JSON 以及 web 请求工具用于传入和呼出的 JSON-RPC 2.0 的调用。呼出的调用通过 RemoteXmlRpcServiceRunner 和 RemoteJsonRpcServiceRunner 类进行处理，它们通过 Moqui XML 配置文件中的 service-facade.service-type 元素进行配置。通过实现服务门面的 ServiceRunner 接口（就像上面两个类所做的一样）和配置文件中添加一个 service-facade.service-type 类型元素，可以添加支持其他的呼出服务调用。

传入的 webservices 服务使用 runtime/component/webroot/screen/webroot/rpc.xml 界面中默认定义的转换去处理。如果 webroot.xml 被安装到服务器的根路径（"/")，这些远程 URL

应该看起来如："http://hostname/rpc/xml"或"http://hostname/rpc/json"。为处理其他类型传入服务的类似界面转换可被添加到 rpc.xml 界面或其他任何界面中。

对于 REST 风格的服务来说，界面转换可以使用 HTTP 请求的方法（get，put 等）来进行声明，和用名称去匹配传入的 URL 一样。为了更灵活的支持 URL 中参数的使用，除了在 URL 路径值中放置转换以外，还可以将转换配置为与命名的参数相同的名称。为了更易于 JSON 的负载，它们同样可以与参数自动地进行映射，看起来就像参数来源于其他的源头，这种方式很轻易的允许服务端的代码复用。一个很少配置的内部服务会被调用去处理这些 REST 服务转换，它提供了高效的映射机制用于外曝的 REST 服务和内部服务之间。

实体门面

实体门面用于普通的数据库交互，包括新增/修改/删除以及查找操作，以及用于更专门的操作，如加载和创建实体 XML 数据文件。虽然这些操作是通用的并覆盖了典型应用中大部分的数据库交互需求，但有时你需要底层访问，你可以通过 Moqui XML 配置文件中基于 entity-facade 数据库配置的实体门面来获得一个 JDBC 连接对象。

实体相当于数据库中的表，并主要被定义在 XML 文件中。这些定义包含了罗列实体中的字段，实体间的关系，指定索引等等。实体可以使用 UserField 及其关联实体的数据库记录来进行扩展。

每个单条记录都通过一个 EntityValue 接口的实例来表现。这个接口为了方便起见扩展了 Map 的接口，同时添加了额外的方法用于获得指定的数据集合，例如主键值。接口同样有用于数据库交互的方法对指定记录进行增加，修改，删除，刷新，获取/设置主/次要序列 ID 值，以及基于实体定义里的关联关系去查找关联记录。使用 EntityFacade.**makeValue**()方法去创建一条新的 EntityValue 对象，尽管大部分的时候你将会通过一个查询操作获得 EntityValue 实例。

使用 EntityFind 接口进行实体记录的查询。通过 EntityFacade.**makeFind**()方法获得这个接口的实例。这个查询接口允许你为查询设置各种的条件（用于 where 和 having 关键字，更多便捷的方法用于 where），指定选择的字段和排序字段，设置偏移和限制数值，以及包括使用缓存，更新及去重的标记。一旦操作设置完成，你就可以调用方法进行实际的查询，包括：**one**()，**list**()，**iterator**()，**count**()，**updateAll**()以及 **deleteAll**()。

多租户

当从执行上下文 ExecutionContext 中获得一个实体门面 EntityFacade 实例时，这个重新取回的实例将会用于当前上下文 ExecutionContext（它将在指定的用户认证之前或通过 servlet，亦或是请求执行前的监听器去进行设置）中活跃的租户 tenantId。如果没有 tenantId，实体门面 EntityFacade 将会用于默认的"DEFAULT"租户并使用 Moqui XML 配置文件中的设置。否则，它将会使用活跃的 tenantId 去查找 Tenant*实体上的设置，去覆盖 Moqui XML 配置文件中默认的数据源设置。

连接池和数据库

实体门面使用 Atomikos TransactionsEssentials 或 Bitronix BTM (默认的)用于 XA-aware 数

据库连接池。使用 jta.properties 文件去配置 Atomikos，bitronix-default-config.properties 文件去配置 Bitronix。通过 Moqui XML 配置文件中 entity-facade 元素的配置，你可以修改默认的连接池并使用任何的 DataSource 或替代的 JNDI XADataSource。

Moqui 框架中包含的默认数据库是 Apache Derby。它很容易通过 Moqui XML 配置文件中 entity-facade 元素的配置去改变。添加一个新的 database-list.database 元素可以新增一个在 MoquiDefaultConf.xml 文件中尚不支持的数据库。当前默认支持的数据库包括 Apache Derby，DB2，HSQL，MySQL，Postgres，Oracle 以及 MS SQL Server。

数据库元数据

实体门面在一个实体上第一次（在 Moqui 框架每次运行时）进行数据库操作时，它将会检查去看实体对应的物理表是否存在（除非配置为不做表的存在检测）。你同样可配置其在启动时检查所有实体的物理表。如果物理表不存在，框架将会基于实体定义去创建表，索引以及外键（为已存在的关联表）。如果物理表存在，框架将会检查并添加缺失的列字段，对于索引和外键也做同样的操作。

事务门面

事务主要用于服务和界面。服务定义上有事务设置，基于此服务调用者可以按需去暂停/恢复以及开始/提交/回滚事务。对于界面来说，事务总是开始于转换（如果当前没有其他的事务），并用于渲染实际的界面，事务仅会开始于如果界面设置使用了事务（大部分是性能的原因）。

你同样可以使用事务门面 TransactionFacade 去手动划分事务边界。JavaDoc 文档中有一些推荐模式的代码样例使用了 try/catch/finally 子句去开始/提交/回滚以及暂停/开始/提交/回滚/恢复事务，来确保事务被合理的管理。

当调试事务的问题时，例如追踪一个仅回滚（rollback-only）的设置在何处，事务门面 TransactionFacade 同样可以使用，因为当调用 **setRollbackOnly()**方法时事务门面保持了一个追踪堆栈。它将在稍后的错误中自动进行日志记录，同时你也可以在其他时间手动的去获得这些日志记录信息。

事务管理（JTA）

事务门面默认使用 Bitronix TM 类库（同样用于实体门面的连接池）。使用 bitronix-default-config.properties 文件配置 Bitronix。Moqui 框架同样开箱即用的支持 Atomikos，通过 jta.properties 文件配置。

任何例如来源于应用服务器的一个 JTA 事务管理，可用于替代配置于 Moqui XML 配置文件里 entity-facade 元素中 UserTransaction 和 TransactionManager 实现的 JNDI 方式。

构件执行门面

构件执行门面被其他门面所调用，用以保持追踪执行上下文 ExecutionContext 生命周期

中哪些构件正在“运行”。它保持了所有构件的一个历史以及当前运行构件的一个堆栈。举个例子，如果一个界面调用了一个子界面，同时子界面调用了一个实体查询的服务，堆栈将为（从下到上）第一个界面，然后是第二个界面，接着是服务，最后是实体。

构件认证

在有助于调试和满足好奇心的同时，保持追踪构件栈的主要目的是为了授权和权限。对于 Moqui 框架中的界面，转换，服务和实体这里有隐式的权限。其他的也许稍后会被添加上，但这里的是最重要的并在 1.0 版本中所支持的一种（查看 SecurityTypeData.xml 文件中的 "ArtifactType" Enumeration 记录获取细节）。

ArtifactAuthz*和 ArtifactGroup*实体用于配置用户（或用户组）访问指定构件的授权。为简化配置可以使用“可以继承的”授权，这意味着不仅仅是指定的构件可以用继承的方式授权，而且任何东西都可以使用。

在 ExampleSecurityData.xml 文件中有各种设置不同授权模式的例子。一个常用的授权模式是允许访问一个界面及其所有子界面，并且这个界面是个高等级的界面，类似于 ExampleApp.xml 界面是案例应用的根界面。另一个常用的模式是应用中只有某一个界面是被授权了，其余不是。如果一个子界面被授权了，甚至如果其父界面没有授权，用户也能够使用这个子界面。

构件点击追踪

这里同样有功能用于构件“点击”的性能数据追踪。这是通过执行上下文工厂完成而不是构件执行门面，因为构件执行门面为每个执行上下文进行创建，同时构件点击的性能数据需要通过大量当前以及一个时间段内的构件点击进行追踪。构件点击的数据在 ArtifactHit 和 ArtifactHitBin 实体中进行存储。ArtifactHit 的记录关联 Visit 记录（每个 web 会话都有一个访问 visit），这样你可以查看一个访问审计，用户体验回顾以及用于其他各种目的的点击历史。

用户，本地化，消息和日志门面

用户门面用于管理当前用户和访问，登录，授权及登出的信息。用户信息包括区域设置，时区以及币种。这里同样有选项去设置系统的当前日期/时间被视为使用用户的有效日期/时间（通过 ec.user.nowTimestamp），而不是使用当前系统的日期/时间。

L10n（本地化）门面使用用户门面中的区域设置，并使用来源于 LocalizedMessage 实体的缓存数据去本地化其接收到的消息。实体门面 EntityFacade 同样可以使用 LocalizedEntityField 实体去本地化实体的字段。本地化门面同样有格式化币种金额的方法，还有按需使用源于用户门面的 Locale 和 TimeZone 去解释和格式化 Number, Timestamp, Date, Time 以及 Calendar 对象的方法。

消息门面用于追踪用户的消息以及错误消息。错误消息列表（ec.message.errors）同样用于确定在一个服务调用或其他动作中是否存在错误。

日志门面用于记录日志信息到系统日志里。这意味着在脚本和其他普通日志记录中使用日志。为了更精确的和可追溯的进行日志记录，代码应该直接使用 SLF4J 日志类（org.slf4j.Logger）。LoggerFacade 接口的 JavaDoc 文档解释中包含了这些例子代码。

扩展和附加组件

引人注目的组件

Moqui 框架的组件是一个构建于 Moqui 之上、组成一个应用的构件集合，或是意味着能被其他组件（如 mantle-udm 和 mantle-usl 组件，主题组件，或集成 Moqui 框架中其他的工具和类库来扩展基于 Moqui 潜在范围的应用组件）所使用的、可复用的构件。

组件目录结构

组件的结构相对于配置是由约定进行驱动的。这意味着你必须使用下面这些指定的目录名称，并且你所看到的所有 Moqui 组件都使用相同的方式进行组织。

- **data –** 包含实体 XML 数据文件，其带有根元素 entity-facade-xml，按照和命令行指定的类型匹配的 **type** 属性进行加载（使用-load 的可执行 WAR 文件）或如果没有指定类型，可以为任何类型
- **entity –** 这个目录下存放的是所有将会被加载的实体定义和实体 ECA XML 文件；实体 ECA 文件必须放在这个目录中并有".eecas.xml"的双重扩展名
- **lib –** 当 web 应用被部署的时候，这个目录下的 JAR 文件将会被添加到 classpath 中
- **screen –** 约定这个目录下的界面会被明确的引用（通常通过"component://*" URL）
- **script –** 约定这个目录下的脚本会被明确的引用（通常通过"component://*" URL）；Groovy，XML 动作以及其他脚本都应位于这个目录下
- **service –** 服务将会按照服务定义 XML 文件中定义的路径进行加载，同时不但位于组件的服务目录之下还是位于"classpath://service/"之下的路径都会被找到；服务 ECA 文件必须放在这个目录中并有".secas.xml"的双重扩展名；邮件 ECA 文件必须放在这个目录中并有".emecas.xml"的扩展名

组件安装

加载组件

这里有两种方式去告知 Moqui 一个组件：

- 将组件目录放在 runtime/component 目录里
- 在 Moqui XML 配置文件中添加一个 component-list.**component** 元素

装载界面

每个 Moqui 框架中的 web 应用（包括默认的 webroot 应用）都必须有一个根界面，其指定在 moqui-conf.webapp-list.webapp.**root-screen-location** 属性中。默认根界面被调用时，将会加载位于 runtime/component/webroot/screen/webroot.xml 中的 webroot 应用。

对于源自你组件中位于 webroot 界面下、界面路径所指定的可用界面来说，你需要令你组件中的每个顶层界面（例如位于组件界面目录中的每个界面）成为 webroot 界面祖先的子

界面。 这里有两种方式去实现（这并不包括将其放在 webroot 目录下，作为一个隐式的子界面，因为那种方式不是在其他地方进行界面定义的选项）：

- 添加一个 screen.subscreens.subscreen-item 元素到父界面（子界面之上的界面）；举个例子，你可以查看”装载”了 example 和工具应用根界面的 apps 界面（runtime/component/webroot/screen/WebRoot/apps.xml）
- 添加一条 SubscreensItem 实体的记录，在 **screenLocation** 字段中指定父界面，**subscreenLocation** 字段中指定子界面，”装载点”在 **subscreenName** 字段中（等同于 subscreens-item.**name** 属性），同时在 **userGroupId** 字段中如果其值为 ALL_USERS，则表示适用于所有的用户；如果为一个实际的 userGroupId，则表示仅适用于这个用户组

如果你想要你的界面去使用其自己的样式主题并与其他界面独立开来，那么就直接把它放到 webroot 界面之下。如果要让你的界面变成默认应用菜单结构的一部分，并能够被默认应用的样式主题所呈现，那么就要将其放到 app 界面下。

➢ *Moqui XML 配置文件的设置*

你可能想要在你的组件中添加或修改各种源于 Moqui 框架默认配置的东西，包括：

- **Resource Reference:** 查看 moqui-conf.resource-facade.resource-reference 元素
- **Template Renderer:** 查看 moqui-conf.resource-facade.template-renderer 元素
- **Screen Text Output Template:** 查看 moqui-conf.screen-facade.screen-text-output 元素
- **Service Type Runner:** 查看 moqui-conf.service-facade.service-type 元素
- **Entity Data and Definition:**查看 moqui-conf.entity-facade.load-entity 及 moqui-conf.entity-facade.load-data 元素

既然框架使用了 Moqui XML 配置文件作为其默认配置，那么在 MoquiDefaultConf.xml 文件中就有上面这些所有的例子。

本书由 HotWax Media 进行赞助 (http://www.hotwaxmedia.com/)

四、 创建你的第一个组件

概要

本章是篇教你一步步创建并运行你自己的 Moqui 组件的向导，包含了人机交互界面，业务逻辑以及数据库交互。

- **第一步：**以创建一个“Hello world！”的界面开始你的第一个组件
- **第二步：**延续上步，你将会定义一个自己的实体（数据库表），然后添加表单去查询和创建实体的记录
- **第三步：**创建一些自定义的业务逻辑，而不使用框架自带的默认基于实体定义的 CrUD 逻辑操作

本文描述的运行方式就是简单的使用内嵌的 servlet 容器。

本文中向导例子的源码在官网上可以进行下载：http://www.moqui.org/tutorial.zip

第一步

下载 Moqui 框架

如果你还没下载 Moqui 框架就赶快去下载吧。你需要有一个 moqui-<版本号>的目录，用来存放最新版本的 moqui 的 war 文件以及默认的运行时目录。然后以这个目录为根目录开始我们的教程。

如果你是新下载的，快速的初始化数据并运行吧：

```
$ ant load
$ ant run
```

使用浏览器访问 http://localhost:8080/，点击界面左下方名称为“John Doe”的按钮登录，然后随便看一下演示环境。

现在退出系统（命令行界面按快捷键**<ctrl-c>**），你已经可以继续下一个步骤了。

创建一个组件

Moqui 依照“**约定/配置优于编码**”的原则进行组件开发，所以创建组件你所要做的就是创建目录：

```
$ cd runtime/component
$ mkdir tutorial
```

然后进入这个目录并创建一些后面要用到的标准目录：

```
$ cd tutorial
$ mkdir data
```

```
$ mkdir entity
$ mkdir screen
$ mkdir script
$ mkdir service
```

组件创建好了就启动框架吧（使用“$ ant run”或者其他类似的命令）。

添加一个界面

使用你常用的 IDE 或者文本编辑器添加一个界面的 XML 文件，存放于：
runtime/component/tutorial/screen/tutorial.xml

现在，制作个超简单的界面，上面只有“"Hello world!”的文本。XML 内容如下：

```
<screen require-authentication="false">
    <widgets>
        <label type="h1" text="Hello world!"/>
    </widgets>
</screen>
```

注意上面的 **require-authentication** 属性被设置为 false。默认的值为 true，并且这个界面会被要求授权和鉴权。我们会在后面的安全的章节中讨论可配置的自动识别的构件授权机制。

以子界面方式挂载

为了能让你的界面访问生效，需要将这个界面作为子界面挂载到根界面下某个已存在的界面下。在 Moqui 框架的界面机制中，访问界面的 URL 路径以及功能菜单的结构都是基于子界面的层级进行驱动的，所以这种挂载方式将会自动设置这个界面的访问 URL 路径并添加一个菜单分页。

因为这个入门例子的目标，我们将使用已有的根界面以及运行时目录中界面的头部、尾部等资源。运行时目录中有一个定义了根界面的 webroot 组件位于：
runtime/component/webroot/screen/webroot.xml

顺便说下，根界面在 Moqui 的 XML 配置文件中通过 **webapp-list.webapp.root-screen** 元素被指定，你可以为不同的域名配置多个入口的根界面。

为了使子界面层级定义的方式更灵活，入口根界面只有一个基础的 HTML 头和 body，没有头部和尾部的页面内容，所以我们把我们的界面放到“apps”界面下，“apps”界面中已有一个头部的菜单并提供了一些内容。修改 apps 的界面：
runtime/component/webroot/screen/webroot/apps.xml

apps.xml 文件中，在 subscreens 元素下添加一个 subscreens-item 元素，如下：

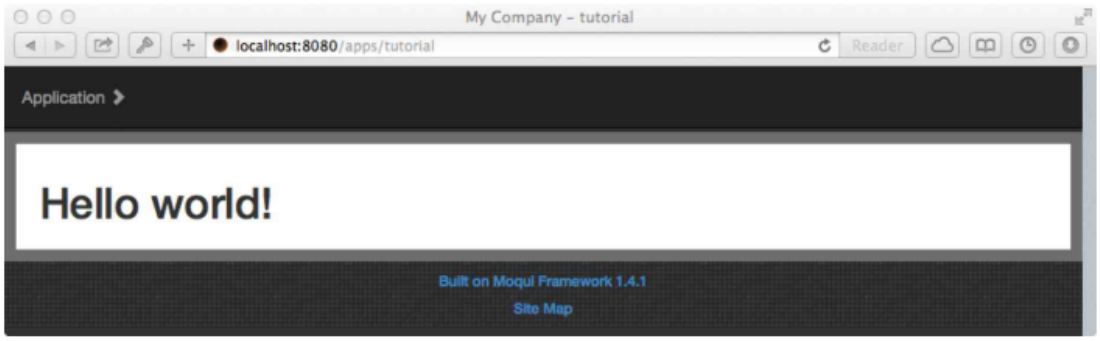
```
<subscreens-item name="tutorial" menu-title="Tutorial"
    location="component://tutorial/screen/tutorial.xml"/>
```

名称（name）属性用于指定 URL 访问这个页面的路径值，所以你的界面在浏览器上的访问路径为：http://localhost:8080/apps/tutorial

如果你不想修改已存在的界面文件，但是仍然希望将你的界面作为子界面挂载到其他界面下，你可以通过数据库记录的方式去达到这个目的。类似于（在 **entity-facade-xml** 中格式化代表实体的元素以及代表字段的元素）：

```
<SubscreensItem screenLocation="component://webroot/screen/webroot/apps.xml"
    subscreenName="tutorial" userGroupId="ALL_USERS"
    subscreenLocation="component://tutorial/screen/tutorial.xml"
    menuTitle="Tutorial" menuIndex="1" menuInclude="Y"/>
```

一旦完成，你的界面应该是这个样子的：



初试界面内嵌入内容

除了使用 label 节点元素，我们还可以在界面“下”使用 HTML 文件的方式。首先新建一个简单的 HTML 文件，放在：runtime/component/tutorial/screen/tutorial/hello.html

这个 HTML 文件能包含任何的 HTML 元素，同时既然界面的父界面又包含了头部/尾部等，这个 HTML 文件又被包含在这个界面中，我们就可以很简单的这样处理这个 HTML 了：

```
<h1>Hello world! (from hello.html file)</h1>
```

现在就可以在 tutorial.xml 界面定义中使用 **render-mode.text** 元素来明确包含这个 HTML 文件了：

```
<screen>
    <widgets>
      <label type="h1" text="Hello world!"/>
      <render-mode>
        <text type="html"
              location="component://tutorial/screen/tutorial/hello.html"/>
      </render-mode>
```

```
    </widgets>
</screen>
```

那么这个 render-mode 到底是什么东西？Moqui 框架的 XML 界面理念中存有**平台不可知论**以及界面可被渲染在各种环境中。源于此，我们不想界面内的任何东西在没有明确清楚的情况下就以某种确定渲染方式去展现。在 render-mode 下，你可以为不同的渲染模式定义各种子元素，甚至是各种文本模式，如 HTML，XML，XSL-FO（*用于格式化 XML 数据的语言，全称为 Extensible Stylesheet Language Formatting Objects 格式化对象的可扩展样式表语言，是 W3C 参考标准，现在通常叫做 XSL*），CSV 等。这样的话，一个界面定义就可以被渲染成不同的模式，并且输出的产品也可以按需进行切换。

这个界面还是按照前面例子的 URL 路径去访问，只是现在采用的是嵌入 HTML 页面内容的方式，而不是在界面定义中内联使用一个 label 元素的方式了。

再试界面子内容方式

另外种方式展现这个 hello.html 的内容就是把它作为界面的子内容。

这样的话，这个 hello.html 文件必须被放置在与这个界面同名的子目录下。例如，放在 tutorial.xml 文件同级的一个名为 tutorial 的文件夹下。

现在我们还需要做到如下几点：

- 设置界面的 **include-child-content** 属性为“true”，用来声明 tutorial.xml 界面包含了子内容
- 添加 **widgets** 下的 **subscreens-active** 元素，用以确定界面包含子界面或者子内容的位置

完成上述操作，你的界面的 XML 文件应该如下所示：

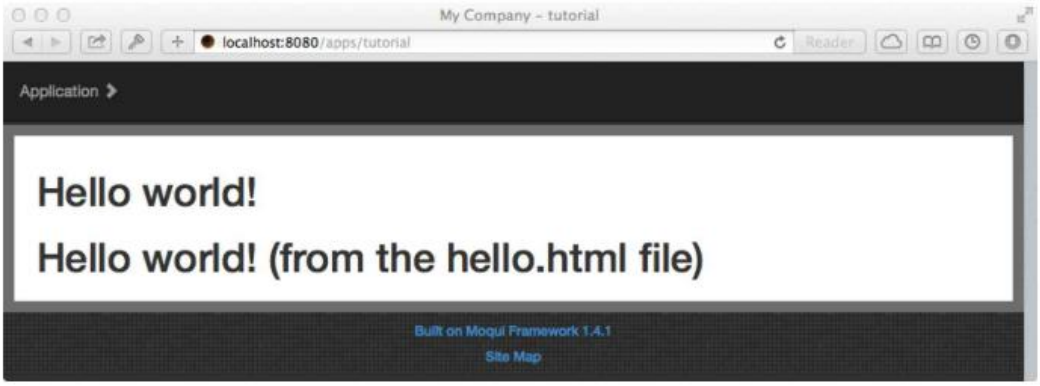
```
<screen include-child-content="true">
    <widgets>
        <label type="h1" text="Hello world!"/>
        <subscreens-active/>
    </widgets>
</screen>
```

为了能看到这个文件内容，你需要访问另外个 URL 路径来知会 Moqui 框架你需要获得 tutorial 界面下的 hello.html 文件：http://localhost:8080/apps/tutorial/hello.html

由于默认的子界面已被指定，你同样可以访问如下地址：

http://localhost:8080/apps/tutorial

下面是你的界面应该呈现的样子，有两行 hello world：



第二步

我的第一个实体

实体是一个基本的扁平列表的数据结构，并且通常为数据库中的一张表。一条实体值等价于数据库表的一行记录。Moqui 框架没有使用对象关系映射方式，所以我们要做的就是定义一个实体，然后通过实体门面模式（或者其他的高抽象层级的工具）去写代码操作使用实体。

我们创建一个包含 tutorialId 和 description 两个字段的实体 XML 文件，并命名为“Tutorial”，位于：runtime/component/tutorial/entity/TutorialEntities.xml

内容如下：

```
<entities>
  <entity entity-name="Tutorial" package-name="tutorial">
      <field name="tutorialId" type="id" is-pk="true"/>
      <field name="description" type="text-long"/>
  </entity>
</entities>
```

如果你以开发模式（***dev mode***）运行 Moqui 框架，实体定义缓存会自动被清理然后创建，所以你无需重启；但是你如果运行产品模式（***production mode***）或者你不想等待（Moqui 框架启动并不是很快）那么你就重启吧。

你什么时候去创建表呢？除非你关闭自动创建的特性（在 Moqui 的 XML 配置文件中有开关），否则实体门面模式将会在你第一次使用到这个实体的时候检查实体是否存在，不存在即会创建它。

添加一些数据

实体门面模式具有从 XML 文件中读取或者写入数据的功能，这些 XML 文件中的节点元素需要和实体名对应，同样的属性名和字段名需对应。

我们稍后将创建一个用户界面去录入数据，同时你也可以使用自动界面（Auto Screen）或者使用工具应用中的实体数据交互界面（Entity Data UI）和你新建的实体进行数据操作。

数据文件对于以下几种数据来说是很有用的：用于代码运行所需的种子数据、测试数据、验证/展示数据模型如何使用的验证数据。那么，我们就开始动手吧。

创建一个实体门面的 XML 文件在：runtime/component/tutorial/data/TutorialData.xml

内容为：

```
<entity-facade-xml type="seed">
    <Tutorial tutorialId="TestOne" description="Test one description."/>
    <Tutorial tutorialId="TestTwo" description="Test two description."/>
</entity-facade-xml>
```

加载数据只要通过“$ ant load”命令即可，或者在“运行 Moqui”章节中提及的其他加载方式也可。

自动查询表单

在 tutorial 界面下添加一个子界面的 XML 定义文件，并将其放到：

runtime/component/tutorial/screen/tutorial/FindTutorial.xml

```
<screen require-authentication="anonymous-view">
    <transition name="findTutorial">
        <default-response url="."/>
    </transition>
    <actions>
        <entity-find entity-name="Tutorial" list="tutorialList">
            <search-form-inputs/>
        </entity-find>
    </actions>
    <widgets>
        <form-list name="ListTutorials" list="tutorialList"
                transition="findTutorial">
            <auto-fields-entity entity-name="Tutorial"
                                field-type="find-display"/>
        </form-list>
    </widgets>
</screen>
```

这个界面中有几个关键部分：

- **转换 transition**

  转换/跳转指的是界面之间的链接。类似于一个顺序图一样，我们把每个界面当成一个个节点，那么界面中定义的转换/跳转就是“线”一般，从一个节点界面指向另外一个节点界面（或者自己指向自己），并且同时有些跳转还包含调用动作或者服务。

  - 单个转换可以按照条件或者不同的错误结果返回各种响应页面，一切都取决于你的界面设计需求
  - 个别转换会指向当前页面（*比如页面刷新就是这种场景*）

- **查询实体动作 actions.entity-find**

这里页面渲染时只有一个界面动作：查询实体（entity-find）。

- 通常使用 entity-find 元素（或者调用 Java API 使用 EntityFind 对象）时，你需要指定过滤条件，排序字段或者其他查询相关的信息去运行
- 在这个例子中，我们使用了 XML 表单内的标准参数去进行实体查询，所以我们可以使用 search-form-inputs 子元素去自动生成处理字段
- 如果想知道这些参数应该长啥样只要查看浏览器中的 HTML 代码即可，这些都是基于 XML 表单定义自动生成的

- **widgets.form-list**

  这个例子里面定义的是个真实的表单，指定的是多条记录/行数据的“列表”表单（相对于“单个”表单）

  - 这里的 **name** 属性可以为任意值，但是需要 XML 界面内唯一
  - 注意这里的 **list** 属性参照的是动作 actions 块中的 entity-find 返回的结果，**transition** 属性参照的是界面定义最上面的 transition 元素
  - 既然目标是自动创建一个基于实体定义的表单，我们便给 auto-fields-entity 元素赋值为我们的"Tutorial"实体，设置 **field-type** 属性的值为"find-display"选项，表示会创建查询字段在头部并且在表格中生成并显示每条记录。

使用 URL 路径：http://localhost:8080/apps/tutorial/FindTutorial 进行访问。

指定字段

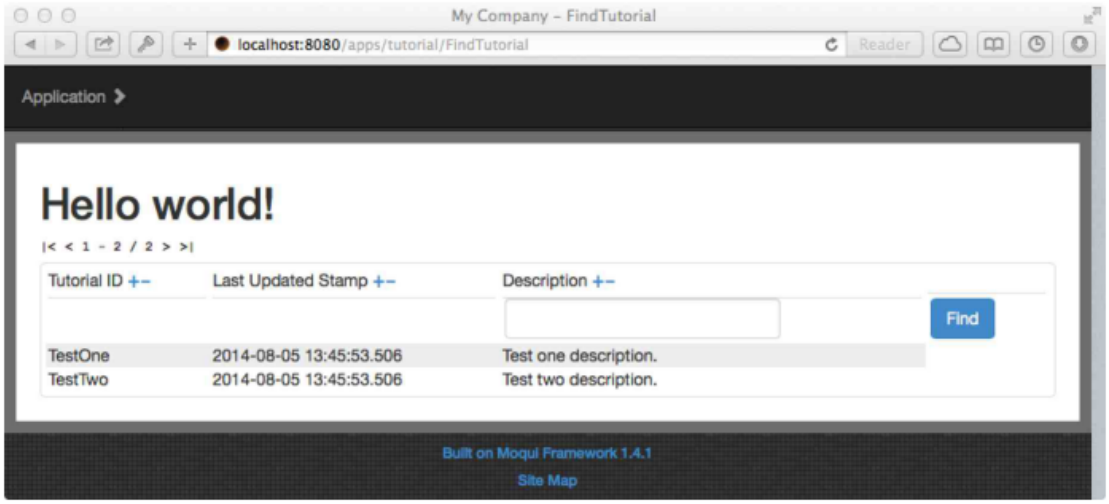
如果不是按照默认的方式去生成 description 字段，你如何按需指定它的展现方式呢？

要达到这个要求，你只要在 form-list 元素内添加一个 field 元素，并且跟在 fields-entity 元素之后，像这样：

```
<form-list name="ListTutorials" list="tutorialList"
        transition="findTutorial">
    <auto-fields-entity entity-name="Tutorial" field-type="find-display"/>
    <field name="description">
        <header-field show-order-by="true">
            <text-find hide-options="true"/>
        </header-field>
        <default-field>
            <display/>
        </default-field>
    </field>
</form-list>
```

由于此处在 field 元素中 **name** 属性的值对应的字段在 auto-fields-entity 元素执行时已被创建了，那么这里将会被重写。如果 **name** 值不在实体定义中，则会新生成一个额外的字段。这个结果看起来和 auto-fields-entity 元素自动处理的机制很像，并且这也是你需要清晰明白的。

当你的界面和表单按照 FindTutorial 界面定义完成，你的页面展示应该如下所示：



添加一个新建表单

让我们添加一个按钮来弹出一个新建表单，并创建一个转换来处理输入数据操作。

首先在之前创建的界面 FindTutorial.xml 中添加一个转换，就跟在 findTutorial 转换之后：

```
<transition name="createTutorial">
    <service-call name="create#Tutorial" />
    <default-response url="." />
</transition>
```

这个转换只是调用了 create#Tutorial 服务，然后跳转回了当前界面。

这个 create#Tutorial 服务从哪里来的呢？我们还没定义任何它的实现内容。Moqui 框架的服务门面支持一种特殊的无需定义实现的实体增删改查（CrUD）操作的服务。这个服务的名字由两部分组成：一个动词和一个名词，中间用#隔开。只要动词为 **create**, **update**, **store**, 或者 **delete**，同时名词是一个有效的实体名称，服务门面就会认为这个服务是个隐式的自动实体服务并会完成预期的操作。这个服务调用是完全基于实体定义和传递的参数。举个例子，如果你使用 create 动词并传递实体的一个主键字段，服务就会使用这个主键字段，否则服务将会使用实体名称作为序列关键字自动生成主键序列号。

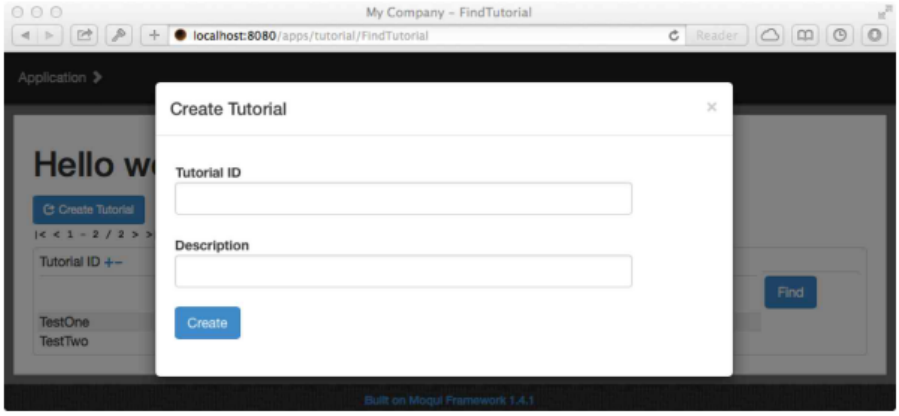
然后我们来添加一个新增表单，当按钮点击时展现一个隐藏的容器界面。在之前创建的 FindTutorial 界面中，添加容器到 widget 元素内的 form-list 元素上面，这样就会在列表表单上展现这个容器：

```
<container-dialog id="CreateTutorialDialog" button-text="Create Tutorial">
    <form-single name="CreateTutorial" transition="createTutorial">
        <auto-fields-entity entity-name="Tutorial" field-type="edit"/>
        <field name="submitButton">
            <default-field title="Create">
                <submit/>
            </default-field>
        </field>
    </form-single>
```

```
</container-dialog>
```

这个界面定义通过刚才添加的 transition 来引用，并且使用 auto-fields-entity 元素中值为"edit"的 **field-type** 元素定义来自动生成编辑字段。最后一个小细节是声明一个按钮去提交表单，这样就可以运行了。尝试着做一下，然后看看原生界面的列表的数据。

这里有一个弹出的新建表单，并且你可以看到背景上添加在查询界面中的按钮：



第三步

自定义新增服务

上面的例子里，createTutorial 转换使用了隐式的自动实体服务 create#Tutorial。我们来看下如何手动的定义和实现一个服务。

首先定义一个服务使用默认的自动实体增删改查（CrUD）实现。将服务定义 XML 文件放在下面的位置：

runtime/component/tutorial/service/tutorial/TutorialServices.xml

内容如下：

```
<services>
    <service verb="create" noun="Tutorial" type="entity-auto">
        <in-parameters>
            <auto-parameters include="all"/>
        </in-parameters>
        <out-parameters>
            <auto-parameters include="pk" required="true"/>
        </out-parameters>
    </service>
</services>
```

这个服务将允许 Tutorial 实体的所有字段传入，并总是返回主键字段（tutorialId）。这个服务定义中我们使用了基于实体的 auto-parameters 元素，这样如果我们新增了实体字段的话，它们将自动会体现在这个服务处理中。

现在，我们修改下这个服务定义来添加一个内联的实现。注意下服务中的 **type** 属性已

经发生了变化，并且新增了一个 actions 元素。

```
<service verb="create" noun="Tutorial" type="inline">
    <in-parameters>
        <auto-parameters include="all"/>
    </in-parameters>
    <out-parameters>
        <auto-parameters include="pk" required="true"/>
    </out-parameters>
    <actions>
        <entity-make-value entity-name="Tutorial" value-field="tutorial"/>
        <entity-set value-field="tutorial" include="all"/>
        <if condition="!tutorial.tutorialId">
            <entity-sequenced-id-primary value-field="tutorial"/>
        </if>
        <entity-create value-field="tutorial"/>
    </actions>
</service>
```

现在只需改变 transition 指向这个服务既可以调用了：

```
<transition name="createTutorial">
    <service-call name="tutorial.TutorialServices.create#Tutorial"/>
    <default-response url="."/>
</transition>
```

注意下这个服务名称定义很像一个标准的 Java class 的命名。它有一个“包名”，本例中是在 component/service 目录下的"tutorial"目录（也许是以点号分割的多重目录结构）的目录名。然后是一个点号以及类名的等价物，本例中“TutorialServices”就是服务的 XML 文件的名称，但是没有.xml 后缀。这之后又是一个点号，然后是服务的动词和名词操作，以#符号分割。

Groovy 服务

如果你想使用 Groovy（或者框架支持的其他脚本语言）而不用内联的 XML 动作去实现服务，怎么做呢？这种情况服务定义像这样：

```
<service verb="create" noun="Tutorial" type="script"
        location="component://tutorial/script/tutorial/createTutorial.groovy">
    <in-parameters>
        <auto-parameters include="all"/>
    </in-parameters>
    <out-parameters>
        <auto-parameters include="pk" required="true"/>
    </out-parameters>
```

```
</service>
```

注意这里服务的 **type** 属性已经变为"script"，并且现在有个 **location** 属性去指定脚本的位置。

下面是这个 Groovy 脚本的内容：

```
EntityValue tutorial = ec.entity.makeValue("Tutorial")
tutorial.setAll(context)
if (!tutorial.tutorialId) tutorial.setSequencedIdPrimary()
tutorial.create()
```

当你使用 Groovy 或者其他语言时，你将会用到基于可执行上下文（ExecutionContext class）这个类去调用 Moqui 的 Java API,通常这个类在脚本里会用**"ec"**这个名称。更多的 API 详情请看 Java API 文档，并查询文档中 ExecutionContext 这个类，这个类关联了很多核心的 API 接口。

五、 数据和资源

资源，内容，模板及脚本

资源位置

资源门面位置的字符串类似于 URL 的构成方式：协议，主机，可选端口和文件名。它支持标准的 Java URL 协议（(http, https, ftp, jar 以及 file）。它同样也支持一些扩展的协议：

- **classpath://** java 类路径的资源
- **content://** 内容仓库的资源（JCR, 经由 Jackrabbit 客户端）；路径中协议前缀后的第一个元素是内容仓库的名称，其在 Moqui 的 XML 配置文件中被 repository.name 属性指定
- **component://** 相对于组件所在位置的路径，无论组件位于何处（文件系统，内容仓库等）
- **dbresource://** 用于一个虚拟的文件系统，通过 moqui.resource.DbResource 和 DbResourceFile 实体在数据库中持久化实体门面

其他的协议可以通过实现 org.moqui.context.ResourceReference 接口，并添加一个 resource-facade.resource-reference 元素到 Moqui XML 配置文件中来完成。上述已支持的协议按照这种方式配置在 MoquiDefaultConf.xml 文件中。

使用资源

最简单的方式去按照所有已支持的协议使用一个资源，就是直接去读取文本或者二进制内容。从一个资源位置获得文本使用 ec.resource.**getLocationText**(String location, boolean cache) 方法。获取二进制输入流或者大文本资源使用 ec.resource.**getLocationStream**(String location) 方法。

除了直接读取资源数据，还有相当多的操作可以通过使用 ec.resource.**getLocationReference**(String location)方法获得 org.moqui.context.ResourceReference 接口的一个实例。这个接口类似于资源门面方法一样，有从资源中获取文本和二进制流数据的一些方法。接口还有一些方法用于目录资源获得下级资源，按照名称递归查找所有下级文件或目录，写文本或者二进制流数据以及移动资源到其他的位置。

模板渲染和运行脚本

这里只有单独的一个方法用于渲染某个资源路径下的模板：ec.resource.**renderTemplateInCurrentContext**(String location, Writer writer)。这个方法不返回任何东西，只是简单的输出模板到 writer。默认支持 FTL（Freemarker Template Language）和 Gstring（Groovy String）模板。

其他的模板可以通过实现 org.moqui.context.TemplateRenderer 接口，并在 Moqui XML 配置文件中添加 resource-facade.template-renderer 元素来完成。

通过资源门面运行脚本使用 Object ec.resource.**runScriptInCurrentContext**(String location, String method)方法。脚本的位置以及脚本中要运行的方法作为指定的入参，将会返回脚本对应的方法返回值或者计算结果的 Object 对象。在资源门面中这个方法还有种变异的用法，它可以方便的接受一个 Map 类型的 **additionalContext** 参数（机制是将这个 Map 压入上下文堆栈中，运行脚本，然后将其弹出上下文堆栈）。通过 javax.script.ScriptEngineManager，Moqui 框架默认支持 Groovy，XML 动作，JavaScript 以及任意脚本引擎支持的脚本。

你有两种选择去添加一个脚本执行器。你可以使用 javax.script 的方式为任何脚本语言去实现 javax.script.ScriptEngine 接口，并可以通过 javax.script.ScriptEngineManager 去发现脚本。Moqui 框架就使用这种扩展方式，基于脚本的文件名去发现脚本引擎并执行脚本。如果脚本引擎实现的是 javax.script.Compilable 接口，那么 Moqui 框架将会编译脚本，并为了在一个给定的位置更快速的反复执行一个脚本，以编译的形式缓存脚本。

另外一个选择是实现 org.moqui.context.ScriptRunner 接口，并在 Moqui XML 配置文件中添加一个 resource-facade.script-runner 元素节点。Moqui 框架通过这个接口使用 Groovy 的 XML 操作是因为它提供了额外的便利性，但通过 javax.script 接口无法使用。

由于 Groovy 在 Moqui 框架中是默认的表达式语言，这里有一些资源门面的方法可用于简单的不同目的的表达式计算：

- boolean **evaluateCondition**(String expression, String debugLocation)　用于计算一个 Groovy 条件表达式并返回布尔结果
- Object **evaluateContextField**(String expression, String debugLocation)　用于计算表达式返回上下文中的一个字段，通常更多的用于计算任何 Groovy 表达式并返回结果
- String **evaluateStringExpand**(String inputString, String debugLocation)　用于扩展输入的字符串，将其当成一个 GString（Groovy String）并返回扩展后的字符串

这些方法都接收了一个 debugLocation 参数用于处理错误信息。为了更快的计算，这些表达式都被缓存起来了，表达式本身就是个极大的重用的关键。

数据模型定义

实体 XML 定义

让我们从一个简单的实体定义开始，展现最常用的元素。这个真实的实体是 Moqui 框架中的一部分：

```
<entity entity-name="DataSource" package-name="moqui.basic" cache="true">
    <field name="dataSourceId" type="id" is-pk="true"/>
    <field name="dataSourceTypeEnumId" type="id"/>
    <field name="description" type="text-medium"/>
    <relationship type="one" title="DataSourceType"
            related-entity-name="Enumeration">
        <key-map field-name="dataSourceTypeEnumId"/>
    </relationship>
    <seed-data>
        <moqui.basic.EnumerationType description="Data Source Type"
            enumTypeId="DataSourceType"/>
        <moqui.basic.Enumeration description="Purchased Data"
            enumId="DST_PURCHASED_DATA" enumTypeId="DataSourceType"/>
    </seed-data>
</entity>
```

就像一个 Java 类，实体有一个包名，并且实体的全名就是包名加上实体名，按照这种格式：${package-name}.${entity-name}

基于这种模式，实体的全名就是：moqui.basic.DataSource

这个例子还有个 entity.**cache** 属性设置为 true，这意味着实体将会被缓存，除非代码另外指定去查找。

第一个字段（**dataSourceId**）的 **is-pk** 属性设置为 true，意味着这个字段是实体的其中一个主键字段。本例的这种情况下只有一个主键，但是其他任意字段都可以设置这个属性为 true 去使它们成为主键的一部分。

第三个字段（**description**）是一个用于保存数据的简单字段。它不是主键的一部分，并且也不是其他实体的外键。

field.**type** 属性用于指定字段的数据类型。MoquiDefaultConf.xml 文件中通过 database-list.dictionary-type 元素定义了默认的选项。这些元素为每个字典类型都指定了默认的类型设置，并且可以使用 database.database-type 元素为每种数据库去重新定义覆盖类型的设置。

你可以使用这些元素在数据类型字典中添加你自己的类型。这些自定义的类型不会在你的 XML 编辑器中为 field.**type** 属性出现自动完成提示，除非你修改 XSD 文件并同样添加上模式定义，这样就可以很好的运行了。

第二个字段（**dataSourceTypeEnumId**）是枚举（Enumeration）实体的外键，通过实体定义中的 relationship 元素来表示。seed-data 元素下的两条记录里定义了枚举类型（EnumerationType）去组成枚举（Enumeration）选项，并且其中一个枚举选项是为 **dataSourceTypeEnumId** 字段配置的。seed-data 元素下的记录通过 seed 类型使用命令行的-**load** 操作（或者相应的 API 调用）进行加载。

这里有个很重要的模式，允许框架知道使用哪个枚举类型 **enumTypeId** 来在自动生成的表单中过滤一个字段的枚举 Enumeration 选项。注意 relationship.**title** 属性的值匹配 enumTypeId。换句话说，无论如何对于枚举来说这里有个约定：relationship.**title** 的值就是类型的 ID 用来过滤下拉列表。

这种模式在 Moqui 框架和地幔业务构件中被广泛使用，这是因为枚举 Enumeration 实体被用于管理很多不同实体可用的类型。

本例中有一个 key-map 元素位于 relationship 元素下面，这仅对于这个实体的字段名称和相应的关联实体的字段名称不匹配才需要（进行手动映射）。换句话说，由于外键字段名字是 **dataSourceTypeEnumId** 而不是简单的 **enumId**，所以我们需要告诉框架使用哪个字段去关联。框架知道关联实体（本例中的 Enumeration）的主键字段，但除非字段的名称匹配主键字段的名称，否则框架不知道实体的哪些字段对应哪些字段。

很多情况下，你可以不使用 key-map 元素，简单的如下使用：

```
<relationship type="one"   related-entity-name="Enumeration"/>
```

seed-data 元素允许你定义使用实体时必要的一些基本数据，并且这也是定义数据模型的一个方面。这些记录通过在 entity-facade-xml 文件中将 **type** 属性设置为 seed，来被加载到数据库中。

随着对一个实体定义中大部分常用元素的介绍，我们现在来看一下实体定义中其他一些可用的元素和属性。

- ***其他实体属性***
  - **group-name:** 实体门面（Entity Facade）用来将一个实体为某个数据库放入分组中，所有数据库都可用。这个值应当匹配 Moqui XML 配置文件中的 moqui-conf.entity-facade.datasource.**group-name** 元素。如果没有值被指定，那么将默认使用 moqui-conf.entity-facade.**default-group-name** 元素的值。配置默认可用的值包括：transactional (默认的)， analytical，tenantcommon 以及 nosql。

- **sequence-bank-size:** 保存在内存中的序列储库的大小。每当在 SequenceValueItem 记录中的序列号 **seqNum** 被内存中的储库消耗完了，将按照这个数量进行增加。
- **sequence-primary-stagger:** 最大数量的交错就是序列 ID。如果增长序列为 1 则会增加 1，此外，当前的序列号 ID 还可以按照一个 1 和最大间隔之间的随机数来进行增长。
- **sequence-secondary-padded-length:** 如果指定了前垫部分，后面都以零进行填充到本属性指定的长度，默认长度为 2。
- **optimistic-lock:** 当设置为 true，实体门面（Entity Facade）在更新记录时，会比较在内存中的 **lastUpdatedStamp** 字段和数据库中相同的字段。如果时间戳不相匹配那么会产生一个错误。默认为" false"（无时间戳锁机制）。
- **no-update-stamp:** 实体门面（Entity Facade）默认的为每个实体添加一个字段（**lastUpdatedStamp**）用于乐观锁机制以及数据同步。如果你不想要为实体创建这个标记字段这可以设置这个属性为" false" 。
- **cache:** 能设置以下值（默认为 false）：
  - true: 使用缓存查找（代码可覆盖）
  - false: 不用缓存查找（代码可覆盖）
  - never: 不用缓存查找（代码不可覆盖）
- **authorize-skip:** 能设置以下值（默认为 false）：
  - true: 略过这个实体所有的权限校验
  - false: 不略过权限校验
  - create: 为新建操作略过权限校验
  - view: 为查询和只读操作略过权限校验
  - view-create: 为查询和新建操作略过权限校验

其他字段属性

- **encrypt:** 当设置为 true 时，在数据库中加密该字段。默认为 false（不加密）。
- **enable-audit-log:** 设置为 true 时，将日志记录本字段的所有变化以及变化时间和修改人。日志数据记录在 EntityAuditLog 实体中。默认为 false（不审计记录）。
- **enable-localization:** 当该属性被设置为 true，获取本字段值时将会同时查看 LocalizedEntityField 实体，如果其中有匹配的记录，那么将返回本地化的记录而不是原始记录。由于展现的原因默认为 false，只有当字段属性设置为 true 时才会进行翻译。

然而有些数据库优化必须在数据库层面去做，因为不同数据库之间的特性差异如此之大，你可以使用实体定义的 index 元素去声明索引。作为 entity 实体元素之下的元素，它应该如下所示：

```
<index name="EX_NAME_IDX1" unique="true">
    <index-field name="exampleName"/>
</index>
```

实体扩展-XML

实体可以无需修改原来的实体定义 XML 文件来进行扩展。尤其当你想在不同的组件中扩展例如地幔通用数据模型（mantle-udm）或甚至 Moqui 框架中的实体的一部分，并且保持你的扩展部分相对独立，这种方式就特别的有用。

你可以在实体定义 XML 文件中使用 extend-entity 元素并可以混合使用 entity 元素去达到效果。这个元素拥有和 entity 元素很多相同的属性去定义扩展原来的实体。简单的保证 **entity-name** 以及 **package-name** 属性和原来的 entity 实体元素的属性一致，那么你指定的任何内容都会添加或者覆盖原有实体的定义。

```
<extend-entity entity-name="Example" package-name="moqui.example">
    <field name="auditedField" type="text-medium" enable-audit-log="true"/>
    <field name="encryptedField" type="text-medium" encrypt="true"/>
</extend-entity>
```

实体扩展-DB

你还可以使用 UserField 实体通过数据库记录来扩展实体。这种方式和通过 extend-entity XML 元素扩展实体有些不同，因为这种方式是一种虚拟的扩展，并且数据存入了独立的 UserFieldValue 实体数据结构中。

这种差异的主要原因就是通常添加的用户字段是为了一组用户还是单个用户，以及除了本组用户，关联的外部用户无法访问。你可以使用 ALL_USERS 用户组来使一个用户字段作用于所有的用户。

尽管它运行在后台，但是 EntityValue 对象用实体对待其他任何字段一样的方式去处理这些字段。

这里有一个源于 ExampleTypeData.xml 文件的例子，去展示你如何为 moqui.example.Example 实体添加一个所有用户可访问的 **testUserField** 字段：

```
<moqui.entity.UserField entityName="moqui.example.Example"
    fieldName="testUserField"  userGroupId="ALL_USERS"  fieldType="text-long"
    enableAuditLog="Y"  enableLocalization="N"  encrypt="N"/>
```

数据模型模式

Moqui 框架提供了很多有用的约定和功能的数据模型模式。这些数据模型模式同样广泛的用于 Moqui 和 Mantle 的数据模型中。

主实体

主实体是独立于其他实体的，并通常只有单一主键字段。例子包括 moqui.example.Example，moqui.security.UserAccount，mantle.party.Party，mantle.product.Product 以及 mantle.order.OrderHeader 实体。

为设置区分主实体主键序列值的主键 ID，可以使用 EntityValue.**setSequencedIdPrimary()** 方法。你同样可以手动的设置主键字段的值，只要是唯一的。

明细实体

明细实体是为了主实体的字段添加明细信息并和主实体有个一对多的关联关系。明细实体的主键通常是两个字段，并且其中一个是主实体的单一主键字段。第二个字段是一个特殊的排序序列 ID，而不是一个绝对的序列值，它存在于主实体主键的上下文中。

一个明细实体的例子就是 ExampleItem，它是 Example 主实体的明细。ExampleItem 有两个主键：**exampleId**（主实体的主键字段）以及 **exampleItemSeqId**，它是一个子序列用以在主实体上下文中区分明细记录。

填充第二个序列 ID 首先要填充主实体的主键值（ExampleItem 的 **exampleId** 字段），然后使用 EntityValue.**setSequencedIdSecondary()** 方法去自动填充它（为 ExampleItem 的 **exampleItemSeqId** 字段）。

单一的主实体可以按照需求，有多个关联的明细实体去构成不同的数据结构。

关联实体

关联实体用于主实体之间的关联，通常为两个主实体。关联实体是一个用于表示实体间多对多关系逻辑模型的物理呈现。

关联实体追踪实体之间的关联记录十分的有用，并且其中任意数据都是两个实体之间的关系而不仅仅只是其中之一。举个例子，如果你想要在另外一个主实体上下文的记录中去指定一个主实体记录的序列号，那么这个序列号字段应该存在关联实体上而不是其中任意的主实体上。

关联实体可能有单一的主键生成，或者一个自然的由每个主实体的唯一主键字段构成的复合主键，并且还有可选的 **fromDate** 和 **thruDate** 字段。

ExampleFeatureAppl 实体就是一个例子，用来关联 Example 和 ExampleFeature 主实体。ExampleFeatureAppl 实体有三个主键字段：**exampleId**（Example 实体的主键），**exampleFeatureId**（ExampleFeature 实体的主键）以及 **fromDate**。它还有一个伴随 **fromDate** 主键字段的 **thruDate** 字段。

为了更好的描述 Example 和 ExampleFeature 之间的关系，ExampleFeatureAppl 实体还有一个 **sequenceNum** 字段用于订单的特性来举例，**exampleFeatureApplEnumId** 字段用于描述应用于例子的特性是什么（必须的，渴望的或者不允许的）。

你可以通过翻看 ExampleEntities.xml（Moqui 框架中的 example 例子组件）文件来查阅实际的 ExampleFeatureAppl 实体定义以及种子数据。

依赖实体

API 和工具应用的一部分支持“依赖”实体的概念。任何实体都会发现有依赖实体，但是这个概念对于主实体的依赖来说最为重要。一般的想法是订单的条目（mantle.order.OrderItem）依赖于订单头（mantle.order.OrderHeader）。这对于类似包括主实体和所有依赖实体的数据导出来说很有用。

这个概念上十分简单，但是实现上稍微复杂点，因为我们必须处理依赖实体的关系信息。通常的想法是每种类型的一个关系从一个依赖实体指向到其主实体，通过这种定义方式，很多依赖实体都有不止一个主实体，同时一个实体既可以是依赖实体也可以是主实体，所以一

个实体是什么类型取决于你怎么看待它。当定义实体时，这里都会为每种类型的一个关系自动生成一个反向类型的关系。并且虽然它通常是一个许多反向关系，但是如果两个实体有相同的主键字段，那么这是一个类型一个自动反向关系。

举个例子，OrderItem 有一个类型一个关系关联到 OrderHeader，所以这里有一个自动反向关系的类型从 OrderHeader 到 OrderItem。这就建立起 OrderItem 作为 OrderHeader 的一个依赖。

当获得实体的依赖时，方法（是内部实体门面 Entity Façade 实现的部分：EntityDefinition.**getDependentsTree()**）同样递归的去执行获取依赖的依赖。通常的想法就是类似订单头 OrderHeader 等实体，你可以获得所有的订单定义的记录。

枚举

枚举是一个简单预配置的可能的数据集合。枚举用于描述单个记录或者记录间的关系。一个实体可以有多个枚举字段。

Moqui 框架中存储所有枚举值的实体名为 Enumeration，并且其中的值按照 EnumerationType 实体中的类别记录进行隔离。

当一个字段有一组可能的枚举值时，它应该有“EnumId”后缀，就像 Example 实体上的 **exampleTypeEnumId** 字段。对于这里的每个字段都应该有一个关系元素去描述从当前实体到 Enumeration 实体的关系。relationship 元素上的 **title** 属性应当和 Enumeration 记录中的 **enumTypeId** 字段有相同的值。通常 **title** 属性应当和“EnumId”后缀的枚举字段名相同。例如 **exampleTypeEnumId** 字段的关系标题是 ExampleType。

状态，流，转换和历史

另一个有用的数据概念是追踪一个记录状态。各种业务概念都有一个一组可以被容易追踪，可能的状态值的生命周期。使用 StatusItem 实体去追踪这些可能的状态值并且使用 StatusType 实体中的 **statusTypeId** 字段去区分指向这些状态记录。

一组状态值就像图中的节点，节点间的状态变化就表现为从一个状态变为另外一个状态。我们使用 StatusFlowTransition 实体去记录从一个状态到另一个状态可能的变化配置。

对于指定 **statusTypeId** 字段的一组状态项可能有多个状态流，每个流表现为一条 StatusFlow 记录。StatusItem 记录使用 StatusFlowItem 实体和 StatusFlow 实体进行关联。举个例子，工作投入 WorkEffort 实体有一个 **statusFlowId** 字段去指定哪一个状态流应当被一个项目或者任务使用。

StatusItem 实体中定义了每个当前实体与状态字段的关联关系。类似 Enumeration 实体的模式，relationship 元素上的标题 **title** 属性应当和 StatusItem 记录的 **statusTypeId** 字段相匹配。

实体门面（Entity Facade）的审计日志是保持状态记录历史最简单的方式，包括谁做了变更，变更的时间以及老的和新的状态值。开启这个功能只需设置 entity.field 元素的 **enable-audit-log** 属性为 true。字段的定义方式如下所示：

```
<field name="statusId"  type="id"  enable-audit-log="true"/>
```

计量单位（UOM）

计量单位是一个标准化或者自定义的计量单位，例如长度，重量，温度，数据量甚至币种。这些都是计量单位的类型。一个 moqui.basic.Uom 记录通过 **uomId** 字段标识，有类型（**uomTypeEnumId**），描述（**description**）以及缩写（**abbreviation**）字段。开箱即用的计量单位数据存在 UnitData.xml 文件中。

大部分的计量单位类型在同类型的不同单位间有个转换。这些转换定义在 UomConversion 实体中。举个例子，1000 米等于一千米，记录定义如下：

```
<moqui.basic.UomConversion  uomConversionId="LEN_km_m"  uomId="LEN_km"
    toUomId="LEN_m"  conversionFactor="1000"/>
```

conversionFactor 乘以 **uomId** 单位的值获得了 **toUomId** 单位的值。你同样可以反向转换去做除法。举个例子，1km = 1000m 所以 LEN_km 单位的值 1 是由 1000 个 **conversionFactor** 分割 1000 个 LEN_m 单位获得的值。

这里同样有个 **conversionOffset** 字段用于类似于摄氏度和华氏温标温度的场景，用于从一个单位增加（或者减少）某个值转换到另一个单位。**conversionFactor** 先做乘法，然后结果加上 **conversionOffset** 的值。当作反向的转换时，**conversionOffset** 先做减法，然后 **conversionFactor** 做除法。

某些 UOM 类型，例如币种的转换因素随着时间变化的。为了处理这种情况，UomConversion 实体有可选的有效时间（**fromDate, thruDate**）字段。

地理边界和关键点

地理边界可以是一个政治地区，商业地区或者其他任何地理区域。每个 moqui.basic.Geo 记录，通过 **geoId** 字段标识，有一个类型（**geoTypeEnumId**）字段如城市，国家或销售区域。每个 Geo 记录都有一个 ISO 3166 模式规范的名称（**geoName**）以及 2 个字母（**geoCodeAlpha2**），3 个字母（**geoCodeAlpha3**）和数字（**geoCodeNumeric**）编码的国家编码（参见 Moqui 框架自带的 GeoCountryData.xml 文件中的国家数据）。

Geo 实体还有个 **wellKnownText** 字段用于机器可读的地理边界的几何明细信息。这意味着其包含了各种数据库和工具（包含 Java 库）支持的 ISO/IEC 13249-3:2011 规范的文本。对于 WKT 的详细介绍参见：http://en.wikipedia.org/wiki/Well-known_text

我们可以使用 GeoAssoc 实体去关联 Geo 记录。它有不同的类型（**geoAssocTypeEnumId**）包含了可用于区域较大的地理边界（GAT_REGIONS;例如州里的城市，国家中的州），通过通常的组关联的 Geo 记录（GAT_GROUP_MEMBER;类似美国本土的 48 州），或者你可能定义的其他类型。**geoId** 字段应该指向组或者大的区域，并且 **toGeoId** 字段指向这个区域的组成员或者地区。查看 GeoUsaData.xml 文件关于这两个字段的例子。

GeoPoint 是一个特殊的地理点，例如地球表面的一个点。它有纬度（**latitude**）,经度（**longitude**）和 海拔（**elevation**）字段，以及 **elevationUomId** 字段用于指定海拔的类型（例如英尺，用 LEN_ft 标识）。这里还有 **dataSourceId** 字段用于指定数据来源，**information** 字段用于地理点的信息文本记录。

实体门面模式

基本的 CrUD 操作

一个实体记录可用的基本 CrUD 操作通过 EntityValue 接口实现。这里有两个主要的方式去获得一个 EntityValue 对象：

- Make a value（ 使用 ec.entity.makeValue(entityName) ）
- Find a value（详见下面的细节）

一旦你有一个 EntityValue 对象，你就可以调用 **create()**，**update()** 或者 **delete()**方法去执行所需的操作。这里同样还有个 **createOrUpdate()**方法用于不存在时新建记录，存在时更新记录。

注意这些方法和 EntityValue 接口的很多方法一样，都返回一个自参照（self-reference）便于你可以使用链式操作。如下例所示：

```
ec.entity.makeValue("Example").setAll(fields)
        .setSequencedIdPrimary().create()
```

尽管这个例子很有意思，但是只有在很罕见的情况下你才需要直接使用实体门面的 API（类似 ec.entity 去存取数据）去新建一条记录。你一般是通过服务去执行 CrUD 操作，并且通过服务门面对每个实体都有可用的自动的 CrUD 服务。这些服务没有定义，它们都隐式存在并被实体定义所驱动。

我们将在下面的业务逻辑层部分详细讨论服务门面，但这里有一个使用隐式自动实体服务操作的例子：

```
ec.service.sync().name("create#Example").parameters(fields).call()
```

很多 Moqui 框架的 API 方法都返回一个自参照便于类似这种链式调用方式。但这两种的主要区别是一个通过服务门面另一个不是。通过服务门面方式有些优点（例如事务管理，流程控制，安全选项等等），但是在底层操作执行之前，很多事情包括自动清理和字段类型转换等使用这两种调用方式是一样的。

同样注意一下，通过隐式的自动实体服务你无需明确的设置主键序列 ID，因为它会自动的判定这里有个主键并且如果服务参数中没有指定那么它将会生成一个。

无论你做什么操作，只有改变的或者传递的实体字段才会被更新。EntityValue 对象将会保持追踪这些变化的字段直到数据库中的新增或者更新完成。你可以使用 **isModified**()方法来请求获知 EntityValue 对象是否被修改，并且你可以使用 **refresh**()方法来重载其数据库中的状态（设置所有的字段，而不仅仅是修改的那些字段）。

如果你想要找到当前 EntityValue 对象中字段和数据库中相应列字段值的差别，使用 **checkAgainstDatabase**（List messages）方法。这个方法用于断言（而不是加载）一个 entity-facade-xml 文件，并同样可以用于手动去编写 Java 或者 Groovy 代码检查数据的状态。

查找实体记录

查找实体的记录可以使用 EntityFind 接口。你可以通过 EntityFind 接口调用查询方法的不同方面来关注你需要的，从而忽略其他的，而不是使用一堆不同参数的不同方法。你可以

如下所示从实体门面 EntityFacade 中获得一个查询对象：

```
ec.getEntity().makeFind("moqui.example.Example")
```

EntityFind 接口的大部分方法返回一个实体的参照，这样你可以使用链式的调用方法而无需分成多个语句调用。例如在 Example 实体上通过主键进行查找的例子如下所示：

```
EntityValue example = ec.entity.makeFind("moqui.example.Example")
        .condition("exampleId", exampleId).useCache(true).one()
```

EntityFind 接口的方法选项有：

- ***条件（where 和 having 的条件）***
  - **condition**(String **fieldName**, Object **value**)： 简单条件，判断指定的字段值是否等于传入的参数值
  - **condition**(String **fieldName**, EntityCondition.ComparisonOperator **operator**, Object **value**)： 比较指定的字段值和传入参数值之间的大小，比较操作符可以是 EQUALS, NOT_EQUAL, LESS_THAN, GREATER_THAN, LESS_THAN_EQUAL_TO, GREATER_THAN_EQUAL_TO, IN, NOT_IN, BETWEEN, LIKE 或者 NOT_LIKE
  - **conditionToField**(String **fieldName**, EntityCondition.ComparisonOperator **operator**, String **toFieldName**)： 使用操作符比较一个字段和另外个字段的值
  - **condition**(Map<String, **?**> **fields**)： 约定每个映射 Map 条目的键值都必须匹配实体的字段名。如果一个字段被设置为和 Map 中键相同的名字，那么这个字段的值将被相应的 Map 值替换。在查询前，字段按照这种方式进行设置并和其他条件（如果适用的话）进行结合。如果有必要将字符串转换为要求的字段类型，这里有一个转换并且只将获得匹配实体字段的键。换句话说，它和 EntityValue.**setFields**(fields, true, null, null)方法做同样的事情。
  - **condition**(EntityCondition **condition**)： 通过 EntityConditionFactory 添加一个条件。
  - **conditionDate**(String **fromFieldName**, String **thruFieldName**, Timestamp **compareStamp**)： 为标准的有效日期查询模式添加条件，包括起始字段为空或者早于等于 compareStamp 比较时间戳，以及截止时间为空或者晚于等于 compareStamp 比较时间戳
  - **havingCondition**(EntityCondition **condition**)： 通过 EntityConditionFactory 添加 having 条件。Having 是标准 SQL 的概念并用于分组和函数之后。
  - **searchFormInputs**(String **inputFieldsMapName**, String **defaultOrderBy**, boolean **alwaysPaginate**)： 为 inputFieldsMapName 标识的 Map 中的字段添加条件。这些字段指定的方式和 XML 表单中支持的*-find 后缀结尾的字段一样。这意味着你可以通过这种方式去处理 XML 表单中生成的各种输入数据。字段后缀包括像*_op 代表操作符，*_ic 代表忽略项。如果 inputFieldsMapName 为空，将会查看 ec.web.parameters map 值去检查 web 门面是否可用，否则查看当前上下文（ec.context）。如果这里没有 orderByField 参数（查询 XML 表单的一个标准参数），则将使用 defaultOrderBy 参数替代。如果 alwaysPaginate 设置为 true，那么页码偏移量/每页限制将会被设置使用，甚至这里没有 pageIndex 参数。

- ***通过 selectField(String fieldToSelect)和 selectFields(Collection<String> fieldsToSelect)进行字段的选择***

- ***结果的字段排序***
  - **orderBy**(String **orderByFieldName**)： 查询实体的排序字段。可选择在最后添加”ASC”或者前面添加“+”进行升序，或者最后添加“DESC”或者前面添加”-”来进行降序。如果还有其他

的字段排序被指定，将会被添加到列表的最后。这个 String 字符串可能是一个逗号分隔的字段名列表。只有实体中实际存在的字段才可以被添加到这个排序列表中。

- **orderBy(List<String> orderByFieldNames)：** 每个 List 条目都调用一次 orderBy(String orderByFieldName)方法。

- ***使用 useCache(Boolean useCache)方法决定是否缓存结果，默认按照实体定义的值***

- ***传递到数据库的偏移量和限制参数，用以限制查询结果***
  - **offset(Integer offset)：** 偏移量，例如返回的起始行。默认（空）意味着真实的第一行作为起点。仅用于 list()和 iterator()方法的查询。
  - **offset(int pageIndex, int pageSize)：** 指定页码和每页记录数。实际的偏移量就是页码*每页记录数。
  - **limit(Integer limit)：** 界线，例如返回的最大记录行数。默认（空）为所有行。仅用于 list()和 iterator()方法的查询。

- ***数据库选项包括使用 distinct(boolean distinct)方法去重和使用 forUpdate(boolean forUpdate)方法进行更新***

- ***JDBC 选项***
  - **resultSetType(int resultSetType)：** 指定 ResultSet 将如何被转换。可用的值有 ResultSet.TYPE_FORWARD_ONLY, ResultSet.TYPE_SCROLL_INSENSITIVE（默认值）或者 ResultSet.TYPE_SCROLL_SENSITIVE。查看 Java 文档中 java.sql.ResultSet 获得更多信息。如果你想要更快的获得数据，一般使用 ResultSet.TYPE_FORWARD_ONLY 选项。如果你想要在部分结果中跳到某个索引确保使用默认的 ResultSet.TYPE_SCROLL_INSENSITIVE 选项。
  - **resultSetConcurrency(int resultSetConcurrency)：** 指定 ResultSet 是否可以被更新。可用的值有 ResultSet.CONCUR_READ_ONLY (默认的) 或者 ResultSet.CONCUR_UPDATABLE。既然更新通常作为独立的操作，那么实体门面 Entity Facade 应该几乎总是使用 ResultSet.CONCUR_READ_ONLY。
  - **fetchSize(Integer fetchSize)：** JDBC 获得默认查询 query. Default (null)的大小，并将会返回数据源设置。这并不是在 OFFSET/FETCH SQL 子句中获取（使用 offset/limit 方法），而是 JDBC 获取决定每次数据库来回返回多少行记录。仅用于 list()和 iterator()方法的查询。
  - **maxRows(Integer maxRows)：** JDBC 获得默认查询 query. Default (null)的最大行数，并将会返回数据源设置。这是在任意一个给定时间内保存在内存中的 ResultSet 的最大记录行数，除非被释放和被要求重新从数据库中请求获取。仅用于 list()和 iterator()方法的查询。

这里有很多的条件选项，EntityFind 接口本身的一部分以及一个更多的可用扩展集合使用 EntityConditionFactory 接口。使用 ec.entity.getConditionFactory()方法获得接口的一个实例，如下所示：

```
EntityConditionFactory ecf = ec.entity.getConditionFactory();
ef.condition(ecf.makeCondition(...));
```

使用 EntityFind.searchFormInputs()方法可以查找遵循标准的 Moqui 框架模式的表单（适用于 XML 表单查询字段以及在模板，JSON 和 XML 参数体中使用）。

一旦这些选项被指定，你就可以使用下面这些实际的操作去获得结果和改变数据了：

- ✧ 获得单一的 EntityValue（使用 **one**()方法）
- ✧ 获得多值对象的 EntityValueList（使用 **list** ()方法）
- ✧ 通过 EntityListIterator 在小点的批处理中去处理大规模集合的数据（使用 **iterator** ()方法）
- ✧ 获得匹配的结果总数（使用 **count**()方法）
- ✧ 更新所有匹配记录的指定字段（使用 **updateAll**()方法）
- ✧ 删除所有匹配的记录（使用 **delete**()方法）

通过视图实体灵活的查询

你可能会发现 EntityFind 接口操作应用于单个实体之上。你可以使用 XML 定义创建一个静态的视图实体并用一个实体名称命名，去进行多个关联在一起的实体间的查询，其他普通实体定义一起存放。

一个视图实体也可以通过数据库记录（使用 DbViewEntity 实体和关联实体）的形式定义，或者使用 EntityDynamicView 接口（使用 EntityFind.**makeEntityDynamicView**()方法获得一个接口）编码实现动态视图实体的创建。

➢ *静态视图实体*

一个视图实体由多个关联在一起的实体通过键值映射以及成员实体的别名字段使用可选的功能关联在一起。视图实体同样可以有关联的条件去封装一些视图中的数据约束。

例子组件的 ExampleViewEntities.xml 文件中有一个视图实体 XML 定义的片段：

```
<view-entity entity-name="ExampleFeatureApplAndEnum"
        package-name="moqui.example">
    <member-entity entity-alias="EXFTAP"    entity-name="ExampleFeatureAppl"/>
    <member-entity entity-alias="ENUM"    entity-name="moqui.basic.Enumeration"
            join-from-alias="EXFTAP">
        <key-map field-name="exampleFeatureApplEnumId"/>
    </member-entity>
    <alias-all entity-alias="EXFTAP"/>
    <alias-all entity-alias="ENUM"/>
</view-entity>
```

就像实体一样，视图实体使用 view-entity 元素上的 **entity-name** 和 **package-name** 属性定义了名称和所在的包名。

每个成员实体通过 member-entity 元素进行表示并且在 **entity-alias** 属性中使用别名唯一定义它们。这么做的部分原因是在一个视图实体中同一个实体可能会使用不同的别名出现多次。

注意第二个 member-entity 元素还有个 **join-from-alias** 属性去指定其关联于第一个成员实体。只有第一个成员实体没有 **join-from-alias** 属性。如果你想要当前实体在关联（SQL 中的左外连接）中是可选的，那么只要设置 **join-optional** 属性为 true。

使用 member-entity 元素下的一个或者多个 key-map 元素来描述两个实体之间是如何进

行互相关联的。key-map 元素有两个属性：**field-name** 和 **related-field-name**。注意当当前成员实体的主键字段匹配时，**related-field-name** 属性是可选的。

使用 alias-all 元素可以对所有的字段进行别名标识，就像上面的例子一样，或者使用 alias 元素去单独的设置别名。如果你想在一个字段上使用函数那么就使用 alias 元素单独去设置别名。注意在 SQL 数据库中，如果任何别名的字段有一个函数同时其他所有的字段没有函数但是在查询中被选中了，那么将会被添加到 group by 子句中以免出现无效 SQL。

- ***视图实体自动最小化查询***

当使用实体门面的 EntityFind 进行查询时，你可以指定查询的字段并且只有这些字段被选中。对于视图实体来说，这可以给你提供一些帮助，而无需进行过多的工作。

静态视图实体一个普遍的问题是你不得不将一大堆的成员实体进行关联来为搜索界面提供大量的操作和类似灵活的查询，并且当你为查询在数据库中使用这个“临时表”时，将会非常庞大。当通常的使用仅仅是选择明确的字段以及条件还有有限的排序字段时，你可能就不会关联一大部分用不到的表了。实际上，你为了自己需要的数据在请求数据库做了很多多于它需要做的事情。

一个解决这个问题的改进方法是创建一个动态视图实体 EntityDynamicView，并且只是连接你指定需要进行查询操作的实体。这种方式有效，但是有些笨重。

简单的方法是利用 EntityFind 里面的优势特性：自动最小化每个特定的查询中的字段和实体。在一个视图实体上只要指定选中的字段，条件以及排序字段。实体门面将自动遍历视图实体的定义并且只有别名的字段才会被上述方式之一使用（选择，条件，排序字段），同时只有关联实体的字段才会被使用(或者用于连接一个成员实体与其他成员实体用于完成连接关系的字段)。

地幔业务构件里 PartyViewEntities.xml 文件中定义的 FindPartyView 视图实体就是个很好的例子。这个视图实体里有相当多数量的 13 个成员实体。如果没有自动最小化方式，对于每个基于它的查询都需要将 13 个表进行关联。当有百万级的客户记录或是其他类似的大量当事人数据时，每个查询都要消耗好几分钟。当查询基于极少的字段并且关联很少量的成员实体以及最小数量的字段时，查询时间会减少到次秒级别。

实际的查询通过 mantle.party.PartyServices.**find#Party** 服务完成。这个服务的实现只有简单的 45 行 Groovy 脚本（findParty.groovy），并且脚本中大部分只是添加基于指定参数的查询条件。而使用 EntityDynamicView 方法做同样的事情需要数百行或者更多复杂的脚本，编写和维护都同样复杂。

- ***数据库定义视图实体***

除了在 XML 中定义视图实体，你还可以使用 DbViewEntity 和其关联的实体在数据库记录中定义它们。这种方式对于创建界面来说十分有用，特别是当用户凭空的定义一个视图（例如工具组件中的 EditDbView.xml 界面，在菜单的工具=>数据视图下），然后查询，查看以及通过一个用户定义的界面（例如 ViewDbView.xml 界面）导出数据的时候。

定义 DB 视图实体时并没有很多的可选项，但是主要的特性和相同的模式应用于此。这里的视图实体有名称(**dbViewEntityName**)，包名(**packageName**)以及结果是否缓存 **cache**。它同样有成员实体（DbViewEntityMember），map 键值去指定成员如何关联在一起（DbViewEntityKeyMap），以及字段别名（DbViewEntityAlias）。这里有个例子组件的案例：

```
<moqui.entity.view.DbViewEntity dbViewEntityName="StatusItemAndTypeDb"
        packageName="moqui.example"  cache="Y"/>
<moqui.entity.view.DbViewEntityMember
        dbViewEntityName="StatusItemAndTypeDb"  entityAlias="SI"
        entityName="moqui.basic.StatusItem"/>
<moqui.entity.view.DbViewEntityMember
        dbViewEntityName="StatusItemAndTypeDb"  entityAlias="ST"
        entityName="moqui.basic.StatusType"  joinFromAlias="SI"/>
<moqui.entity.view.DbViewEntityKeyMap
        dbViewEntityName="StatusItemAndTypeDb"  joinFromAlias="SI"
        entityAlias="ST"  fieldName="statusTypeId"/>
<moqui.entity.view.DbViewEntityAlias dbViewEntityName="StatusItemAndTypeDb"
        entityAlias="SI"  fieldAlias="statusId"/>
<moqui.entity.view.DbViewEntityAlias dbViewEntityName="StatusItemAndTypeDb"
        entityAlias="SI"  fieldAlias="description"/>
<moqui.entity.view.DbViewEntityAlias dbViewEntityName="StatusItemAndTypeDb"
        entityAlias="SI"  fieldAlias="sequenceNum"/>
<moqui.entity.view.DbViewEntityAlias dbViewEntityName="StatusItemAndTypeDb"
        entityAlias="ST"  fieldAlias="typeDescription"  fieldName="description"/>
```

就像你所见到的通过 XML 元素和属性名关联的实体和字段名。使用这些实体只需像其他实体一样，通过名称参照使用它们。

- ***动态视图实体***

即使使用自动视图实体最小化方式，实体门面还是需要通过一个查询，这里你仍然有需求去以编程的方式去凭空创建一个视图而不是定义一个静态的视图实体。

使用 EntityFind.**makeEntityDynamicView**()方法获得一个 EntityDynamicView 接口的实例可以做到。这个接口上有方法能做到和静态视图实体中 XML 元素一样的事情。使用 **addMemberEntity**(String entityAlias,String entityName,String joinFromAlias,Boolean joinOptional, Map<String, String> entityKeyMaps)方法添加成员实体。

静态视图实体（XML 定义方式）中不存在的一个便利操作就是可以基于一个关系定义去添加一个成员实体。使用 **addRelationshipMember**(String entityAlias, String joinFromAlias, String relationshipName, Boolean joinOptional)方法可以做到。

使用 **addAlias**(String entityAlias, String name, String field, String function)方法去对字段设置别名，这个方法的便捷变种是 **addAlias**(String entityAlias,String name)和 **addAliasAll**(String entityAlias, String prefix)方法。

你可以使用 **setEntityName**()方法来选择指定动态视图的名称，但通常这种方式用于调试十分有用，并且通常默认的名称（DynamicView）就可以了。

一旦你通过创建 EntityDynamicView 对象并指定条件和查询操作，就可以像正常使用 EntityFind 对象一样了。

实体 ECA 规则

实体 ECA（EECA）规则用于数据变化或者搜索时触发动作运行。这对于基于其他实体字段的实体字段（数据库中的列）或者在一个独立的系统中更新基于本系统中数据的数据来说十分有用。EECA 规则通常不用于触发业务过程，因为规则的应用太广泛了。服务 ECA 规则是触发过程的一个好工具。

这里的 EECA 规则的例子来源于地幔业务构件里的 Work.eecas.xml 文件，它调用一个服务在一个时间条目创建，更新或者删除时，去更新一个任务（WorkEffort）的总时间：

```
<eeca entity="mantle.work.time.TimeEntry"  on-create="true"  on-update="true"
        on-delete="true"  get-entire-entity="true">
    <actions><service-call in-map="context"
                name="mantle.work.TaskServices.update#TaskFromTime"/></actions>
</eeca>
```

一个 ECA（event-condition-action）规则是一个专门的规则类型去基于事件有条件的运行动作。对于实体 ECA 规则来说，事件是对于一条记录的各种查询和修改操作。可以设置下面任何属性为 true 去触发 EECA 规则：**on-create**，**on-update**，**on-delete**，**on-find-one**，**on-find-list**，**on-find-iterator**，**on-find-count**。

默认的实体 EECA 规则将会在实体操作后运行。设置 **run-before** 属性为 true 可以使规则运行在实体操作之前。这里还有一个 **run-on-error** 属性默认为 false，如果设置为 true 的话，即便是实体操作有错误产生，EECA 规则也将会被触发。

当动作运行时，上下文环境不管有没有服务运行在其中，都将加上传递到操作的实体字段值以便于使用。这里还有特别添加到上下文的字段：

- ✧ **entityValue:** 一个传递到实体操作的字段 Map 值。它可能未包含所有数据库记录设置的字段值。要从数据库记录来填充未传递的字段值，可以设置 **eeca.get-entire-entity** 属性为 true。
- ✧ **originalValue:** 如果 **eeca.get-original-value** 属性被设置为 true 并且 EECA 规则在实体操作之前运行（**run-before=**true），那么 EntityValue 对象则将展现数据库中原始（当前的）的值。
- ✧ **eecaOperation:** 一个展现操作的字符串用于触发 EECA 规则，基本上 **on-***属性名称没有“on-”开头。

condition 元素和 XML 动作中使用的约束条件起到相同的作用并大部分含有 expression 和 compare 元素，按照需求可以结合 or，and 和 not 元素一起使用。

动作 actions 元素和服务定义，界面，表单等定义中使用的 actions 动作元素起到相同的作用。它包含了一个 XML 动作脚本。查看 ***XML 动作总览（Overview of XML Actions）*** 章节部分获得更多的细节点。

实体数据导入和导出

加载实体 XML 和 CSV

实体记录可以使用 EntityDataLoader 来从 XML 和 CSV 文件中导入数据。我们可以通过实

体门面 API 中的 ec.entity.makeDataLoader()方法来获得一个接口实现的对象来达到这个目的，并通过其方法可以指定加载的数据然后加载它（使用 load()方法），获得一个记录的 EntityList 对象（使用 list()方法），或者验证当前的数据和数据库中的对比（使用 check()方法）。

这里对于指定加载的数据有一些选项。你可以使用 location(String location)和 locationList(List<String> locationList)方法去指定一个或者多个资源位置。你也可以使用 xmlText(String xmlText)和 csvText(String csvText)方法去直接使用文本。你还可以从组件的数据目录和 Moqui XML 配置文件里的 entity-facade.load-data 元素通过 dataTypes(Set<String> dataTypes)方法指定的类型数据来进行加载（只有匹配类型的文件才会被加载）。

使用 transactionTimeout(int tt)方法来设置事务超时时间不同于默认的机制，通常大部分用于处理大文件的执行。如果你期望改善插入执行机制，插入前不进行记录存在性的查询，插入失败即作更新操作，那么你可以向 useTryInsert(boolean useTryInsert)方法传递参数值为 true 来达到这个目的。

当初始数据中外键信息故障丢失，但是你又知道所有的数据最终需要被加载，那么可以向 dummyFks(boolean dummyFks)方法传递参数值为 true，这样将会为外键创建一条不存在的空记录。当真实的外键对应的记录被加载时，它会简单的更新这条空的虚拟记录。为了数据加载时禁用实体 ECA 规则可以向 disableEntityEca(boolean disableEeca)方法传递参数值为 true。

对于 CSV 文件来说，你可以指定解析文件使用的分隔符，包括 csvDelimiter(char delimiter)(默认为‘,’)，csvCommentStart(char commentStart) (默认为‘#’)和 csvQuoteChar(char quoteChar)(默认为‘"’)。

注意 EntityDataLoader 上所有的这些方法都会返回一个自参照，这样你可以链式调用，也就是说它是个 DSL 类型的 API。举例说明：

```
ec.entity.makeDataLoader().dataTypes(['seed', 'demo']).load()
```

除了使用 API 去直接加载数据，你还可以在默认的 Moqui 运行时的工具组件中的“工具=>实体=>导入界面”进行数据导入。你还可以对可执行 WAR 包文件通过使用-load 参数的命令行来加载数据。这里是数据加载器可用的命令行参数：

```
-load -------- 运行数据加载器
    -types=<type>[,<type>] – 加载的数据类型(可以是任何类型，通常为：种子,初始化种子，例子...)
-location=<location> ---- 加载的数据文件的位置
-timeout=<seconds> ------ 每个文件的事务超时时间，默认 600 秒 (10 分钟)
-dummy-fks ---------- 使用虚拟外键避免完整性参照错误
-use-try-insert --------- 尝试插入，错误时更新来取代插入前先检查记录
-tenantId=<tenantId> ---- 指定加载数据的租户 ID
```

举个例子：

```
$ java -jar moqui-${version}.war -load -types=seed,demo
```

实体数据文件必须有 entity-facade-xml 根元素，它有个类型 type 属性去指定文件中数据的类型，这个类型会和指定仅允许加载的类型比较，是否在其中或者是否所有的类型都加载。根元素下的每个元素名称都是一个实体或者服务名称。对于实体来说，每个属性都是一个字段名；对于服务来说，每个属性都是一个入参。

这里有个实体数据 XML 文件的例子：

```
<entity-facade-xml type="seed">
    <moqui.basic.LocalizedMessage  original="Example"  locale="es"
```

```
            localized="Ejemplo"/>
    <moqui.basic.LocalizedMessage   original="Example"   locale="zh"
            localized="样例"/>
</entity-facade-xml>
```

这里有个调用服务加载 CSV 文件的例子（同样的模式可应用于加载实体数据）：

```
# first line is ${entityName or serviceName},${dataType}
org.moqui.example.ExampleServices.create#Example, demo
# second line is list of field names
exampleTypeEnumId, statusId, exampleName, exampleSize, exampleDate
# each additional line has values for those fields
EXT_MADE_UP, EXST_IN_DESIGN, Test Example Name 3, 13, 2014-03-03 15:00:00
```

写入/导出实体 XML

最简单的导出数据到 XML 文件中是使用 EntityDataWriter，你可以通过 ec.entity.makeDataWriter()方法获得。通过这个接口你可以指定导出的实体名称以及其他各种选项，然后它将执行查询并导出数据到一个文件中（使用 int file(String filename)方法），每个实体生成一个文件在指定目录下（使用 int directory(String path)方法），或者到一个 Writer 对象（使用 int writer(Writer writer)方法）。所有的这些方法都会返回一个 int 值，表示的是写入的记录条数。

指定选项的方法返回一个自参照以开启链式调用方式。下面的是查询和导出选项的方法：

- **entityName**(String entityName)：指定查询和导出的实体名称。实体的数据按照多次调用这个方法或者 **entityNames**()方法的顺序进行查询和导出。
- **entityNames**(List<String> entityNames)：一个 List 的查询和导出的实体。实体的数据按照 list 列表指定的顺序或者 **entityName**()方法的调用次序进行查询和导出。
- **dependentRecords**(boolean dependents)：如果设置为 true，那么将为每条记录导出依赖的数据。这将大大降低导出的效率，所以只在较小的数据集合中才会使用它。查看 ***Dependent Entities*** 章节部分获取更多其包含的细节点。
- **filterMap**(Map<String, Object> filterMap)：一个过滤结果的条件字段名称和值的映射 Map。每个名称/值只用于实体上并且字段和名称相匹配。
- **orderBy**(List<String> orderByList)：结果的排序字段名称。每个名称只用于实体上并与字段名相匹配。也许会调用多次。每个条目可能都是逗号分隔的字段名称列表。
- **fromDate**(Timestamp fromDate), **thruDate**(Timestamp thruDate)：起始和截止日期用于过滤记录，和实体门面为每个实体自动添加的 **lastUpdatedStamp** 字段进行比较（除非在实体定义中关闭这种特性）。

这里有个导出某个时间段中所有订单头 OrderHeader 记录及其依赖的例子：

```
ec.entity.makeDataWriter().entityName("mantle.order.OrderHeader")
    .dependentRecords(true).orderBy(["orderId"]).fromDate(lastExportDate)
    .thruDate(ec.user.nowTimestamp).file("/tmp/TestOrderExport.xml")
```

另一种导出实体记录的方式是先做一个查询并得到 EntityList 或者 EntityListIterator 对象，然后调用 int **writeXmlText**(Writer writer, String prefix, boolean dependents)方法进行导出。这个方法将 XML 写入到 writer 对象中，可选的在每个元素包括依赖元素前添加前缀 prefix 值。

类似于实体数据导入交互界面，你可以使用默认的 Moqui 运行时工具组件中的“工具=>实体=>导出”界面进行导出。

界面和表单的简单查看和导出

许多在一起的工具使得查看和导出来源于不同表的数据库数据十分的容易。我们已经探寻过静态的（XML），动态的以及数据库定义的实体选项。在 ***User Interface*** 章节中有关于 XML 表单以及特定的列表形式表单的细节。

当一个 form-list 中的 auto-fields-entity.**entity-name** 属性有 **dynamic**=true 和一个${}字符串扩展，那么在界面渲染时它将会自动的呈现，这意味着单个表单可用于为任何作为界面参数给定的实体名称生成扁平列表形式的 HTML 或者 CSV 格式输出。

为了使事情更加有趣，列表展示结果的过滤可以使用一个动态的含有 auto-fields-entity 元素的 form-single，通过基于实体生成的查询表单来完成，并且含有 search-form-inputs 属性的 entity-find 元素会基于实体名称参数以及来源于查询表单的查询参数去进行查询操作。

下面实例展示的就是这些特性以及一个转换（DbView.csv）去导出 CSV 文件。无需过于担心理解关于界面，转换，表单以及渲染操作等细节，这些都在 ***User Interface*** 章节中涵盖了。这个界面定义是 Moqui 框架里默认的工具组件中的 ViewDbView.xml 界面的一个摘录：

```
<screen>
  <parameter name="dbViewEntityName"/>

  <transition name="filter"><default-response url="."/></transition>
  <transition name="DbView.csv">
    <default-response url="."><parameter name="renderMode" value="csv"/>
      <parameter name="pageNoLimit" value="true"/>
      <parameter name="lastStandalone" value="true"/></default-response>
  </transition>

  <actions>
    <entity-find entity-name="${dbViewEntityName}" list="dbViewList">
      <search-form-inputs/></entity-find>
  </actions>
  <widgets>
    <link url="DbView.csv" text="Get as CSV"/>
    <label text="Data View for: ${dbViewEntityName}" type="h2"/>

    <form-single name="FilterDbView" transition="filter" dynamic="true">
      <auto-fields-entity entity-name="${dbViewEntityName}"
          field-type="find"/>
      <field name="dbViewEntityName"><default-field>
        <hidden/></default-field></field>
      <field name="submitButton"><default-field title="Find">
        <submit/></default-field></field>
    </form-single>

    <form-list name="ViewList" list="dbViewList" dynamic="true">
      <auto-fields-entity entity-name="${dbViewEntityName}"
          field-type="display"/>
    </form-list>
  </widgets>
</screen>
```

虽然这个界面是为了设计给用户交互使用，但是它也可以在一个 web 或者其他 UI 环境

之外去生成 CSV 到一个文件中或者其他位置。如果你只是要写一个界面那是相当的容易，基本上只要 parameter 元素，单个 entity-find 动作以及简单的 form-list 定义。转换和查询界面可以不需要。

界面渲染的代码如下所示：

```
ec.context.putAll( [pageNoLimit:"true", lastStandalone:"true",
    dbViewEntityName:"moqui.example.ExampleStatusDetail"] )
String csvOutput = ec.screen.makeRender()
  .rootScreen("component://tools/screen/Tools/DataView/ViewDbView.xml")
  .renderMode("csv").render()
```

数据文件

数据文件是将数据库记录装配为一个 JSON 文件或者一个 Java 内嵌的 Map/List 表现的文件。

下面是一个数据文件实例的案例并且用 DataDocument*记录来定义它。这个例子从基于 Moqui 和 Mantle 的 HiveMind PM 项目中选择。这个文件是为了定义一个工作投入 WorkEffort 的项目类型。

```
{
    "_index": "hivemind",
    "_type": "HmProject",
    "_id": "HM",
    "_timestamp": "2013-12-27T00:46:07",
  "WorkEffort": {
        "workEffortId": "HM",
        "name": "HiveMind PM Build Out",
        "workEffortTypeEnumId": "WetProject"
    },
    "StatusItem": { "status": "In Progress" },
    "WorkEffortType": { "type": "Project" },
    "Party": [
        {
            "Person": { "firstName": "John", "lastName": "Doe" },
            "RoleType": { "role": "Person - Manager" },
            "partyId": "EX_JOHN_DOE"
        },
        {
            "Person": { "firstName": "Joe", "lastName": "Developer" },
            "RoleType": { "role": "Person - Worker" },
            "partyId": "ORG_BIZI_JD"
        }
    ]
}
```

下面的是在实体门面文件中所用的格式来定义数据库记录的数据文件：

```
<moqui.entity.document.DataDocument dataDocumentId="HmProject"
    indexName="hivemind" documentName="Project"
    primaryEntityName="mantle.work.effort.WorkEffort"
    documentTitle="${name}"/>
<moqui.entity.document.DataDocumentField dataDocumentId="HmProject"
    fieldPath="workEffortId"/>
<moqui.entity.document.DataDocumentField dataDocumentId="HmProject"
    fieldPath="workEffortName" fieldNameAlias="name"/>
<!-- this is aliased so we can have a condition on it -->
<moqui.entity.document.DataDocumentField dataDocumentId="HmProject"
    fieldPath="workEffortTypeEnumId"/>
<moqui.entity.document.DataDocumentField dataDocumentId="HmProject"
    fieldPath="WorkEffort#moqui.basic.StatusItem:description"
    fieldNameAlias="status"/>
<moqui.entity.document.DataDocumentField dataDocumentId="HmProject"
    fieldPath="mantle.work.effort.WorkEffortParty:partyId"/>
<moqui.entity.document.DataDocumentField dataDocumentId="HmProject"

fieldPath="mantle.work.effort.WorkEffortParty:mantle.party.RoleType:description"
    fieldNameAlias="role"/>
<moqui.entity.document.DataDocumentRelAlias dataDocumentId="HmProject"
    relationshipName="mantle.work.effort.WorkEffort"
    documentAlias="WorkEffort"/>
<moqui.entity.document.DataDocumentRelAlias dataDocumentId="HmProject"
    relationshipName="WorkEffort#moqui.basic.StatusItem"
    documentAlias="StatusItem"/>
<moqui.entity.document.DataDocumentRelAlias dataDocumentId="HmProject"
    relationshipName="mantle.work.effort.WorkEffortParty"
    documentAlias="Party"/>
<moqui.entity.document.DataDocumentRelAlias dataDocumentId="HmProject"
    relationshipName="mantle.party.RoleType" documentAlias="RoleType"/>
<moqui.entity.document.DataDocumentCondition dataDocumentId="HmProject"
    fieldNameAlias="workEffortTypeEnumId" fieldValue="WetProject"/>
<moqui.entity.document.DataDocumentLink dataDocumentId="HmProject"
    label="Edit Project"
    linkUrl="/apps/hm/Project/EditProject?workEffortId=${workEffortId}"/>
```

数据文件实例的顶级对象（JSON 条目，Java 中的 Map）有 3 个字段定义在文件中：

- **_index:** 文件的索引必须存在，来源于文件定义的 DataDocument.**indexName** 字段
- **_type:** Moqui 框架使用文件类型，索引以及 ID 来进行数据文件定义，来源于 DataDocument.**dataDocumentId** 字段
- **_id:** 特定数据文件实例的 ID，基于实体的主键字段，定义在字段 DataDocument.**primaryEntityName** 中

顶级对象中还包含**_timestamp** 字段用于记录文件生成的日期和时间。

这四个字段如此命名是为了使用 ElasticSearch 方便的检索，这个数据搜索特性的工具是基于数据文件的特性之上。一般来说这些字段和数据文件，除了搜索之外，对于通知，集成和各种事情来说都很有用。

数据文件定义通过这些记录构成：

- DataDocument: 通过 **dataDocumentId** 字段定义的主记录，包含索引名称，文件名称（为了显示的目的）
  - **primaryEntityName:** 文件的主实体，文件中其他实体的字段都与之关联并且有普通的字段名
  - **documentTitle:** 为了显示的目的，特别是搜索结果等。注意 **documentTitle** 值使用来源于数据文件的一个 Map 值来扩展，所以扩展字段的名称必须匹配文件中字段名（或者别名）
- DataDocumentField: 每条记录为文件指定的字段
  - **fieldPath：** 字段名称，冒号分隔的关系名称列表之前部分是可选的，关系名称是从主实

体到字段所在的实体。

- **fieldNameAlias：** 可选项，用于当来源于实体的字段有不同时，指定文件中使用的字段名称。文件中的字段名称应当是文件内唯一的，而不仅仅是实体内唯一。不管实体字段名或者别名是否被使用都应当正确。这么做的原因是：这是在查询中使用别名去数据库里为文件获取数据，并且有利于参数化搜索。

- DataDocumentRelAlias: 使用这些记录来产生一个更清洁的文件，通过在 **fieldPath** 字段里指定一个关系的别名以及 **primaryEntityName** 字段
- DataDocumentCondition: 这些记录包含了从数据库中为文件获得数据的查询约束条件。在上面的案例中只是为了约束获得 WetProject 类型的工作投入 WorkEffort 记录的查询，所以这里只包含了项目。
- DataDocumentLink: 在检索结果，其他用途，还有系统接口中，一个关于文件更多信息的链接十分有用，特别是可用的主实体在其中时。使用这些记录去指定这样的链接。注意 **linkUrl** 值使用来源于数据文件的一个 Map 值来扩展，所以扩展字段的名称必须匹配文件中字段名（或者别名）。

在这个案例文件里的顶级对象中，WorkEffort 对象是文件中的主实体。这里总是会有一个类似本文件中的对象并且其名称和主实体的名称相同。这个名称将是一个逐字的 DataDocument.**primaryEntityName** 字段的值，除非在 DataDocumentRelAlias 记录中定义了别名，这也是在文件中这个对象的名字是"WorkEffort"而不是"mantle.work.effort.WorkEffort"。

所有带有普通字段名称(无冒号分隔的关系前缀)的 **fieldPath** 字段的 DataDocumentField 记录都映射了主实体上的字段，并且都被文件中主实体的对象所包含。

所有冒号分隔关系名称前缀的文件字段，都将导致带有条目键值的顶级文件对象（Map）中其他条目作为关系的名称或者关系的别名，如果配置别名的话。条目的值将是个对象/Map，如果它是一个关系一个类型；或者一个对象数组（Java 中由 Maps 组成的 List），如果它是多个关系一个类型。

同样的模式可适用于 **fieldPath** 中多于一个冒号分隔的关系名称。object/Map 条目将会按照需要被内嵌到指定字段的路径后。上面 HmProject 案例文件中包含的这个例子就是 "mantle.work.effort.WorkEffortParty:mantle.party.RoleType:description" 这个 **fieldPath** 的值。注意一下，这两个关系名称设置别名排斥包名称，并且字段别名命名使用 role 而不是 description。JSON 文件这部分的结果就是：

```
{"Party": [ { "RoleType":{ "role": "Person - Manager" } } ] }
```

JSON 语法对于一个对象（Map）使用大括号（{}），对于一个数组（List）使用中括号（[]）。所以上面我们获得的顶级对象有一个 Party 条目，其值为一个数组。数组中的每个对象都是一个 RoleType 条目，它的值是单个键为 **role**，值为 RoleType.description 实体字段值的条目。**description** 字段别名命名为 **role** 的原因是上面描述的是 DataDocumentField.**fieldNameAlias** 字段：在整个文件中，数据文件中的每个字段都必须有一个唯一的名称。

这里有几个从数据库中的数据生成数据文件的方法。最常用的方式就是下面描述的供给数据 Data Feed，但你同样可以通过 API 调用的方式这样来获取：

```
List<Map> docMapList = ec.entity.getDataDocuments(dataDocumentId, condition,
        fromUpdateStamp, thruUpdatedStamp)
```

在返回的 List 中，每个 Map 都代表一个数据文件。Condition，fromUpdateStamp 以及 thruUpdatedStamp 参数可以为空，但如果指定的话作为额外的约束条件去进行数据库查询。条件在文件中必须使用字段的别名。在某个确定的时间段内查看文件的任意部分是否被修改，可以使用*UpdatedStamp 参数查询任意实体记录，以及在起止时间段内的自动添加的 **lastUpdatedStamp** 字段值。

数据文件的 Map 和上面 JSON 文件案例使用相同的结构。ElasticSearch API 支持这个 Map 形成一个文件，但是在某些情况下你希望它变成一个 JSON 字符串。你可以在 Groovy 中使用一个这样简单的声明去从这个 Map 中创建一个 JSON 字符串：

```
String docString = groovy.json.JsonOutput.toJson(docMap)
```

如果你想要一个更友好点的人为可读版本的 JSON 字符串可以这么做：

```
String prettyDocString = groovy.json.JsonOutput.prettyPrint(docString)
```

反过来转换（从一个 JSON 字符串中得到一个 Map 展现方式）的声明如下所示：

```
Map docMap = (Map) new groovy.json.JsonSlurper().parseText(docString)
```

数据供给

数据供给是一个可配置化的方式去推送数据文件到一个服务上或者通过一个 API 调用去在多组文件中进行检索。

下面的案例是一个推送供给（**dataFeedTypeEnumId**="DTFDTP_RT_PUSH"），当数据文件中的任意数据通过 Moqui 框架的实体门面（Entity Facade）在数据库中发生变化的时候，发送文件到 HiveMind.SearchServices.indexAndNotify#HiveMindDocuments 服务上。框架自动保持追踪数据供给推送，以及数据文件中关联的实体部分的新增，修改以及删除操作完成的变化情况。这是一个实时有效率的获得更新的数据文件方式。

这里有一个 entity-facade-xml 的案例用于数据供给推送的配置记录：

```
<moqui.entity.feed.DataFeed  dataFeedId="HiveMindSearch"
    dataFeedTypeEnumId="DTFDTP_RT_PUSH"  feedName="HiveMind Search"
feedReceiveServiceName="HiveMind.SearchServices.indexAndNotify#HiveMindDocuments"/>
<moqui.entity.feed.DataFeedDocument  dataFeedId="HiveMindSearch"
    dataDocumentId="HmProject"/>
<moqui.entity.feed.DataFeedDocument  dataFeedId="HiveMindSearch"
    dataDocumentId="HmTask"/>
```

供给中的每个 DataFeedDocument 记录都会关联一个 DataDocument 记录到 DataFeed 记录上。

在另一个侧面说明，当你想要在加载过程中给其中的 XML 数据文件中的数据建立索引，并且这个加载过程可能需要在数据供给加载之前进行激活时，你可以在一条 ServiceTrigger 记录中添加一个元素，然后服务门面将会调用服务为这个供给在加载过程中进行索引建立。这里是一个例子：

```
<moqui.entity.ServiceTrigger  serviceTriggerId="HM_SEARCH_INIT"
    statusId="SrtrNotRun"  mapString="[dataFeedId:'HiveMindSearch']"
    serviceName="org.moqui.impl.EntityServices.index#DataFeedDocuments"/>
```

上面的 DataFeed 案例是一个数据供给的推送。为手动的推送安装一个供给只需要在 DataFeed 记录上设置 **dataFeedTypeEnumId**="DTFDTP_ MAN_PUSH"。任何类型的数据供给都可以被手动检索，但是如果是手动类型的话，供给将不会自动运行了。任何供给的文件获取可以通过 API 如下进行声明：

```
List<Map> docList = ec.entity.getDataFeedDocuments(dataFeedId, fromUpdateStamp,
thruUpdatedStamp)
```

数据检索

Moqui 框架的数据检索特性基于 ElasticSearch（http://www.elasticsearch.org/）。这是一个基于 Apache Lucene 的分布式文本检索工具。ElasticSearch 使用 JSON 文件作为工件进行检索，并且 JSON 文件上面每个命名的字段都是一个检索的方面。数据文件特性生成的文件包含 4 个指定字段提供 ElasticSearch 使用，作为数据文件部分的描述（**_index**， **_type**， **_id** 以及 **_timestamp**）。

数据检索里有两个主要的触点：建立索引和检索。框架中建立索引的服务是 org.moqui.impl.EntityServices.index#DataDocuments 。这个服务实现了 org.moqui.EntityServices.receive#DataFeed 接口，并且它接受所有来源于接口的参数但只使用 **documentList** 参数，这个参数是使用 ElasticSearch 建立索引的数据文件的列表。

它还有另一个参数 **getOriginalDocuments**，当其设置为 true 时，服务会填充并返回 **originalDocumentList**，这是一个之前的索引列表版本并匹配来源于 ElasticSearch 中已存在的文件。服务总会返回一个 **documentVersionList** 参数，这个参数是一个记录每个原始列表中的文件在索引建立后版本编号的列表，去展示每个文件在索引中被更新了多少次。

前一章节的实例中使用了一个特定的应用程序服务去接收数据供给推送，所以这里的示例是一个使用索引配置的数据供给推送服务，这是框架的一部分：

```
<moqui.entity.feed.DataFeed  dataFeedId="PopCommerceSearch"
    dataFeedTypeEnumId="DTFDTP_RT_PUSH"  feedName="PopCommerce Search"
    feedReceiveServiceName="org.moqui.impl.EntityServices.index#DataDocuments"/>
<moqui.entity.feed.DataFeedDocument  dataFeedId="PopCommerceSearch"
    dataDocumentId="PopcProduct"/>
```

你同样可以使用 ElasticSearch API 去直接建立文件索引，无论是通过实体门面生成的数据文件，还是任何你想要检索的 JSON 文件。更完整的信息请查看 ElasticSearch 文档。这里有一个内嵌 Map 形式的 JSON 文件使用**_index**，**_type** 以及 **_id** 条目建立索引的例子：

```
IndexResponse response = ec.elasticSearchClient
    .prepareIndex(document._index, document._type, document._id)
    .setSource(document).execute().actionGet()
```

使用 org.moqui.impl.EntityServices.search#DataDocuments 服务去检索数据文件，像这样：

```
<service-call  name="org.moqui.impl.EntityServices.search#DataDocuments"
    out-map="context"  in-map="context + [indexName:'popc']"/>
```

注意在这个案例中，**queryString**，**pageIndex** 以及 **pageSize** 参数来源于查询的表单，并且从上下文 context 里的请求参数中获得。这个服务的参数有：

- **queryString:** 检索查询字符串将会被传递到经典的 Lucene 查询解析器中，文档参阅：http://lucene.apache.org/core/4_8_1/queryparser/org/apache/lucene/queryparser/classic/package-summary.html
- **documentType:** ElasticSearch 的文件类型，匹配文件中的**_type** 字段和 DataDocument.**dataDocumentId**；上一章节中的案例包括 PopcProduct 和 HmProject
- **pageIndex, pageSize:** 这些是 Moqui XML 列表表单的标准分页参数，所以服务可以很轻易的使用它们；只有 **pageSize** 大小的结果将被返回，并且结果开始于 **pageIndex** * **pageSize** 的索引

- **flattenDocument:** 默认为 false，如果设置为 true，那么所有的文件都会以内嵌 Map 的形式进行展现，所有来源于内嵌 Map 和 List 中 Map 的结果将被简单拉平为一个扁平的名称/值对形式的 Map 值；稍后文件中的数值将会在发现相同的 Map 条目键超过一次时，覆盖之前的数值（查看 StupidUtilities.flattenNestedMap()方法）

服务返回一个 **documentList** 参数，这是一个 Map 类型的 List，每个 Map 都表示一个数据文件。它同样返回各种 **documentList***参数，这些参数是 Moqui XML 列表表单分页模式的一部分（*Count，*PageIndex，*PageSize，*PageMaxIndex，*PageRangeLow 以及 *PageRangeHigh）。这些用于渲染一个列表表单，并且基于某些目的可以被用于其它有用的地方。

除了服务之外，你还可以直接使用 ElasticSearch API 去检索结果。注意这里有两个主要的步骤，第一步，检索取到每个文件中 3 个标识字段；第二步，一个多值返回取到所有的文件。本例中，我们将使用 Map（使用 **getSourceAsMap()**方法）类型去获取每个文件，ElasticSearch API 同样支持 JSON 文件的获取方式（使用 **getSourceAsString()**方法）。

```
SearchHits hits =
ec.elasticSearchClient.prepareSearch().setIndices(indexName)
    .setTypes(documentType).setQuery( QueryBuilders.queryString(queryString) )
    .setFrom(fromOffset).setSize(sizeLimit).execute().actionGet().getHits()
if ( hits.getTotalHits() > 0 ) {
    MultiGetRequestBuilder mgrb = ec.elasticSearchClient.prepareMultiGet()
    for (SearchHit hit in hits)
        mgrb.add( hit.getIndex(), hit.getType(), hit.getId() )
    Iterator mgirIt = mgrb.execute().actionGet().iterator()
    while( mgirIt.hasNext() ) {
        MultiGetItemResponse mgir = mgirIt.next()
        Map document = mgir.getResponse().getSourceAsMap()
        documentList.add(document)
    }
}
```

除了索引和检索之外，通晓 ElasticSearch 的另一个方面就是部署选项。Moqui 框架为了快速方便的访问，默认有一个内嵌的 ElasticSearch 节点运行在同一个 JVM 中。一个远程的 ElasticSearch 服务也可以使用。

最简单的分布式部署模式是将每个 Moqui 应用服务器作为 ElasticSearch 集群中的一个节点，并且如果你已经将 ES（ElasticSearch）节点和节点上实际进行检索的数据文件分离开来了，那么就要设置 ES 应用服务器节点上不保存任何数据。通过这种方式，可能检索结果聚合在应用服务器上，但是实际的数据检索工作将在集群中其他的服务器上完成。

六、 逻辑和服务

服务定义

Moqui 框架中业务逻辑的单元是服务。Moqui 框架是一个面向服务的架构，它是一个内部细粒度化逻辑单元和外部粗粒度化逻辑单元的聚合体。Moqui 服务有：

- ✧ 事务处理
- ✧ 安全机制（鉴权和授权，加上为了速度限制的伺服器）
- ✧ 验证（数据类型以及输入参数的各种约束）
- ✧ 各种语言和工具的实现，包括脚本语言，Java 方法，甚至 Apache Camel 终点
- ✧ 本地或者远程调用
- ✧ 同步运行，异步运行或者任务调度运行
- ✧ 通过服务事件-条件-动作（SECA）规则，在各种阶段的执行中运行其他服务的来源
- ✧ 可选的通过数据库信号去限制单个运行中的实例

服务使用服务元素定义在服务 XML 文件中。服务的名称由一个路径，一个动词和一个名词按照**"${path.verb#noun}"**结构组成。注意服务定义中的名词是可选的，并且在服务名称中动词和名词之间的井号（#）也是可选的，这里有个例子，mantle.party.PartyServices.create#Person 服务（来源于地幔业务构件）：

```
<service verb="create"    noun="Person">
    <in-parameters>
        <parameter name="partyId"/>
        <auto-parameters entity-name="mantle.party.Person"    include="nonpk"/>
        <parameter name="firstName"    required="true"/>
        <parameter name="lastName"    required="true"/>
        <parameter name="roleTypeId"/>
    </in-parameters>
    <out-parameters><parameter name="partyId"/></out-parameters>
    <actions>
        <service-call name="create#mantle.party.Party"    out-map="context"
                    in-map="[partyId:partyId, partyTypeEnumId:'PtyPerson']"/>
        <service-call name="create#mantle.party.Person"    in-map="context"/>
        <if condition="roleTypeId">
            <service-call name="create#mantle.party.PartyRole"
                        in-map="[partyId:partyId, roleTypeId:roleTypeId]"/>
        </if>
    </actions>
</service>
```

服务中唯一必须的属性就是 **verb**，尽管通常建议使用 **noun** 属性。**type** 属性通常也被使用，但一般默认为"inline"，就像上面的服务一样由一个 actions 元素包含了这个服务的实现。对于其他类型的服务，例如其他方式实现的一个服务，**location** 和 **method** 属性用于指定运

行什么。

上面的例子中有 in-parameters 元素，包含了个别 parameter 元素和一个 auto-parameters 元素用于 mantle.party.Person 实体上所有的非主键字段的推送。它同样还有一个 out-parameters 元素，这里参数 partyId 要么当没有作为传入参数传递时生成，要么简单的通过参数传递一个值。

actions 元素包含了一个 XML 动作脚本的服务实现。本例的情况下调用了两个服务，并且有条件的调用第三个服务，如果传递了 roleTypeId 参数则调用。注意这里对 partyId 出参（在结果 Map 中）并没有明确的设置，这是因为服务门面会自动的在服务实现运行去设置输出/结果 Map 之后，从上下文中获得每个公开的出参。

服务 service 元素中可用的属性有：

- **verb:** 它可以是任意的动词并通常是这些之一：create, update, store, delete 或者 find。服务的全名是："${path}.${verb}#${noun}"。动词是必须的并且名词是可选的，所以如果没有名词，那么服务名就只有动词。
- **noun:** 对于实体自动服务来说必须是有效的实体名称。在很多其他的情况下，实体名称是最好的方式去描述当前操作的是什么，但是这个名词确实可以是任何东西。
- **type:** 指定服务如何实现的服务类型。默认可用的选项包括：inline, entity-auto, script, java, interface, remote-xml-rpc,remote-json-rpc 以及 camel。额外的类型可以通过实现 org.moqui.impl.service.ServiceRunner 接口和在 Moqui XMl 配置文件中添加一个 service-facade.service-type 元素来进行添加。默认值 inline 表示服务实现在 service.actions 元素下。
- **location:** 服务的位置。对于脚本来说，这是脚本文件的资源门面位置。对于 Java 类方法来说，这是全类名。对于远程服务来说，这是远程服务的 URL 地址。还可以参照一个 Moqui XML 配置文件中 service-facade.service-location 元素预定义的位置来替代一个真实的位置。这对于远程服务 URLs 来说特别有用。
- **method:** 如果服务类型可用，表示路径里指定的方法。
- **authenticate:** 如果不设置为 false（默认是 true），那么必须由一个登录用户去运行服务。如果服务运行在可执行上下文中是通过一个登录用户的话，那么是有资格的。如果不是，那么不管是"授权用户帐号"（authUserAccount）参数还是"授权用户名"（authUsername）和"授权密码"（authPassword）参数都必须被指定，并且都必须包含系统中用户的有效数据。"授权租户 ID"（authTenantId）参数也可能需要被指定，用于鉴定这个用户存在于指定的租户实例中。如果这个参数指定了，那么该租户将作为上下文租户去运行服务。服务参数也可以设置给匿名所有操作（anonymous-all）和匿名查看（anonymous-view），这样不仅无需进行服务权限鉴定，并且服务也将为所有动作或者仅查看去运行，就像被认证过了一样（使用_NA_ UserAccount）。
- **allow-remote:** 默认为 false，意味着这个服务不能被远程接口类似 JSON-RPC 和 XML-RPC 进行调用。如果设置为 true，那么就可以。在设置为 true 之前，确保服务被充分的保护了（授权和鉴权）。
- **validate:** 默认为 true。设置为 false 去取消入参校验以及不自动删除未指定的参数。
- **transaction:**
  - ignore: 不进行任何事务处理（如果有人正好在使用它，那么不要开启一个事务）
  - use-or-begin: 使用活跃的事务或者如果没有活跃的事务则开启一个。这是默认配置。
  - force-new: 总是开启一个新的事务，如果当前存在一个则暂停或者重新开始活跃的事务
  - cache: 类似 use-or-begin，但是当前每个事务连续写入缓存（甚至当前有活跃的 TX 都这样工作）。查看 JavaDoc 文档中 TransactionCache 类的说明获取更多注意事项和警告的细节。
  - force-cache: 类似 force-new，在当前有一个事务缓存使用 cache 选项

- **transaction-timeout:** 事务超时时间，单位：秒。这个值仅用于服务开启一个事务（force-new, force-cache, use-or-begin 或者 cache，并且当前没有其他的事务存在）。
- **semaphore:** 特用于长时间运行的服务（通常为任务调度）。使用数据库中的一条记录去锁住服务，这样在任何给定的时间段内对于指定的数据库只有一个服务实例。选项包括 none（默认的），fail 和 wait。
- **semaphore-timeout:** 服务超时等待时间，单位：秒。默认为 120s。
- **semaphore-sleep:** 检查服务信号的休眠间隔，单位：秒。默认为 5s。
- **semaphore-ignore:** 本次服务之后忽略已存在的信号，单位：秒。默认为 3600s（1 小时）。

服务的输入和输出都是一个名称/数值条目的 Map。入参通过 in-parameters 元素指定，出参通过 out-parameters 元素指定。在这些元素下使用 parameter 元素去定义单个的参数，并且 auto-parameters 元素用于自动定义基于主键（pk），非主键（nonpk）或者实体所有字段的参数。

个体的 parameter 元素有如下属性去定义它，包括：

- **name:** 参数名称，匹配服务传入和返回的参数 Map 条目的键
- **type:** 属性类型，一个 Java 类的全名称或者一个普通的 Java API 类型（包括 String, Timestamp, Time, Date, Integer, Long, Float, Double, BigDecimal, BigInteger, Boolean, Object, Blob, Clob, Collection, List, Map, Set, Node）。
- **required:** 默认为 false，如果参数为必要参数则为 true。也可以设置为 disabled，用于展现类似参数不存在一样的情况，当覆盖一个预定义的参数时十分有用。
- **allow-html:** 仅适用于 String 字符串字段。仅仅检查入参（意味着校验来源于用户，其他系统等的输入）。默认为 none，表示不允许 HTML（将会导致出现错误信息）。如果期望有一些 HTML 使用 safe，它将遵循 antisamy-esapi.xml 文件中定义的安全规则。这对于内部和公开的用户来说都是安全的。当用户是可信任的或者非敏感字段这种罕见场景时，any 选项可以使用于一点都不检查 HTML 内容。
- **format:** 仅用于参数是作为 String 类型传入的，但是需要进行类型转换。对于日期/时间类型来说，使用标准的 Java SimpleDateFormat 字符串。
- **default:** 当参数没有数值传入时，指定的默认使用的字段或者表达式（仅用于 **required**=false 的情况）。类似于默认值，但这里是一个字段名称或者表达式，而不是一个文本值。如果这个属性和 **default-value** 属性都被指定了，那么将首先进行评估计算，并只有 **default** 属性的值为空时才使用 **default-value** 属性值。
- **default-value:** 当参数没有数值传入时，指定的默认文本值（仅用于 **required**=false 的情况）。如果本属性和 **default** 属性都被指定了，默认首先进行 **default** 属性值的计算，并且本属性仅在 **default** 属性值计算结果为一个空值时才使用。
- **entity-name:** 可选项，实体的名称，并且参数和该实体上的某个字段有关联。
- **field-name:** 可选项，字段名称，是已设置实体名的实体字段，并且和参数关联。对于通过表单字段定义自动生成的服务参数来说最有用。当参数定义使用 auto-parameters 元素时，相应传递的参数会被自动填充数值。

对于包含了其他对象（例如 List，Map 和 Node）的参数对象类型来说，参数元素能够被内嵌去指定在参数对象中期待的东西（如果验证可用）。

除了 **required** 属性之外，每个参数的校验还可以通过这些子元素进行指定：

- matches: 验证当前的参数和 **regexp** 属性指定的普通表达式是否匹配
- number-range: 验证数字是否在 **min** 和 **max** 范围内

- number-integer: 验证数字是否为整数
- number-decimal: 验证数字是否为小数
- text-length: 验证文本长度是否在 **min** 和 **max** 范围内
- text-email: 验证文本是否是一个有效的 email 地址
- text-url: 验证文本是否是一个有效的 URL
- text-letters: 验证文本是否只包含字母
- text-digits: 验证文本是否只包含数字
- time-range: 通过指定的 **format** 去验证日期/时间是否在 **before** 和 **after** 范围内
- credit-card: 如果指定了给定的卡类型 **types**，验证文本是否是一个有效的使用 Luhn MOD-10 规则的信用卡号

验证元素可以和 val-or 及 val-and 元素一起使用，或者使用 val-not 元素用于不验证的场景。

当一个 XML 表单字段的配置基于带验证的服务参数，那么明确的验证将会通过 JavaScript 在浏览器中自动的进行验证，包括 **required**, matches, number-integer, number-decimal, text-email, text-url 以及 text-digits。

现在你的服务已经定义好了，当服务调用时，服务门面的行为基本上已经配置完成，现在是时候去实现它了。

服务实现

有一些服务类型有本地实现，然而其他没有实现（interface），或者服务定义是别的东西的一个代理，并且服务位置指向一个外部的实现（remote-xml-rpc, remote-json-rpc 以及 camel）。远程和 Apache Camel 类型在系统接口章节有详细的描述。

服务脚本

通常脚本是实现服务最好的方式，除非是一个自动实现的实体 CrUD 操作的服务。当脚本缓存被清理时，脚本会被自动重载，并且在开发模式中这些缓存默认在一个很短的时间内期满以便于自动更新。

脚本能够高效率的运行，特别是 Groovy 脚本，可以在运行时编译成 Java classes 并以编译后的形式进行缓存，所以它们可以很快的运行。XML 动作脚本先被转换成 Groovy 脚本（查看 XmlActions.groovy.ftl 文件获得详细信息），然后编译和缓存，所以其性能概要和一个普通的 Groovy 脚本一样。

任何资源门面能够运行的脚本都可以被用于服务的实现。查看 ***Rendering Templates and Running Scripts*** 章节获得详细信息。总之，默认支持的脚本有 Groovy，XML 动作以及 JavaScript。任何脚本语言都可以通过 javax.script 或者 Moqui 指定的接口实现去得到支持。这里有一个使用 Groovy 脚本的服务实现，定义在 org.moqui.impl.EmailServices.xml 文件中：

```
<service verb="send"  noun="EmailTemplate"  type="script"
         location="classpath://org/moqui/impl/sendEmailTemplate.groovy" >
    <implements service="org.moqui.EmailServices.send#EmailTemplate" />
</service>
```

在这个例子中，**location** 是一个 classpath 类路径，但是任何资源门面支持的位置类型都可以被使用。查看 ***Resource Locations*** 章节中关于如何在组件，本地文件系统或甚至普通的

URLs 中指定文件位置。

所有在脚本开始时传递到服务上的入参，或者通过服务定义设置的默认值，都将会在上下文环境中作为可用的字段来供脚本使用。就像 Moqui 中的其他构件，这里也有一个 ec 字段存在当前的可执行上下文 ExecutionContext 对象中。

注意一下，脚本有一个单独的上下文环境，独立于使用 ContextStack.**pushContext**() 和 **popContext**()方法调用它的上下文。这意味着在服务运行创建字段后，不仅上下文中不持久化数据，而且服务不访问调用它的上下文，尽管它在本地运行并在相同的调用服务的可执行上下文 ExecutionContext 中。

为了方便起见，这里上下文环境中有一个 result 字段，类型为 Map<String, Object>。你可以将出参放在这个 Map 中进行返回，但是这样做并不是必须的。脚本运行后，脚本服务执行器在上下文中查找所有定义在服务上的出参，并将它们加到结果中。脚本也可以返回（计算得到）一个返回结果的 Map 对象。

- ***内联动作***

本章开始附近的服务定义案例展示了一个默认服务类型 inline 的服务。本例中的实现在 service.actions 元素中，其中还包含了一个 XML 动作脚本。它被服务路径视为一个外部的脚本对待，但是为简单起见并减少使用的文件数量可以在服务定义中使用内联动作。

Java 方法

一个服务实现也可以是一个 Java 方法，既可以是一个类（静态的）方法，也可以是一个对象方法。如果方法不是静态的，那么服务执行器会使用默认（无参数的）的构造函数去新建这个对象的实例。

这个方法必须带一个 ExecutionContext 参数并返回一个 Map<String,Object>，因此这个方法定义应该如下所示：

```
Map<String, Object>    myService(ExecutionContext ec)
```

实体自动服务

entity-auto 类型的服务你无需去实现它，其实现基于 **verb** 和 **noun** 属性值自动生成。动词可以是 create,update,delete 或 store（如果记录不存在则新建，存在时更新）。名词是一个实体名称，既可以是一个带包路径的全名也可以仅仅是不带包名的实体名称。

实体自动服务可以仅仅调用不带路径（包名或者文件名）的，名为**${verb}#${noun}**的服务来隐式（自动）的定义。例如：

```
ec.service.sync().name( "create",   "moqui.example.Example" )
        .parameters( [exampleName:'Test Example'] ).call( )
```

当你定义服务并使用 entity-auto 类型的实现时，你可以指定所用的入参（必须匹配实体的字段），以及是否必需，默认值等。当你使用隐式的实体自动服务定义时，它将决定服务调用传递的参数行为。上面的案例中没有传递 **exampleId** 参数，这是 moqui.example.Example 实体的主键字段，所以服务自动为这个字段生成一个序列 ID 并将其作为出参返回回来。

对于 create 操作，服务将额外的自动生成缺省的主键序列 ID。如果实体有两个主键并

且一个指定另一个缺省，那么服务同样将生成一个次要主键序列 ID。这里还有一个特殊的服务行为，如果这里有 **fromDate** 主键字段未传递，那么服务将用当前时间戳（now Timestamp）去填充这个字段。

update 模式用于传递所有的主键字段（这是必须的）以及任何期望的非主键字段。特殊的服务行为同样适用于 update 模式。如果实体有一个 **statusId** 字段并且传入的 statusId 参数和字段值不同，那么服务将自动返回原始（DB 中）值放在 oldStatusId 出参中。无论何时实体有一个 **statusId** 字段，服务都还将返回一个 boolean 类型的 statusChanged 参数，如果状态字段和数据库中原始值不同，那么该字段为 true，反之为 false。实体自动服务还会通过检查匹配的 moqui.basic.StatusFlowTransition 存在记录，去执行有效的状态变化。如果没有有效的状态变化，那么服务将返回一个错误。

添加自定义的服务运行器

添加自定义的服务运行器并添加自己的服务类型，你可以实现 org.moqui.impl.service.ServiceRunner 接口，并在 Moqui XML 配置文件中添加一个 service-facade.service-type 元素节点来达到这个目的。

这个 ServiceRunner 接口有 3 个方法需要去实现：

```
ServiceRunner    init( ServiceFacadeImpl sfi );
Map<String, Object>    runService( ServiceDefinition sd,
            Map<String, Object> parameters ) throws ServiceException;
void    destroy( );
```

这里有一个源于 MoquiDefaultConf.xml 文件的 service-facade.service-type 元素配置案例：

```
<service-type    name="script"
        runner-class="org.moqui.impl.service.runner.ScriptServiceRunner"/>
```

service-type.**name** 属性值匹配 service.**type** 属性值，并且 **runner-class** 属性值仅仅就是 ServiceRunner 接口的实现类。

服务调用和任务调度

通过服务门面 ServiceFacade（ec.**getService**()方法或 Groovy 中的 ec.service 对象）这里有 DSL 风格的可用接口，它们有各种可选的可用方式去调用一个服务。所有的这些服务调用接口都有一个 **name**()方法用于指定调用的服务名称，**parameter**()以及 **parameters**()方法用于指定服务的入参。这些和其他各种接口上的方法都返回一个它们自身的实例，以便于进行链式调用。大部分都有一个变种的 **call**( )方法去实际的调用服务。

举个例子：

```
Map ahp = [visitId:ec.user.visitId, artifactType:artifactType, ...]
ec.service.async( ).name( "create", "moqui.server.ArtifactHit" )
        .parameters( ahp ).call( )
Map result = ec.service.sync( ).name( "org.moqui.impl.UserServices.create#UserAccount" )
        .parameters( params ).call( )
```

第一个服务调用是一个隐式定义的实体 CrUD 服务去异步的新建一条 ArtifactHit 记录。注意一下，**async**( )和 **call**( )方法没有返回任何东西，并且这种情况下会忽略服务调用的结果。第二个服务调用是同步的调用一个入参为 Map 参数的服务，并且由于是 **sync**( )方法同步调

用，所以 call( )方法返回了一个服务调用的 Map 结果。

除了每个用于不同服务调用方式的接口中的这些基本方法，还有一些可用的选项，包括：

- **sync():** 同步的调用服务并返回结果。
  - **requireNewTransaction**(boolean requireNewTransaction): 如果为 true，则延缓/重新开始当前事务（如果有一个活跃的事务），并且在这个服务调用范围内开启一个新的事务。
  - **multi**(boolean mlt): 如果为 true，表示期望在单个 map 中传递多个参数集合。每个集合的命名是名称后面带下划线的后缀和行数，例如“userId_8”标识第 8 行的 userId 参数。
  - **disableAuthz**(): 在服务调用时，对当前线程禁用认证。
- **async():** 异步的服务调用并忽略结果，返回一个 ServiceResultWaiter 对象去等待结果，或者传入一个 ServiceResultReceiver 接口的实现在服务完成时接收结果。
  - **maxRetry**(int maxRetry): 设置服务运行发生错误时最大的服务重试次数。
  - **resultReceiver**(ServiceResultReceiver resultReceiver): 指定 ServiceResultReceiver 接口实现的对象用于服务调用。随后使用 call()方法去实际调用服务。
  - **callWaiter**(): 调用服务（比如 call()方法）并返回一个 ServiceResultWaiter 的实例，用于等待和接收服务结果。
- **schedule():** 将一个或多个服务调用列入到一个计划任务中。
  - **jobName**(String jobName): 任务的名称。如果重复指定相同的任务名称（jobName），则将使用相同的潜在任务。
  - **startTime**(long startTime): 服务第一次运行的时间（时间点以毫秒为单位）。
  - **count**(int count): 服务重复运行次数。
  - **endTime**(long endTime): 服务计划任务的期满时间（时间点以毫秒为单位）。
  - **interval**(int interval, TimeUnit intervalUnit): 时间间隔用于指定服务多久运行一次。intervalUnit 参数是来源于 ServiceCall.IntervalUnit { SECONDS, MINUTES, HOURS, DAYS, WEEKS, MONTHS, YEARS }方法的枚举类型值。
  - **cron**(String cronString): cron 使用的一个相同格式化的字符串，去定义一个循环操作。
  - **maxRetry**(int maxRetry): 服务重试运行的最大次数。
- **special():** 在当前事务已提交（使用 **registerOnCommit**()方法）或者已回滚（使用 **registerOnRollback**()方法）时，注册当前服务将被调用。这个接口没有 **call**( )方法。

异步计划任务的调用运行使用 Quartz Scheduler。直接的使用 Quartz 工具，可以通过使用 ec.**getServices**( ).**getScheduler**( )方法来获得一个 org.quartz.Scheduler 对象的实例。关于你能用 Quartz 做什么，详见文档：http://quartz-scheduler.org/documentation

Quartz 任务默认存储在内存中，并且可以使用 Quartz JDBC 任务存储将其放入数据库中，或者使用 Moqui 框架的 EntityJobStore，它为了简单的配置和部署目的，使用实体门面去进行持久化。当使用 RAM 任务存储或确保一个明确的任务被安排计划了，可以使用 Quartz 中的 XMLSchedulingDataProcessorPlugin，并将其配置到 quartz.properties 文件中。部分配置是任务设置 XML 文件的文件名，Moqui 框架中默认是 quartz_data.xml 文件。

这里有一个 Moqui 框架当前默认的计划任务案例：

```
<schedule>
    <job>
        <name>clean_ArtifactData_single</name>
        <group>org.moqui.impl.ServerServices.clean#ArtifactData</group>
        <job-class>org.moqui.impl.service.ServiceQuartzJob</job-class>
        <job-data-map><entry><key>daysToKeep</key><value>90</value>
            </entry></job-data-map>
    </job>
    <trigger>
        <cron>
            <name>clean_ArtifactData_daily</name>
            <group>ServerServices</group>
            <job-name>clean_ArtifactData_single</job-name>
            <job-group>org.moqui.impl.ServerServices.clean#ArtifactData
                </job-group>
            <!-- trigger every night at 2:00 am -->
            <cron-expression>0 0 2 * * ?</cron-expression>
            <!-- for testing, run every 2 minutes:
                <cron-expression>0 0/2 * * * ?</cron-expression> -->
        </cron>
    </trigger>
</schedule>
```

这里最重要的元素是 job.job-class，它应当设置为 org.moqui.impl.service.ServiceQuartzJob Moqui 服务门面任务，并且 job.group 是服务的名称。注意 trigger.job-name 必须匹配 job.name，trigger.job-group 必须匹配 job.group。

Moqui 框架默认运行时中自带的工具应用有一些界面可用于查看，暂停，重新开始以及取消 Quartz 任务。界面中包含了计划明细的概要，任务运行的历史，以及管理当前任务和触发器。这些界面在“工具=>服务=>计划界面”中。

服务的 ECA 规则

ECA（event-condition-action）规则是一种基于事件，有条件的运行动作的专业类型的规则。对于服务 ECA（SECA）规则来说，事件是执行服务的各种阶段，并且它们能被所有的服务调用所触发。

服务 ECAs 意味着用于触发业务过程并且扩展那些已存在的，但是你不想也不能修改的服务的功能。服务 ECAs 通常不应用于维护其他实体而来的数据，实体 ECA 规则这种情况下是更好的工具选择。

这里有个 SECA 规则的案例，来源于地幔业务构件里的 AccountingInvoice.secas.xml 文件，它调用一个服务在装运被打包完成时去新建订单的发票：

```
<seca  service="update#mantle.shipment.Shipment"  when="post-service">
    <condition><expression>
        statusChanged &amp;&amp; statusId == 'ShipPacked'
    </expression></condition>
    <actions><service-call
        name="mantle.account.InvoiceServices.create#SalesShipmentInvoices"
        in-map="context + [statusId:'InvoiceFinalized']"/></actions>
</seca>
```

seca 元素上的必要属性是带有服务名称的 **service** 属性，以及表示服务调用中某个阶段的 **when** 属性。这两个属性一起构成了触发 SECA 规则的事件。这里还有个 **run-on-error** 属性默认为 false，如果设置为 true，那么即使服务调用中有错误出现，SECA 规则也将被触发。

when 属性的选项包括有：

- ✧ pre-auth: 运行在认证和授权校验之前，但是在使用 authUsername,authPassword 以及 authTenantId 参数和指定用户进行登录之后；对于任何与认证及授权相关的自定义行为来说十分有用
- ✧ pre-validate: 运行在入参校验之前；对于在校验和数据类型转换之前进行参数新增和修改来说有用
- ✧ pre-service: 在服务自身运行前运行；在服务运行之前，完成普通事情的最好位置
- ✧ post-service: 刚好在服务运行之后运行；在服务运行之后，完成普通事情的最好位置，并且有独立的事务
- ✧ post-commit: 刚好在提交即将完成后运行，不管其是否真的完成或者没有（依赖于服务设置以及当前 TX 的存在情况等）；使用 tx-commit 选项在实际的提交上面去运行一些东西
- ✧ tx-commit: 在服务运行所在的事务成功提交时运行。在服务运行之后获得其数据，这样将得到服务的输出/结果，就像入参一样
- ✧ tx-rollback: 在服务运行所在的事务已回滚时运行。在服务运行之后获得其数据，这样将得到服务的输出/结果，就像入参一样

当动作运行时，不管服务运行在什么样的上下文中，为了方便起见，加上服务的入参以供使用。如果 **when** 是在服务自身运行前，那么这里将会有一个名为 parameters 的上下文字段，其内包含了你可以按需修改的 ECA 动作入参 Map。如果 **when** 是在服务自身运行后，那么 parameters 字段将包含入参和一个 results 字段，results 字段中包含了同样可能被修改的出参（结果）。

condition 元素和在 XML 动作中的约束条件用法相同，并且可能包含 expression 以及 compare 元素，根据需要可以结合 or,and 和 not 元素一起使用。

actions 元素和服务定义，界面，表单等之中的 actions 元素用法相同。它包含了一个 XML 动作脚本。查看 ***Overview of XML Actions*** 章节获得更多的信息。

XML 动作综述

xml-actions-${version}.xsd 文件已经是个详细注释的文档，本章节只是一个可用的概览介绍，帮助你起步。你可以通过很多优秀的 XML 编辑器（包括较好的 Java IDEs 或者 IDE 插件）去查看注释，也可以直接查看 XSD 文件，或者查看 moqui.org 网站上通过 XSD 文件生成的 PDF。

下面是个最重要的，需要掌握的 XML 动作元素概要：

set	设置一个字段 **field**，既可以来源于（**from**）另一个字段也可以来源于一个 **value** 数值，可选的指定 **type** 类型，**default-value** 默认值以及空时是否设置 **set-if-empty**
if	有条件的运行直接位于 if 元素下的元素部分，或者 if.then 元素内的元素。约束条件可以放在 if.**condition** 属性中或者位于 if.condition 元素下的 compare 以及 expression 元素中（结合 and 或 or 元素使用，用 not 元素表示否定）。交替的动作间使用 else-if 和 else 子元素。
while	重复子元素直到条件为真。就像 if 元素一样，约束条件可以在 if.**condition** 属性内，或者也可以在 if.condition 元素中
iterate	在给定的 **list** 属性中遍历元素，并使用 **entry** 属性指定的名字在上下文中创建一个字段。如果 **list** 属性命名的字段是一个 Map 类型，遍历所有的 map 条目，并且将每个条目的键值使用 **key** 属性指定的名称命名，存放在上下文中。同样创建上下文字段${**entry**}_index 和${**entry**}_has_next。

script	运行本元素下通过 **location** 指定的任意资源门面能够运行的各种脚本或者 Groovy 脚本文本（内联脚本）。
service-call	调用 **name** 属性指定的服务，在 **in-map** 属性（这是个 Groovy 表达式，所以可以为内联 Map 使用中括号[]语法）或 field-map 子元素中使用输入参数，在 **out-map** 属性中放置输出参数。可选属性有 **async** 和 **include-user-login**。如果服务运行出错，则将立即返回简单的方法，除非 **ignore-error** 属性等于 true。
entity-find-one	为 **entity-name** 指定的实体查找单条记录并将其放入到一个 EntityValue 对象中。在 **value-field** 中可以使用的属性包括 **auto-field-map**, **cache** 和 **for-update**，子元素包括 field-map 和 select-field。
entity-find	为 **entity-name** 指定的实体查找多条记录并将其放入到一个 EntityList 对象中。在 list 中可以使用的属性包括 **cache**, **for-update**, **distinct**, **offset** 和 **limit**，子元素包括 search-form-inputs, date-filter, econdition, econditions, econdition-object, having-econditions, select-field, order-by, limit-range, limit-view 和 use-iterator。
entity-find-count	查找匹配给定条件的记录总数。条件和其他应用选项遵循 entity-find 操作相同的结构。
entity-make-value	对于给定的实体名 **entity-name**，新建一条 **value-field** 实体值对象，并可选的基于 **map** 去设置字段。
entity-create	为 **value-field** 实体值新建（**or-update**）一条记录。
entity-update	为 **value-field** 实体值更新一条记录。
entity-delete	删除相应的 **value-field** 实体值的记录。
entity-set	在 EntityValue 对象上设置来源于 **map**（默认为 context）的 **value-field** 字段包含 **include**（pk, nonpk 或 all），可选 **prefix** 以及 **set-if-empty**。
entity-sequenced-id-primary	对于单一主键字段的实体 **value-field**，使用一个序列值（序列名称是实体的全称）填充主键字段。
entity-sequenced-id-secondary	对于双主键字段的实体 **value-field** 并且其中一个字段已经被填充，使用一个次要主键填充另外一个字段。次要主键的值为已存在的、匹配设置字段的记录中最高次要主键值加 1。
entity-data	对于给定的 **mode**，加载（load）或 asset 指定位置 **location** 的实体门面 XML 文件。
filter-map-list	过滤 **list** 列表，并如果指定的话，将结果放入 **to-list** 中;如果未指定，则返回到 **list** 中。使用一个或多个 field-map 或 date-filter 子元素去指定如何过滤列表。
order-map-list	按照 order-by 子元素指定的字段，去对一个 Map 对象的 **list** 列表进行排序。
message	如果 **error**=true，添加 message 元素下的文本到消息门面（Message Facade）的错误列表中；否则添加到消息列表中。
check-errors	检查消息门面（Message Facade）的错误信息列表（ec.message.errors），如果列表非空则返回一条错误信息，否则啥都不用做。
return	立即返回。可选操作：指定一个 **message**，如果 **error**=true，添加到消息门面错误列表中；否则添加到消息列表中。
log	使用指定的级别（**level**）去记录消息（**message**）。

七、 用户界面

在 Moqui 框架中，创建用户界面主要的构件就是界面 XML。

界面 XML 被设计成使用相同的界面定义，用于多种渲染模式。它包含了为用户和系统展现提供了各种类型的文本输出，以及代码驱动的客户端用户界面。

为了适应这种设计目标，大部分的界面元素是展现模式不可知论的。对于特定渲染模式的指定元素，这里有 render-mode 元素和其子元素被设计用来特定渲染模式。为了在同一个界面中支持多种渲染模式指定的元素，仅仅需要在 render-mode 元素下为每种期望的类型放置一个子元素即可。

在基于 Web 的应用中，界面 XML 是为传入的请求生成输出的主要途径。界面的结构使得我们可以很容易的去支持各种界面 URL。

XML 界面

Moqui 中界面的组织有两种途径：

- ✧ 每个界面都存在层次结构的子界面
- ✧ 界面之间类似图中的节点一样通过跳转进行绑定

界面中引用了层次结构的模型，并且在 URL 中通过路径去指定渲染层次结构中的哪个界面。界面中还包含了链接，通过转换链到其他的界面（就是超链接或者表单提交）上，这更像图中的从一个节点到另一个节点的结构了。

子界面

子界面的层次结构主要用于动态的包含另一个界面或一个子界面。界面中的子界面还可以用于设置菜单。当一个界面被渲染，它将伴随着一个根界面和一列界面名称的完成。

每个 web 应用（webapp）都会在 Moqui XML 配置文件中，使用 moqui-conf.webapp-list.webapp.root-screen 元素去配置其根界面。多个根界面基于一个主机（hostname）模式被每个 web 应用配置，为单个 web 应用提供一个方便的虚拟主机的手段。注意在 MoquiDefaultConf.xml 文件中这里没有指定根界面，所以它需要在 runtime 里指定的配置文件中进行规定。

你至少要有一个容器 root-screen 元素，这意味着 **host** 设置成".*"形式的正则表单式。查看简单的 runtime 配置文件，比如 MoquiDevConf.xml 文件案例。

如果子页面名称的列表中没有叶子界面（里面没有子界面），那么使用 screen.subscreens.**default-item** 元素指定的默认界面。有鉴于此，任何带有子界面的界面都应该有一个默认界面。

这里有三种添加子界面的方式：

1. 在单个应用中的界面，通过目录结构添加：在父界面所在目录下新建一个目录，并使用和父界面的文件名相同的名称去命名这个目录，然后将界面 XML 文件都放在这个目录下（**name**=文件名加上.xml，**title**=screen.**default-title**，**location**=父界面的位置减去父文件名，加上目录名和子界面的文件名）。
2. 对于嵌入其他应用的部分界面，或共享的不存在于任何应用中的界面，使用 screen.subscreens 元素下面的 subscreens-item 元素。

3. 对于添加界面，移除界面，或对一个独立的应用改变界面的次序和标题，添加一条 moqui.screen.SubscreensItem 实体的记录。

对于#1 的方式，一个目录结构应该看起来如下所示（来源于 Example 应用）：

- ExampleApp.xml
- ExampleApp
  - Feature.xml
  - Feature
    - FindExampleFeature.xml
    - EditExampleFeature.xml
  - Example.xml
  - Example
    - FindExample.xml
    - EditExample.xml

这个模式需要注意的一点是：如果这里有子界面，那么这里应该有一个目录使用和界面 XML 文件相同的名称，仅仅没有.xml 的扩展名。Feature.xml 文件就是一个有子界面的界面例子，反之 FindExampleFeature.xml 没有子界面（它是界面层级结构的一个叶子）。

对于#2 的方式，subscreens-item 元素应该看起来类似于 apps.xml 文件里使用这个元素去挂载 Example 应用的根界面一样：

```
<subscreens-item  name="example"  menu-title="Example"  menu-index="8"
    location="component://example/screen/ExampleApp.xml"/>
```

对于#3 的方式，数据库中 moqui.screen.SubscreensItem 实体的记录应该看起来如下所示（上面 XML 元素的一个改写版）：

```
<moqui.screen.SubscreensItem  subscreenName="example"
    userGroupId="ALL_USERS"
    menuTitle="Example"  menuIndex="8"  menuInclude="Y"
    screenLocation="component://webroot/screen/webroot/apps.xml"
    subscreenLocation="component://example/screen/ExampleApp.xml"/>
```

在你的界面部分里的部件（可视元素）中，你可以使用 subscreens-active 元素去指定渲染激活的子界面。你同样可以使用 subscreens-menu 元素去指定在哪里去渲染所有的子元素菜单。对于单个元素去实现上面两种情况并有一个默认的布局，使用 subscreens-panel 元素。

虽然界面的全路径总是必须明确的，但是当每个界面下的默认子界面项有多个为默认的，那么这里会有一个条件，使得只有一个会成为默认的显示。在 webroot.xml 界面中，有一个为 iPad 提供的默认替代的子界面的例子：

```
<subscreens  default-item="apps">
    <conditional-default  item="ipad"
     condition="(ec.web.request.getHeader('User-Agent')?:'').matches('.*iPad.*')"/>
</subscreens>
```

在适当的时候，一个明确的界面路径将会即可用于“apps”子界面又可用于”ipad”子界面，但是如果两者都没有明确路径并且用户代理匹配，则将会默认使用 ipad.xml 子界面，否则将会默认使用普通的 apps.xml 子界面。这两者都有案例并且它们之下的工具界面有层次结构，但是为适应不同的平台有些微 HTML 和 CSS 的区别。

一旦一个界面，类似于 FindExample 界面，通过这两种方式之一进行呈现，其链接将会保存在源于关联的界面路径生成的 URLs 里的基本界面路径中，所以用户将会停留在原始默

认指定的路径上。

独立的界面

通常的，界面都将起始于根界面，依据渲染路径进行呈现。每个界面按照这种方式可以添加到输出中。更进一步，一个界面呈现时，其路径中无任何前置的界面并被添加到输出中，这就是一个“独立的”界面。

这十分的有用，当你想控制一个界面的所有输出，并且不使用源于子界面层级结构下的界面的标题，菜单，页脚等。

这里有两种方式去让界面独立：

- 设置 screen.**standalone** 属性为 true，可以令界面总是独立的
- 通过传入参数 **lastStandalone**=true 去渲染独立的界面，或者将其设置在界面预动作中（screen.pre-actions 元素下的动作）

第一种选项对于一个分离的应用的根，并且需要不同展现的界面来说最有用。第二种选项对于那些有时用于一个应用的上下文环境，有时用于生成简朴的输出，例如 CSV 文件，或者在一个对话框窗口或界面部分动态加载的界面来说最有用。

转换

转换被定义为界面的一部分，而且是你如何从一个界面流转到另一个界面，通过这种方式可以处理输入。转换当然可以流转回到相同的界面，并且处理输入时经常这么做。

转换的逻辑（转换动作）应当只用于处理输入，并不应为展现去准备数据。那是界面动作的工作，相反的，界面动作不应被用于处理输入（下面将详细描述）。

当一个界面 XML 运行于一个 web 应用时，转换的请求跟在界面的 URL 之后。在任何的上下文环境中，转换是子界面路径元素列表的最后一个条目。举个例子，第一个路径是前往 EditExample 界面，第二个是前往这个界面中的 **updateExample** 转换：

/apps/example/Example/EditExample

/apps/example/Example/EditExample/updateExample

当一个 HTTP 请求的目标是一个转换，那么所有这个转换相关联的动作都将会运行，然后一个重定向将会被发送到 HTTP 客户端（通常是个 web 浏览器），要求其跳转到这个转换指定的界面 URL 上。如果转换没有逻辑和指向到另一个界面或外部 URL，那么当一个链接为这个转换生成时，将会自动的跳转到其他的界面或外部 URL，并且略过整个转换。注意一下，这些点只适应于界面 XML 运行在一个基于 web 的应用中。

一个简单的从一个界面到另一个界面的转换，本例是从 FindExample 到 EditExample，如下所示：

```
<transition   name="editExample">
    <default-response   url="../EditExample"/>
</transition>
```

url 属性中的路径基于这两个位于同一个父界面下的兄弟界面的位置。在这个属性中，一个简单的点号（"."）参照的是当前界面，两个点号（".."）参照的是父级界面，遵循 Unix 文件路径相同的模式。

对于界面来说，输入处理最好的模式就是使用转换去调用单个服务。通过这种方式，服务被定义去适应由关联的转换所提交的界面。这就令两者的设计更为清晰，同时也提供了其

他的效益， 例如服务定义上的一些校验可被用于生成匹配客户端层面的校验。这类转换应当如下所示（EditExample 界面上的 **updateExample** 转换）：

```
<transition  name="updateExample">
    <service-call  name="org.moqui.example.ExampleServices.updateExample"/>
    <default-response  url="."/>
</transition>
```

在这个例子中，default-response.**url** 属性是一个简单的参照当前界面的点号，这意味着转换处理完成之后，它将会跳转回当前界面。

界面转换同样可以用动作替代单个服务的调用，通过使用 actions 元素去替换 service-call 元素。就像 Moqui 里所有 XML 文件中使用的 actions 元素一样，其子元素都是标准的 Moqui XML 动作并会转化为 Groovy 脚本。下面就是一个使用动作的界面转换的样例（简化的例子，同样来源于 EditExample 界面）：

```
<transition  name="getExampleTypeEnumList">
    <actions>
        <entity-find  entity-name="..."  list="...">
            <econdition  field-name="..."  from="..."/>
            <order-by  field-name="..."/>
        </entity-find>
        <script>
        ec.web.sendJsonResponse( [exampleTypeEnumList:exampleTypeEnumList] )
        </script>
    </actions>
    <default-response  type="none"/>
</transition>
```

这个例子同样展示了你如何去做一个简单的实体查找操作，并返回一个 JSON 格式的响应结果到 HTTP 客户端上。注意一下这里 ec.web.**sendJsonResponse** ()方法的调用，以及 default-response.**type** 属性的 none 值，这些告诉框架不要去处理任何额外的响应。

就像暗指的 default-response 元素，你也可以通过 conditional-response 元素有条件的选择一个响应。这个元素是可选的，同时尽管你可以指定任意条，但你应该总是至少指定一个 default-response 元素，用于在没有任何条件满足的情况。这里还有个可选的 error-response 元素，你可以用其指定在转换动作出现错误的情况下，返回的响应。

这里有个来源于 DataExport 界面，带有 conditional-response 元素的简化转换的案例，如下所示：

```
<transition  name="EntityExport.xml">
    <actions>
        <script><![ CDATA[ if (...)  noResponse = true ] ]> </script>
    </actions>
    <conditional-response  type="none">
        <condition><expression>noResponse</expression></condition>
    </conditional-response>
    <default-response  url="."/>
</transition>
```

这里允许脚本指定无响应应该被发送（当其发送回来的是数据导出），否则将会跳转回到当前界面。注意在 condition. expression 元素下是个简单的 Groovy 表达式，将会作为一个

boolean 类型去计算。

当重定向 url 或者激活到其他的目标界面时，所有的*-response 元素都可以使用子元素参数。由于每个界面都有一个预期的参数列表，所以仅当你需要覆盖传入的参数值（默认定义于界面下的参数标签中）或者传递额外的参数时，这种方式才是必须的。

对于 default-response，conditional-response 和 error-response 元素，这里有些共享的属性：

type	默认为 url，可以是： ✧ none: 无响应，除了转换动作之外不做任何事情。 ✧ screen-last: 按照最后一个请求跳转到一个界面，除非之前的请求已保存了一个跳转的界面（使用 **save-current-screen** 属性，完成自动登录）。如果没找到最后的界面那么将使用 url 中的值，同时如果什么都没有则将会前往默认的界面（不管每个子界面做了什么默认的设置，就是跳到根界面）。 ✧ screen-last-noparam: 类似 screen-last，但是不传递任何参数。 ✧ url: 按照 **url-type** 类型的 **url** 属性中的 URL 值进行重定向。
url	基于 **url-type** 类型，作为响应的 URL。默认 **url-type** 类型的是 screen-path，意味着这里的值是一个从当前界面到期望的界面，转换或者子界面内容的路径。 使用"."表示当前界面，同时".."表示运行时界面路径中的父界面。".."可以被使用多次，例如"../.."是获取父界面的父界面（祖父界面）。如果界面路径类型的 url 起始于"/"，那么它的路径是相对于根界面而言，而不是当前界面。 如果 **url-type** 是 plain，那么这里可以为任何有效 URL（相对于当前域或者绝对路径）。
url-type	可以为 screen-path（默认值）或 plain。普通响应都是跳转到另外一个界面，所以是默认值。但如果你想要跳转到一个相对或者绝对路径的 URL 就要使用 plain 类型。
parameter-map	就像 parameter 参数的子元素，可以用于指定通过重定向传递的参数。
save-current-screen	保存当前界面的路径和参数为了将来使用，一般用于 screen-last **type** 类型的响应。
save-parameters	在做重定向之前保存当前的参数（以及请求的属性值），这样界面呈现在重定向渲染之后并在同一个上下文中，类似于原始请求到转换。

参数和 Web 设置

界面定义首要的事情之一就是传递到界面的参数。它用于构建一个 URL 链接到界面或者准备界面呈现的上下文的时候。你可以使用 parameter 元素去定义参数，通常看起来如下所示：

```
<parameter name="exampleId"/>
```

name 属性是唯一必需项，这里还有其他的属性比如你要一个静态的默认值（使用 **value** 属性），或者默认获得上下文中的一个匹配参数名的字段（通过 **from** 属性）。

虽然参数适用于所有的渲染模式，但是仅当界面呈现是在一个基于 web 的应用中，这里才有适用的明确设置。这些存在于 screen.web-settings 元素上的可选项包括：

- ✧ **allow-web-request**: 默认为 true。设置为 false 表明不允许 HTTP 客户端进行访问。
- ✧ **require-encryption**: 默认为 true。设置为 false 表明界面更不安全并且无需加密（例如 HTTPS）。
- ✧ **mime-type**: 默认为 text/html。该选项可基于界面如何进行渲染（渲染模式）来发生变化，但

是当总是生成一种明确的输出时，这里才会设置关联的 MIME 类型。

✧ **character-encoding**: 默认使用 UTF-8 进行文本输出。如果你要使用不同的编码进行文本呈现，这里设置它即可。

界面动作，前处理动作和持续动作

在界面可视元素（widgets 界面部件）渲染之前，数据准备使用 screen.actions 元素下的 XML 动作进行完成。这些 XML 动作和逻辑及服务章节中描述的服务以及其他工具使用的 XML 动作是相同的。其中的元素用于运行服务和脚本（内联的 Groovy 或任意类型的通过资源门面支持的脚本），进行基本的实体和数据活动操作等等。界面动作应该仅被用于为输出准备数据。使用转换动作去处理输入。

当界面按照顺序进行渲染完成时，动作会在界面路径中被发现，并为每个界面类似于界面按照列表进行渲染一样去运行。在路径中的第一个界面呈现前运行动作使用 pre-actions 元素。它主要用于准备将被当前界面（例如，在界面路径中当前界面之前运行动作）包含的，界面所需的数据。使用它时需记住：一个界面可以在不同的场景下被不同的界面所包含。

如果你想要动作在界面呈现前并且在任何转换执行前运行，那么使用 always-actions 元素。always-actions 和 pre-actions 之前主要的区别是：pre-actions 仅在界面或子界面呈现之前运行，而 always-actions 将会在任何当前界面中的转换以及任何子界面的转换之前运行。always-actions 还可以不管界面是否将呈现都会运行，然而 pre-actions 只会运行在界面即将呈现之前（例如，在界面路径中的一个独立界面下面运行）。

XML 界面组件

screen. widgets 元素下的元素是用于呈现的元素，或生成文本时实际生成输出文本的元素。最常用的部件是 XML 表单（使用 form-single 和 form-list 元素）以及内嵌的模板。XML 表单详见下面的章节内容。

虽然 XML 表单并未指定任何的呈现模式，但是模板的性质却特别的决定了一种呈现模式。这意味着要支持多种类型的输出，你就要有多个模板。webroot.xml 界面中有个为不同呈现模式包含多模板的例子：

```
<render-mode>
    <text   type="html"
            location="component://webroot/screen/includes/Header.html.ftl"/>
    <text   type="xsl-fo"   no-boundary-comment="true"
            location="component://webroot/screen/includes/Header.xsl-fo.ftl"/>
</render-mode>
```

这个例子中还包含了一个使用内联文本支持多种渲染模式的例子：

```
<render-mode>
    <text type="html"> < ![ CDATA[ </body></html> ] ] > </text>
    <text type="xsl-fo">
        < ![ CDATA[ </fo:flow></fo:page-sequence></fo:root> ] ] ></text>
</render-mode>
```

下面是用于基本显示的部件元素：

- **link**: 一个超链接到转换，其他界面或者任意 URL
- **image**: 显示图片
- **label**: 显示文本

组织界面结构使用这些部件元素：

- section: 带有条件，动作，部件以及失败部件（当条件计算失败时运行）的一个已命名的界面部分
- section-iterate: 类似于 section，但是为一个集合中的每个条目都运行
- container: 界面中的一个区域
- container-panel: 由一个页眉，页脚，左边，中间和右边面板组成的界面中的一个区域
- container-dialog: 一个初始化时隐藏的界面区域，同时当按钮点击时弹出
- dynamic-dialog: 一个按钮和弹出框的占位符，通过当前界面的转换去从服务中加载它的内容
- include-screen: 同字面的意思，包含另一个界面

截面，条件和失败时部件

截面是一个特殊的部件包含其他的小部件。它可用于其他任何界面部件元素作用的地方。截面有 widgets，condition 和 fail-widgets 子元素。screen 元素同样支持这些子元素，它们令其成为界面中一种顶层的截面。

condition 元素用于指定一个条件。如果计算结果是 true，widgets 元素下的部件将会被呈现；如果结果是 false，fail-widgets 元素下的部件将会被渲染。

宏模板和自定义元素

Moqui 界面 XML 和 XML 表单中的字段使用 Freemarker（FTL）模板文件中的宏定义集合去转变为期望的输出。这里为每个 XML 元素在界面呈现时生成对应输出都有一个宏定义。

这里有两种途径为呈现界面去指定宏定义模板：

- **为所有的界面：** Moqui XML 配置文件中 moqui-conf.screen-facade.screen-text-output.**macro-template-location** 属性；这里有个 screen-text-output 元素通过 screen-text-output.**type** 属性定义，用于每种展现模式（例如 html，xml，csv，xsl-fo 等等）
- **为单个界面：** screen.macro-template.**location** 属性；你同样可以为每种展现模式去指定通过 macro-template.**type** 属性定义的 macro-template 元素

宏定义模板的位置可以是任何资源门面支持的路径位置。你最常用的位置类型包括组件，内容以及运行时目录位置。

MoquiDefaultConf.xml 文件指定了 Moqui 中默认包含的宏定义模板以及其他默认设置。你可以在运行时中的 Moqui XML 配置文件中指定自己的配置去覆盖它。

当你使用一个自定义的宏定义模板文件时，你无需为每个你想要不同展现的元素去包含一个宏定义。你可以开始的时候使用一个默认的宏定义或其他你想使用的模板文件，然后为期望的元素去覆盖宏定义即可。在你自己的文件中包含另一个宏定义如下所示：

```
<#include "classpath://template/DefaultScreenMacros.html.ftl"/>
```

这里的位置可以是任何资源门面支持的路径位置。

你可以使用这种方式去添加你自己的自定义元素。换句话说，你的自定义宏模板文件中的宏定义并非要覆盖 Moqui 中的元素，它可以是任何你想要的东西。

使用这种方式，你可以添加你自己的部件元素和表单字段类型，以便在你的应用中统一访问界面的交互方式。举个例子，你可以为特定的包含动态 HTML 的容器，类似默认宏定义中的对话框添加宏定义，或者一个特定的表单字段类似滑动块，或者一个使用自定义的 JavaScript 生成的表单字段部件。

当你为一个自定义元素添加一个宏定义时，哪怕它们没有通过 XSD 文件的校验，你也可以开始使用它。如果你想要它们通过校验：

1. 创建你自己的自定义 XSD 文件
2. 包含一个或多个默认的 Moqui XSD 文件
3. 在自定义 XSD 中添加你自己的元素定义
4. 在你的界面 XML 文件中 screen.**xsi:noNamespaceSchemaLocation** 属性里引用你的自定义 XSD 文件

CSV，XML，PDF 和其他的界面输出

由于单个界面 XML 文件能够支持多种呈现模式的输出，呈现模式通过界面的参数去选择：**renderMode** 参数。对于基于 web 的应用来说，它可以是一个 URL 参数。对于任何应用来说，它可以在一个界面动作中进行设置，通常是一个预处理动作（例如在 screen.pre-actions 元素下）。

这个参数的值可以为任何匹配 Moqui XML 配置文件中 screen-text-output.**type** 属性类型的字符串。开箱即用的类型和你在运行时配置文件中添加的一样。

渲染路径中所有界面的呈现都不管呈现模式，所以对于你仅想在路径中最后一个界面的内容输出类型（比如 CSV）来说，使用 **lastStandalone**=true 参数和 **renderMode** 参数与一起使用。

XML 表单

XML 表单有两种类型：单个表单和列表表单。单个表单表现为单个带有标签和小部件的字段的集合。列表表单表现为像是一个字段为列的表格，表格页眉为标签，每行的字段都有个小部件，并且一行基于表单列表每个条目的输出。

然而还有其他的方式去获取数据，最常用的是从一个 Map 中获取单个表单的字段值，同时从一个 Map 的列表中获取一个列表表单的值。

XML 表单类似于界面 XML，它们都是用一个 FTL 宏定义为每个元素进行渲染，同时它们都支持多种呈现模式。就像界面 XML 部件一样，你可以通过添加宏定义去添加你自己的部件。XML 表单的宏定义存在和界面 XML 宏定义相同的 FTL 文件中，所以使用相同的方式去添加自定义宏。

表单字段

表单最主要的元素就是 field，通过其 **name** 属性进行定义。当一个表单扩展另一个表单的字段时，使用相同的字段名称进行覆盖。对于 HTML 输出这也是 HTML 表单字段的名称。这个名称还可以用于获得字段值 map 的键或者参数名称（如果没有找到 map 的键值对，或

表单提交时出现错误的情况）。从上下文中获得字段值，参数的 **name** 属性依然可以使用，同时 **entry-name** 属性可以为任何 Groovy 表达式，用于计算出期望的数值。

你可以在基于服务参数生成 HTML 时，使用字段元素上的 **validate-service** 和 **validate-parameter** 属性去自动进行客户端的校验。当表单字段使用 auto-fields-service 元素基于服务自动定义时，这两个属性都会被自动进行填充。XML 表单渲染时将会查看表单提交的转换 transition，并且如果它有单个 service-call 元素（相对于使用 actions 元素去处理输入），那么它将会查看服务的输入参数，同时如果其名称匹配字段名称则使用其校验。

字段类型或字段的“部件”（可视的/交互元素）存在于 field 元素下的子元素中。默认部件使用在 default-field 子元素中，并且所有的字段都需要有一个（并且仅有一个）。如果你想要在特定的条件下使用不同的部件，可以使用 conditional-field 元素及一个 Groovy 表单式去计算出一个 boolean 值存在 **condition** 属性中。这在单个表单和列表表单都可生效，同时对于列表表单，将会为每一行都进行计算。

这里还有部件的 field.header-field 子元素用于列表表单的行数据头。当使用这些数据头字段部件在一个独立的表单时，通常意味着为搜索操作而使用。排序链接自然的和搜索操作一起在列表表单的数据头中，并且它们可以通过设置 header-field**.show-order-by** 属性为 true 或者 case-insensitive 来开启。

字段的标题来源于 default-field**.title** 属性，除非这里有 header-field 元素，然后它将从这个元素上的 **title** 属性中获取。default-field 元素同样有 **tooltip** 属性用于当焦点在字段上或者鼠标悬停在字段上时（指定悬停依赖于生成的 HTML 或其他指定的表单呈现），显示一个弹出的提示信息。

通常当起始时间还没到或截止时间之后，将时间值标为红色比较好。可以使用 default-field**.red-when** 属性进行控制，这个属性值默认为 by-name，意味着如果字段 **name** 是起始时间 fromDate，那么当前时间还没到则这个字段就是红色，同时如果字段 **name** 是截止时间 thruDate，那么当当前时间已经超过这个时间则字段为红色。属性值还可以为 before-now，after-now 和 never。

字段组件

这里有大量的开箱即用的表单字段小部件，同时额外的小部件可以使用宏定义模板和自定义元素截面中描述的扩展机制进行添加。

任何界面中的可用部件都可以在表单 XML 字段中使用（参看 ***XML Screen Widgets*** 章节）。这里还有各种表单字段指定的部件。下面是 Moqui 中开箱即用的字段部件总结：

auto-widget-service	基于 **parameter-name** 入参和 **service-name** 服务自动的定义字段的部件。使用 **field-type** 属性去指定字段部件一般的类型，指定的字段部件是基于参数对象的类型去进行选择。它可以是 edit（默认值），find，display，find-display（同时添加 find 和 display 部件）或 hidden。
auto-widget-entity	基于 **entity-name** 实体的 **field-name** 字段去自动的定义字段部件。使用 **field-type** 属性去指定使用的字段部件一般的类型，指定的字段部件是基于字段类型进行选择的。它可以是 edit，find，display，find-display（默认值；同时添加 find 和 display 部件）或 hidden。
widget-template-include	在一个 XML 文件中使用 widget-templates 根元素去定义表单字段部件模板。每个 widget-template 元素可以包含任意按需要带有**${}**参数的字段部件元素。 使用一个部件模板需要指定它的 **location** 以及按需要 set 用于呈现模板中定义字段的

	子元素。
check	为 entity-options，list-options 和/或 option 子元素（查看 drop-down 描述中的细节）中的可选项列表显示一个选择框。可选操作：默认选中使用 **no-current-selected-key** 属性去指定一个选择框，或设置 **all-checked** 为 true 去勾选所有的框子。
date-find	和 date-time 使用一样的 **type** 和 **format** 属性，去显示两个日期/时间输入部件。使用 **default-value-from** 属性作为表单（左边的）输入框默认值，同时 **default-value-thru** 属性作为截止时间框（右边的）的值。
date-time	一个通过 **type** 指定的，可以为 timestamp，date-time，date 或 time 类型的日期/时间部件。日期/时间字符串的格式化通过 **format** 属性，使用一个 Java 的 SimpleDateFormat 字符串类型来进行指定。这个部件的文本输入框的宽度在一行中为 **size** 字符数，并允许输入字符数最大为 **maxlength**，尽管这些都是可选的并基于 **type** 自动设置了。使用 **default-value** 属性在没有上下文或参数值时，指定字段的默认值去使用。
display	一个普通的扩展的字符串文本显示，其来源于 **text** 属性（或为空时，字段的值）加上一个相应的表单提交的隐藏字段，除非 **also-hidden** 被设置为 false。使用 **format** 属性去为日期/时间（SimpleDateFormat），数字（DecimalFormat）等值去指定 Java 格式化字符串。对于币种字段进行格式化，需包含币种中 **currency-unit-field** 的 Uom.**uomId**。对于 HTML 输出文本默认进行编码，除非 **encode** 被设置为 false。
display-entity	为 **entity-name** 查找实体的值并显示包含实体字段值的扩展 **text** 字符串。它仅限于通过单一主键字段进行查找，并且如果实体的主键字段名称和 field.**name** 不同，则通过 **key-field-name** 属性去指定它。默认的这里有个缓存的查询，不使用实体缓存设置 **use-cache** 为 false。就像 display 一样，它有一个相应的表单提交的隐藏字段，除非 **also-hidden** 被设置为 false。对于 HTML 输出文本默认进行编码，除非 **encode** 被设置为 false。
drop-down	一个 **size** 设置为数字大于 1 的下拉或多值选择框。允许选择多个数值设置 **allow-multiple** 为 true。当前选中的值可以为下拉中的第一个值，设置 **current** 为 first-in-list（默认值），或者通过 selected 选项去设置选中值。设置 **allow-empty** 为 true 去添加一个空的选项到列表中。 选项列表使用 entity-options，list-options 和/或 option 子元素，或者二选一的通过 dynamic-options 元素从一个界面转换请求中去获取选项进行装配。 使用 entity-options 从数据库记录中获取选项。指定实体字段用于键/值对中的 **key** 属性，并且 **text** 属性用于字段的文本标签值。**entity-find** 元素用于查询约束的选项，和 XML 动作脚本中相同的元素使用方式一样。 使用 list-options 元素用于 Map 值的列表选项，通过 Groovy 表达式来计算获得 **list** 属性中的列表，并且 Map 中键值对选项的键值在 **key** 属性中，同时 Map 键值的文本标签值在 **text** 属性中。使用 option 元素去明确的为每个选项指定 **key** 和 **text** 属性。 界面 **transition** 返回一个包含了 Map 的 List，加上为每个 Map 的键和通过键获取文本值的 **value-field** 和 **label-field** 属性的 JSON 字符串去指定 dynamic-options。使用动态选项的主要原因是当另一个字段变化时去改变选项。需要使用一个或多个 depends-on 子元素以及 **field** 属性中的表单字段名去实现它。当一个参照的字段改变新的选项时将会请求界面跳转，将所有的参照字段值作为参数传递到请求中。 使用 **no-current-selected-key** 属性设置默认选项的键值。如果设置的选项不在已存在的选项中，使用 **current-description** 属性去指定它的描述。 默认的，动态下拉框部件基于输入的文本进行选项过滤。使用普通的下拉设置 **search** 为 false。允许用户输入一个新的不存在于下拉中并提交的选项，设置 **combo-box** 为

	true。
file	一个宽度为 **size**（默认为 30）字符并允许最大输入 **maxlength** 字符的文件上传输入框（有一个按钮/链接去弹出文件选择窗口）。如果没有上下文或这个字段没有参数值，则用 **default-value** 属性去指定默认值。
hidden	隐藏字段的值会和表单一起提交但不会显示给用户。如果没有上下文或这个字段没有参数值，则用 **default-value** 属性去指定默认值。
ignored	将这个字段视为甚至没有被定义。当扩展另一个表单排除不需要的字段时很有用。
password	一个宽度为 **size**（默认为 30）字符并允许最大输入 **maxlength** 字符的密码输入框。标记输入为安全的。
radio	为源于 entity-options，list-options 和/或 option 子元素（查看 drop-down 描述中的细节）中的可选项列表显示一组单选框。可选的使用 **no-current-selected-key** 属性去指定默认选中值（如果该字段没有值或者参数指定）。
range-find	主要用于数字范围查找，显示两个宽度为 **size**（默认为 10）字符并允许最大输入 **maxlength** 字符的小输入框。使用 **default-value-from** 属性作为表单（左边的）输入框默认值，同时 **default-value-thru** 属性作为截止（右边的）的值。
reset	重设表单的按钮。按钮的文本来源于字段标题。
submit	表单提交按钮。按钮文本来源于字段标题，除非使用了 image 子元素在按钮上放置了一个图片。可以使用 **icon** 属性去设置一个来源于图标库（默认运行时 webroot 里 Bootstrap 模板 Glyphicons 中的图标都可用，举个例子 **icon**="glyphicon glyphicon-plus"或利用"fa fa-search"来使用 Font Awesome 图标）的图标样式放在文本旁边。在 **confirmation** 属性中放置信息，可以在按钮点击时显示一条信息并让用户进行确认。
text-area	一个文本域，宽度为 **cols** 字符高度为 **rows** 行，并且允许最大输入字符为 **maxlength**。如果该字段没有上下文或参数值，可以使用 **default-value** 属性去指定一个默认值。设置 **read-only** 为 true，可以使文本域仅用于显示而不允许改变其值。
text-line	一个宽度为 **size** 字符的单行简单文本输入框，允许最大输入字符为 **maxlength**。如果该字段没有上下文或参数值，可以使用 **default-value** 属性去指定一个默认值。设置 **disabled** 为 true，可以使文本框仅用于显示而不允许改变其值。使用 **format** 属性去为日期/时间（SimpleDateFormat），数字（DecimalFormat）等值去指定 Java 格式化字符串。 通过实现一个界面转换来提供数据，并使用 **ac-transition** 属性去指定转换的名称，可以让 text-line 自动完成。这个转换应当使用一个带有 value 和 label 字段的 Map 列表 List 的 JSON 字符串（使用 ec.web.**sendJsonResponse()**）来进行响应。可选的，使用 **ac-delay** 和查询前的 **ac-min-length**（默认为 1）最小输入字符值去指定时间延时的毫秒数（默认为 300）。
text-find	类似 text-line 有 **size**，**maxlength** 和 **default-value** 属性，并同时有一个用于 **ignore-case**（默认为 true，例如选中）的选择框，以及一个为搜索操作使用 **default-operator** 属性指定默认操作符（可以为 equals，like，contains 或 empty）的下拉框。 隐藏忽略大小写选择框以及操作符下拉框（默认作为隐藏参数传递，UI 部件不可见）可以使用 **hide-options** 属性，对于是否隐藏的特性来说，选项设置有 false（默认值，显示两者），true（隐藏两者），ignore-case（只隐藏忽略大小写选择框），以及 operator（只隐藏操作符下拉框）。

单个表单

使用 form-single 元素去定义单个表单。下面是 form-single 元素的属性：

- **name**: 表单的名称。用于和界面 XML 文件位置一起去引用表单。对于 HTML 输出来说，这是表单的名称和 id，同时对于其他的输出来说，它也可被用于定义表单相关的输出部分。
- **extends**: 扩展使用井号（#）分隔位置和名称的表单。如果这里没有位置则它将被视为当前界面中的一个表单名称。
- **transition**: 当前界面提交表单跳转到的转换。
- **map**: 获取字段值的 Map。通常是一个 EntityValue 对象或从各种地方拉取的用于填充表单的 Map。Map 键匹配字段的名称。如果 field.**entry-name** 属性在每个字段被渲染的同时，用于上下文计算，它将会被忽略。默认为 fieldValues。
- **focus-field**: 当界面呈现时焦点所在的字段名称 **name**。
- **skip-start**: 表单渲染跳过的字段的起点。在一个表单之后使用 **skip-end**=true，这会有效的将多个表单组合成一个表单。
- **skip-end**: 表单渲染跳过的字段的终点。使用它去离开一个打开的表单，这样额外的表单可以和它一起进行组合。
- **dynamic**: 如果为 true，表单将被认为是动态的，并且内部定义每次都会被构建而不仅仅是第一次引用的时候去构建。当 auto-fields-*元素为服务或实体名称有**${}**字符扩展，这就十分的必要。
- **background-submit**: 在后台提交表单而无需重载界面。
- **background-reload-id**: 在后台提交表单之后，通过这个 id 去重载 dynamic-container。
- **background-message**: 在后台提交表单之后，在一个对话框中展示这个信息。

使用 form-single.field-layout 元素进行字段的布局，而不仅仅是平铺的字段列表。对于 HTML 输出来说，这里有个可选的 **id** 属性用于辅助样式。如果字段布局包含字段分组，设置 **collapsible** 属性为 true，使用一个手风琴部件去保存空间位置。可选的指定 **active** 激活的分组索引，而不是第一个分组初始化时展开。这里的子元素用于定义一个布局：

- field-ref: 指定名称为 **name** 的字段位于何处
- fields-not-referenced: 包含所有未在别处引用的字段；如果这个元素未展现在 field-layout 中没有引用的字段，则将不会被呈现
- field-row: 创建一行通过 field-ref 子元素指定的字段；如果这里行中有两个字段将会显示为四列，都带有标题；如果这里有多余两个字段只有第一个字段的标题会显示，其他的字段部件将会在行中并排的排列，按照需要进行包装
- field-group: 如果 field-layout.**collapsible** 被设置为 true，创建一个手风琴类型的字段分组，同时带有一个可选的 **title** 显示在分组上，一个可选的分组外部容器（div）的 HTML 输出样式 **style**；使用 field-ref，fields-not-referenced 和 field-row 子元素用于指定分组包含的字段以及可选的将它们放入行中

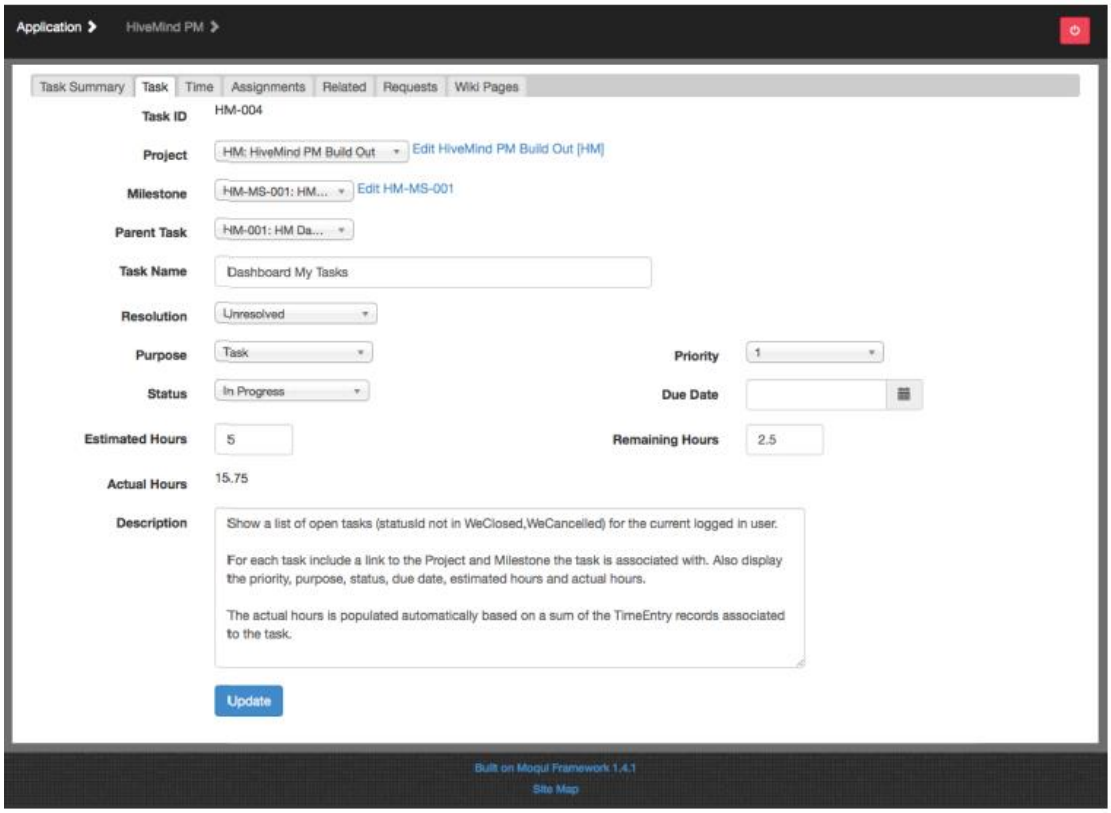
➢ *单个表单的实例*

为了更好的了解单个表单不同方面的功用，我们来看一个更复杂的例子。这个表单来源于 HiveMind 工程管理应用的任务编辑界面。

这个界面有以下的例子（参阅下方的全部代码）：

- **Project**: 一个使用 entity-options 填充的下拉框 drop-down，以及一个独立的 link 链接到任务关联的当前工程

- **Milestone** 和 **Parent Task**: 使用 dynamic-options 填充的下拉框 drop-down 字段，两者都使用 depends-on 元素去依赖于工程（**rootWorkEffortId**）
- **Task Name**: 简单的文本行 text-line 输入框
- **Resolution** 和 **Purpose**: 使用 widget-template-include 元素及 set 子元素的标准枚举下拉字段（Enumeration drop-down）；鉴于 Resolution 包含了所有的 EnumerationType 值（**enumTypeId**），意图通过父 Enumeration（**parentEnumId**）驱使去使用一个部件模板
- **Status**: 标准的状态下拉框 drop-down，其选项基于源自当前状态的转换，使用 StatusFlowTransition 实体
- **Due Date**: 简单的日期数据类型的输入框
- **Estimated Hours** 和 **Remaining Hours**: 简单的数字类型输入框
- **Actual Hours**: 一个数字格式化字符串显示 display
- **Description**: 简单的文本域 text-area



这个表单使用 field-layout 去并排的放置各种字段，否则使用默认的布局。带有 field-group 手风琴布局的案例参见 Moqui 样例应用中的样例编辑界面。

下面是这个单个表单的代码，同时为上下文环境，查看界面 XML 部分，transition 定义，数据准备的界面动作 actions 等：

```
<screen xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="http://moqui.org/xsd/xml-screen-1.4.xsd"
        default-menu-title="Task" default-menu-index="1">

  <parameter name="workEffortId"/>

  <transition name="updateTask">
    <service-call name="mantle.work.TaskServices.update#Task"
            in-map="context"/>
    <default-response url="."/>
  </transition>
  <transition name="editProject">
    <default-response url="../../Project/EditProject"/></transition>
  <transition name="milestoneSummary">
    <default-response url="../../Project/MilestoneSummary"/>
  </transition>
  <transition name="getProjectMilestones">
    <actions>
      <service-call in-map="context" out-map="context"
          name="mantle.work.ProjectServices.get#ProjectMilestones"/>
      <script>ec.web.sendJsonResponse(resultList)</script>
    </actions>
    <default-response type="none"/>
  </transition>
  <transition name="getProjectTasks">
    <actions>
      <service-call in-map="context" out-map="context"
          name="mantle.work.ProjectServices.get#ProjectTasks"/>
      <script>ec.web.sendJsonResponse(resultList)</script>
    </actions>
    <default-response type="none"/>
  </transition>
  <actions>
    <entity-find-one entity-name="mantle.work.effort.WorkEffort"
            value-field="task"/>
    <entity-find-one entity-name="mantle.work.effort.WorkEffort"
            value-field="project">
      <field-map field-name="workEffortId" from="task.rootWorkEffortId"/>
    </entity-find-one>

    <entity-find entity-name="mantle.work.effort.WorkEffortAssoc"
        list="milestoneAssocList">
      <date-filter/>
      <econdition field-name="toWorkEffortId" from="task.workEffortId"/>
      <econdition field-name="workEffortAssocTypeEnumId"
          value="WeatMilestone"/>
    </entity-find>
    <set field="milestoneAssoc" from="milestoneAssocList?.getAt(0)"/>
    <set field="statusFlowId"
        from="(task.statusFlowId ?: project.statusFlowId) ?: 'Default'"/>
  </actions>

  <widgets>
    <form-single name="EditTask" transition="updateTask" map="task">
      <field name="workEffortId">
        <default-field title="Task ID"><display/></default-field>
      </field>
```

```
        <field name="rootWorkEffortId"><default-field title="Project">
          <drop-down>
            <entity-options key="${workEffortId}"
                text="${workEffortId}: ${workEffortName}">
              <entity-find entity-name="WorkEffortAndParty">
                <date-filter/>
                <econdition field-name="partyId"
                    from="ec.user.userAccount.partyId"/>
                <econdition field-name="workEffortTypeEnumId"
                    value="WetProject"/>
              </entity-find>
            </entity-options>
          </drop-down>
          <link text="Edit ${project.workEffortName} [${task.rootWorkEffortId}]"
              url="editProject">
            <parameter name="workEffortId" from="task.rootWorkEffortId"/>
          </link>
        </default-field></field>
        <field name="milestoneWorkEffortId"
            entry-name="milestoneAssoc?.workEffortId">
          <default-field title="Milestone">
            <drop-down combo-box="true">
              <dynamic-options transition="getProjectMilestones"
                  value-field="workEffortId" label-field="milestoneLabel">
                <depends-on field="rootWorkEffortId"/>
              </dynamic-options>
            </drop-down>
            <link url="milestoneSummary"
text="${milestoneAssoc ? 'Edit ' + milestoneAssoc.workEffortId : ''}">
              <parameter name="milestoneWorkEffortId"
                  from="milestoneAssoc?.workEffortId"/>
            </link>
          </default-field>
        </field>

         <field name="parentWorkEffortId"><default-field title="Parent Task">
           <drop-down combo-box="true">
             <dynamic-options transition="getProjectTasks"
                 value-field="workEffortId" label-field="taskLabel">
               <depends-on field="rootWorkEffortId"/>
             </dynamic-options>
           </drop-down>
         </default-field></field>
         <field name="workEffortName"><default-field title="Task Name">


           <text-line/></default-field></field>
         <field name="priority"><default-field>
           <widget-template-include location="component://HiveMind/template/
               screen/ProjectWidgetTemplates.xml#priority"/>
         </default-field></field>
         <field name="purposeEnumId"><default-field title="Purpose">
           <widget-template-include location="component://webroot/template/
               screen/BasicWidgetTemplates.xml#enumWithParentDropDown">
             <set field="enumTypeId" value="WorkEffortPurpose"/>
             <set field="parentEnumId" value="WetTask"/>
           </widget-template-include>
         </default-field></field>
```

```
        <field name="statusId"><default-field title="Status">
          <widget-template-include location="component://webroot/template/
    screen/BasicWidgetTemplates.xml#statusTransitionWithFlowDropDown">
            <set field="currentDescription"
              from="task?.'WorkEffort#moqui.basic.StatusItem'?.description"/>
            <set field="statusId" from="task.statusId"/>
          </widget-template-include>
        </default-field></field>
        <field name="resolutionEnumId"><default-field title="Resolution">
          <widget-template-include location="component://webroot/template/
              screen/BasicWidgetTemplates.xml#enumDropDown">
            <set field="enumTypeId" value="WorkEffortResolution"/>
          </widget-template-include>
        </default-field></field>
        <field name="estimatedCompletionDate">
          <default-field title="Due Date">
            <date-time type="date" format="yyyy-MM-dd"/></default-field>
        </field>
        <field name="estimatedWorkTime">
          <default-field title="Estimated Hours">
            <text-line size="5"/></default-field>
        </field>
        <field name="remainingWorkTime">
          <default-field title="Remaining Hours">
            <text-line size="5"/></default-field>
        </field>
        <field name="actualWorkTime"><default-field title="Actual Hours">
          <display format="#.00"/></default-field></field>
        <field name="description"><default-field title="Description">
          <text-area rows="20" cols="100"/></default-field></field>
        <field name="submitButton"><default-field title="Update">
          <submit/></default-field></field>

        <field-layout>
          <fields-not-referenced/>
          <field-row><field-ref name="purposeEnumId"/>
            <field-ref name="priority"/></field-row>
          <field-row><field-ref name="statusId"/>

           <field-ref name="estimatedCompletionDate"/></field-row>
         <field-row><field-ref name="estimatedWorkTime"/>
           <field-ref name="remainingWorkTime"/></field-row>
         <field-ref name="actualWorkTime"/>
         <field-ref name="description"/>
         <field-ref name="submitButton"/>
       </field-layout>
     </form-single>
   </widgets>
 </screen>
```

这个界面基于单个 workEffortId 参数，任务的 ID，查找到所有的数据。

列表表单

使用 form-single 元素定义单个表单。下面是 form-list 元素的属性：

- **name:** 表单的名称。用于通过界面 XML 文件的位置去引用表单。对于 HTML 输出来说，这是表单的名称和 id，同时对于其他的输出来说，它也可被用于定义表单相关的输出部分。
- **extends:** 扩展使用井号（#）分隔位置和名称的表单。如果这里没有位置则它将被视为当前界面中的一个表单名称。

- **transition:** 当前界面提交表单跳转到的转换。
- **multi:** 将表单作为多值提交的表单，页面上所有的行在单个请求中一起提交，同时每个字段都有个"**_${rownumber}**"后缀。同样还传递一个**_isMulti**=true 参数，这样服务门面知道为每一行去运行服务（在 transition 中的单个 service-call）。默认为 true，因此设置为 false 去禁用上面的行为并为每一行有一个独立的表单（单独提交）。
- **list:** 一个遍历列表去计算的表单式。
- **list-entry:** 如果指定则每个列表条目都会使用这个名称被放到上下文中，否则列表条目必须为一个 Map，并且每行的 map 中的条目将会被放到上下文中。
- **paginate:** 表明表单是否需要进行分页。默认为 true。
- **paginate-always-show:** 总是显示带有记录行总数的分页控制，哪怕只有一页。默认为 true。
- **skip-start:** 表单渲染跳过的字段的起点。在一个表单之后使用 **skip-end**=true，这会有效的将多个表单组合成一个表单。
- **skip-end:** 表单渲染跳过的字段的终点。使用它去离开一个打开的表单，这样额外的表单可以和它一起进行组合。
- **skip-form:** 令输出成为一个普通的表格，不能提交（在 HTML 中不生成 form 元素）。最小化的输出只读的列表表单时很有用。
- **dynamic:** 如果为 true，表单将被认为是动态的，并且内部定义每次都会被构建而不仅仅是第一次引用的时候去构建。当 auto-fields-*元素为服务或实体名称有**${}**字符扩展，这就十分的必要。

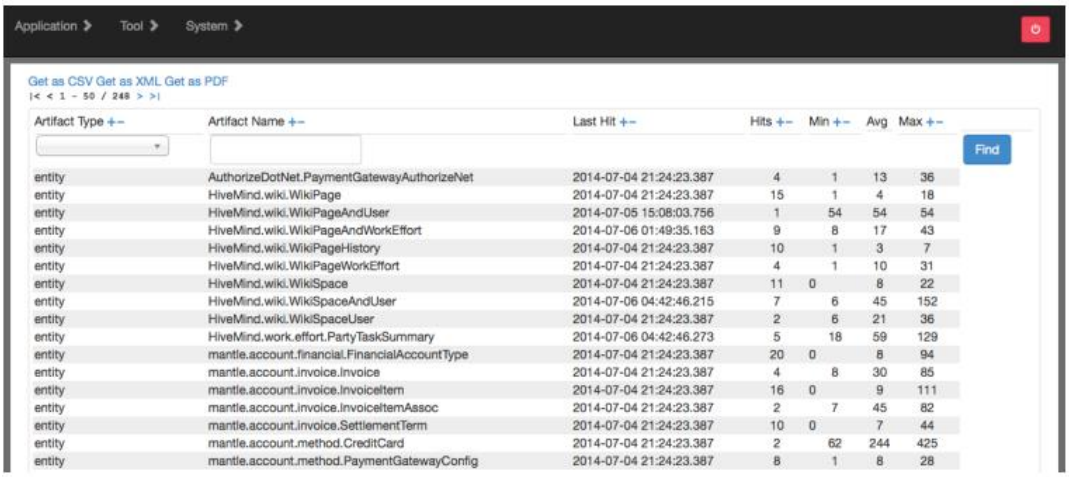
类似于单个表单中的 field-layout，列表表单中有个 form-list-column 元素。当使用它时需要每个表格中的列都有一个元素，并且所有的字段都参照一列，否则将不会被显示。form-list-column 元素有单个子元素，用于单个表单 field-layout 中相同的 field-ref 元素。

表单的数据准备最好通过界面 XML 的动作 actions 去完成，但有时你需要为列表表单的每一行去准备数据。这可以通过高级的 Map 的 List 对象为列表表单条目字段准备数据去实现。通过这种方式，准备 List 的业务逻辑可以为准备数据进行额外的数据查找或计算。另一种方式是在 form-list.row-actions 元素下添加 XML 动作。这些为每一行运行的动作都在一个独立的上下文中，因此任何上下文中定义的字段都只能用于对应的那一行。

➢ *列表表单展现/导出的实例*

列表表单有两个主要的分类：一类用于搜索，展现和导出，以及用于在单个界面中编辑大量数据的一类。

Moqui 工具应用中的构件概要界面就是一个很好的例子，它的界面用于查询，展现数据以及使用相同的界面和表单定义去导出结果到 CSV，XML 和 PDF 文件中。界面上的列表表单为每个构件概要展示了一行源于 moqui.server.ArtifactHitReport view-entity 视图实体的 moqui.server.ArtifactHitBin 记录。



注意左上角的"Get as CSV"链接（类似的 XML 和 PDF 链接）。这个链接会前往简单的 ArtifactHitSummaryStats.csv 转换到相同的界面并添加 **renderMode**=csv，**pageNoLimit**=true 和 **lastStandalone**=true 参数，这样界面渲染进行 csv 输出而不是 html，禁用分页（所有的结果都输出），并只有最后一个界面呈现（跳过所有的父界面去避免装饰，例如最后一个界面是“独立的”）。查看 ***XML, CSV and Plain Text Handling*** 章节获得更多细节。

在"Get as"链接下面是分页控制，它默认为启用的，并且它默认在多于一页的结果进行展示时显示。在表单的标题行上的是列标题和用于每列结果进行排序的"+-"链接，加上一个带有的构件类型（Artifact Type）下拉框 drop-down 以及构件名称（Artifact Name）text-find 框的查询表单头。这些都定义在每个字段下的 header-field 元素中。

列表表单使用 form-list.row-actions 元素去为每行计算平均次数 averageTime，然后用一个表单字段去展现。

下面是 ArtifactHitSummary.xml 界面的代码，其中展示了上面概要中的细节：

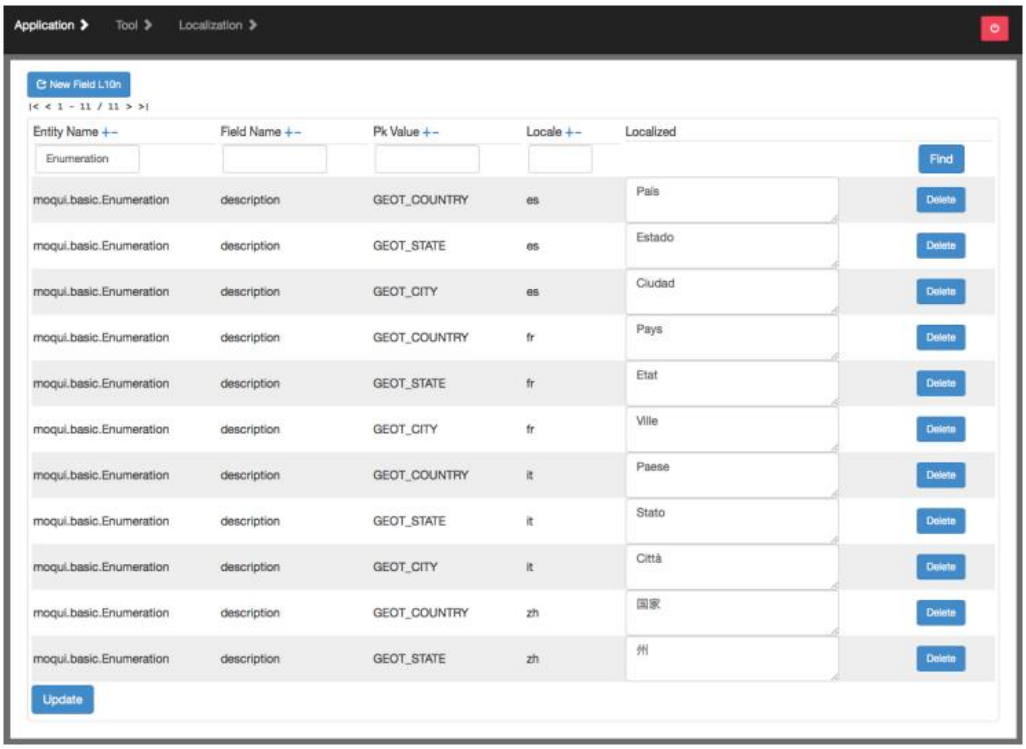
```
<screen xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="http://moqui.org/xsd/xml-screen-1.4.xsd"
        default-menu-title="Artifact Summary">

  <transition name="ArtifactHitSummaryStats.csv">
    <default-response url="."><parameter name="renderMode" value="csv"/>
      <parameter name="pageNoLimit" value="true"/>
      <parameter name="lastStandalone" value="true"/></default-response>
  </transition>
  <transition name="ArtifactHitSummaryStats.xml">
    <default-response url="."><parameter name="renderMode" value="xml"/>
      <parameter name="pageNoLimit" value="true"/>
      <parameter name="lastStandalone" value="true"/></default-response>
  </transition>
  <transition name="ArtifactHitSummaryStats.pdf">
    <default-response url-type="plain"
        url="${ec.web.getWebappRootUrl(false, null)}/fop/apps/tools/System/
          ArtifactHitSummary">
      <parameter name="renderMode" value="xsl-fo"/>
      <parameter name="pageNoLimit" value="true"/>
    </default-response>
  </transition>

  <actions>
    <entity-find entity-name="moqui.server.ArtifactHitReport"
        list="artifactHitReportList" limit="50">
      <search-form-inputs default-order-by="artifactType,artifactName"/>
    </entity-find>
  </actions>
```

```
  <widgets>
    <container>
      <link url="ArtifactHitSummaryStats.csv" text="Get as CSV"
          target-window="_blank" expand-transition-url="false"/>
      <link url="ArtifactHitSummaryStats.xml" text="Get as XML"
          target-window="_blank" expand-transition-url="false"/>
      <link url="ArtifactHitSummaryStats.pdf" text="Get as PDF"
          target-window="_blank"/>
    </container>
    <form-list name="ArtifactHitSummaryList" list="artifactHitReportList">
      <row-actions>
        <set field="averageTime" from="(totalTimeMillis/hitCount as
            BigDecimal).setScale(0,BigDecimal.ROUND_UP)"/>
      </row-actions>

      <field name="artifactType">
        <header-field show-order-by="true">
          <drop-down allow-empty="true">
            <option key="screen"/><option key="screen-content"/>
            <option key="transition"/>
            <option key="service"/><option key="entity"/>
          </drop-down>
        </header-field>
        <default-field><display also-hidden="false"/></default-field>
      </field>
      <field name="artifactName">
        <header-field show-order-by="true">
          <text-find hide-options="true" size="20"/></header-field>
        <default-field><display text="${artifactName}"
            also-hidden="false"/></default-field>
      </field>
      <field name="lastHitDateTime">
        <header-field title="Last Hit" show-order-by="true"/>
        <default-field><display also-hidden="false"/></default-field>
      </field>
      <field name="hitCount">
        <header-field title="Hits" show-order-by="true"/>
        <default-field><display also-hidden="false"/></default-field>
      </field>
      <field name="minTimeMillis">
        <header-field title="Min" show-order-by="true"/>
        <default-field><display also-hidden="false"/></default-field>
      </field>
      <field name="averageTime">
        <default-field title="Avg">
          <display also-hidden="false"/></default-field>
      </field>
      <field name="maxTimeMillis">
        <header-field title="Max" show-order-by="true"/>
        <default-field><display also-hidden="false"/></default-field>

      </field>
      <field name="find"><header-field title="Find">
        <submit/></header-field></field>
    </form-list>
  </widgets>
</screen>
```

➢ ***列表表单编辑的实例***

Moqui 工具应用中的实体字段本地化界面是一个在单个页面中更新多条记录的列表表单很好的实例。这个界面被设计用于添加，修改和删除指定使用本地化文本而非实体记录字段的实际值的 moqui.basic.LocalizedEntityField 记录。

在下面的屏幕截图中左上角有一个按钮，用于在一个 container-dialog 模态弹出窗口中添加一条新的记录。按钮下面是默认启用的分页控制条。表单的标题行有字段标题（既然这里没有 header-field.**title** 属性，那么这种情况下就会基于所有的字段名称进行生成），“+-”排序链接（通过 header-field.**show-order-by**=true 设置），以及查询使用的字段头部部件。



列表表单中的表格的每一条记录行都有一个删除按钮，但是更新按钮在底部并在单个界面提交中一次更新所有本地化数据的行记录值。注意查询按钮在标题行中，并与每行中的删除按钮在相同的列上。在表单定义中去定义查询按钮，使用 header-field 元素的子元素并使用 delete 字段列。

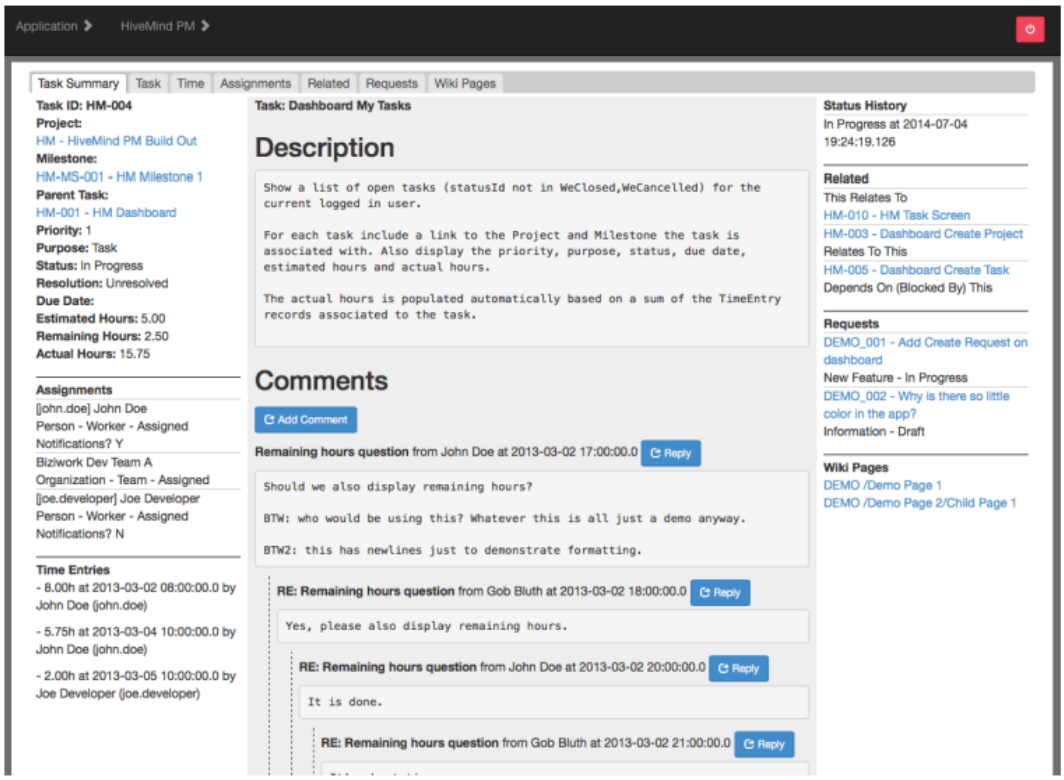
下面的是 EntityFields.xml 界面的代码。新增，修改和删除转换使用了隐式的实体自动服务定义，所以这里没有为它们进行服务定义或实现。此功能仅依赖于一个界面 XML 文件以及 LocalizedEntityField 实体的定义。

```
<screen xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="http://moqui.org/xsd/xml-screen-1.4.xsd"
        default-menu-title="Entity Fields" default-menu-index="2">

  <transition name="createLocalizedEntityField">
    <service-call name="create#moqui.basic.LocalizedEntityField"/>
    <default-response url="."/>
  </transition>
  <transition name="updateLocalizedEntityField">
    <service-call name="update#moqui.basic.LocalizedEntityField"
        multi="true"/>
    <default-response url="."/>
  </transition>
  <transition name="deleteLocalizedEntityField">
    <service-call name="delete#moqui.basic.LocalizedEntityField"/>
    <default-response url="."/>
  </transition>

  <actions>
    <entity-find entity-name="moqui.basic.LocalizedEntityField"
        list="localizedEntityFieldList" offset="0" limit="50">
      <search-form-inputs default-order-by="entityName,fieldName,locale"/>
    </entity-find>
  </actions>
  <widgets>
    <container>
      <container-dialog id="CreateEntityFieldDialog"
          button-text="New Field L10n">
        <form-single name="CreateLocalizedEntityField"
            transition="createLocalizedEntityField">
          <field name="entityName"><default-field>
            <text-line size="15"/></default-field></field>
          <field name="fieldName"><default-field>
            <text-line size="15"/></default-field></field>
          <field name="pkValue"><default-field>
            <text-line size="20"/></default-field></field>
          <field name="locale"><default-field>
              <text-line size="5"/></default-field></field>
            <field name="localized"><default-field>
              <text-area rows="5" cols="60"/></default-field></field>
            <field name="submitButton"><default-field title="Create">
              <submit/></default-field></field>
        </form-single>
      </container-dialog>
    </container>
    <form-list name="UpdateLocalizedEntityFields"
        list="localizedEntityFieldList"
        transition="updateLocalizedEntityField" multi="true">
      <field name="entityName">
        <header-field show-order-by="true">
          <text-find hide-options="true" size="12"/></header-field>
        <default-field><display/></default-field>
      </field>
      <field name="fieldName">
        <header-field show-order-by="true">
          <text-find hide-options="true" size="12"/></header-field>
        <default-field><display/></default-field>
      </field>
      <field name="pkValue">
        <header-field show-order-by="true">
          <text-find hide-options="true" size="12"/></header-field>
        <default-field><display/></default-field>
      </field>
```

```
        <field name="locale">
          <header-field show-order-by="true">
            <text-find hide-options="true" size="4"/></header-field>
          <default-field><display/></default-field>
        </field>
        <field name="localized"><default-field>
          <text-area rows="2" cols="35"/></default-field></field>
        <field name="update"><default-field title="Update">
          <submit/></default-field></field>
        <field name="delete">
          <header-field title="Find"><submit/></header-field>
          <default-field>
            <link text="Delete" url="deleteLocalizedEntityField">
              <parameter name="entityName"/>
              <parameter name="fieldName"/><parameter name="locale"/></link>
          </default-field>
        </field>
      </form-list>
    </widgets>
  </screen>
```

模板

各种各样的界面可以构建于 XML 表单，各种界面 XML 部件以及布局元素。使用开箱即用的元素可以做非常多的事情。这里有个更为复杂的界面例子，源自 HiveMind PM 项目的任务总结界面就是由开箱即用的元素和一些自定义 CSS 构成的：



有时你需要一个更灵活的布局，样式，部件或自定义交互行为。使用 FTL 宏定义添加或者扩展界面 XML 元素的方式去添加界面和表单部件（包括布局元素），可以保证界面呈现的一致性，并用于很多的地方。对于其他任何事情，特别是一次性的呈现，一个明确的模板是你想要获得任意种类 HTML 输出的方式。

对于定制化的 web 站点，例如需要个性化的 HTML 去获得一个特别明确的形式和功能的企业或电子商务网站来说特别的有用。

自定义模板同样可用于其他类型的表单输出如 XML，CSS 以及 XSL-FO。在界面 XML 中使用 render-mode 元素和一个或多个 text 子元素为每个 render-mode.text.**type** 来支持界面呈现。当前版本的 Moqui 框架中界面渲染仅支持文本输出，但未来或客制化代码实现的其他在 render-mode 元素下的元素可用于定义界面的输出呈现，比如 GWT 或 Swing。

如果界面通过渲染模式进行呈现并且没有匹配当前渲染模式的 **type** 属性的 text 子元素，那么将会简单的呈现空白页，同时继续进行界面呈现。text.**type** 属性的选项匹配每种输出类型定义使用的宏模板所在的 Moqui XML 配置文件里的 screen-facade.screen-text-output 元素上的 **type** 属性。当前支持的选项包括：csv，html，text，xml 和 xsl-fo。

text 元素上其他可用的属性（**type** 属性之外的）包括：

- **location:** 这是模板或文本文件的位置，并且可以是任何资源门面支持的路径位置，包括：文件路径，http，组件，内容等。
- **template:** 将路径位置上的文本当作 FTL 或其他资源门面支持的模板类型的模板进行解释。默认为 true，如果你想要逐字的输出文本，将其设置为 false。
- **encode:** 如果为 true，则文本将会被编码，这样它就不会被目标输出的标记所干扰。模板忽略这个设置并永不编码。举个例子，如果输出为 HTML 则数据将会被 HTML 编码，这样所有的 HTML 特性字符将会被转义。
- **no-boundary-comment:** 默认为 false。如果为 true，它将永不在此之前（XML 标签开始之前等）放置边界注释。

在开箱即用的运行时目录中 webroot.xml 界面是默认的根界面，同时包含了一个为不同界面呈现的模板的好例子：

```
<widgets>
  <render-mode>
    <text type="html"
        location="component://webroot/screen/includes/Header.html.ftl"/>
    <text type="xsl-fo" no-boundary-comment="true"
        location="component://webroot/screen/includes/Header.xsl-fo.ftl"/>
  </render-mode>

  <subscreens-active/>

  <render-mode>
    <text type="html"
        location="component://webroot/screen/includes/Footer.html.ftl"/>
    <text type="xsl-fo"><![CDATA[
        ${sri.getAfterScreenWriterText()}
        </fo:flow></fo:page-sequence></fo:root>
    ]]></text>
  </render-mode>
</widgets>
```

这个界面的例子有子界面，所以它在 subscreens-active 元素前后都有 render-mode 元素，用于装饰（或包装）子元素的内容。例子展示了带有包含了一个 FTL 模板 **location** 的 text 元素，以及 text 元素下 CDATA 块中的内联文本。

发送和接收邮件

发送和接收邮件的第一步是安装一个邮件服务 EmailServer，加载的记录如下所示：

```
<moqui.basic.email.EmailServer  emailServerId="SYSTEM"
    smtpHost="mail.test.com"  smtpPort="25"  smtpStartTls="N"  smtpSsl="N"
```

storeHost="mail.test.com" storePort="143" storeProtocol="imap"
storeDelete="N" mailUsername="TestUser" mailPassword="TestPassword"/>

注意这些都是例子数据并需要被改为真实值，特别是 **smtpHost**，**storeHost**，**mailUsername** 以及 **mailPassword** 字段。**store***字段用于为收件箱设置远程邮件存储。这里还有端口字段的其他常用值：

- **smtpPort**: 25 (SMTP), 465 (SSMTP), 587 (SSMTP)
- **storeProtocol**=imap 的 **storePort**: 143 (IMAP), 585 (IMAP4-SSL), 993 (IMAPS)
- **storeProtocol**=pop3 的 **storePort**: 110 (POP3), 995 (SSL-POP)

如果你需要使用多邮件服务进行工作，只需要添加 EmailServer 记录并为每个进行设置即可。当使用一个邮件模板发送邮件时，EmailServer 通过 EmailTemplate 记录的 **emailServerId** 字段进行指定。

说到 EmailTemplate，发送邮件的第二步就是创建一个邮件模板。这里有个源自 HiveMind PM 中发送任务更新通知邮件的例子：

```
<moqui.basic.email.EmailTemplate emailTemplateId="HM_TASK_UPDATE"
    description="HiveMind Task Update Notification"
    emailServerId="SYSTEM" webappName="webroot"
    bodyScreenLocation="component://HiveMind/screen/TaskUpdateNotification.xml"
    fromAddress="test@test.com" ccAddresses="" bccAddresses=""
    subject="Task Updated: ${document._id} - ${document.WorkEffort.name}"/>
```

通常的想法是在邮件发送时，定义一个界面去呈现邮件主体（**bodyScreenLocation**）。邮件主体界面和普通 UI 界面有点区别，因为当其通过一个 web 请求进行渲染时，这里没有可用的 Web 门面。URL 前缀（域名，端口等）基于 Moqui XML 配置文件中的 webapp 设置进行生成，这就是为什么有必要去指定一个匹配 moqui-conf.webapp-list.webapp.**name** 属性的 **webappName**。

邮件主题 **subject** 同样也是种简单的模板，它是个扩展的 Groovy String，当邮件被发送时使用和呈现主体相同的上下文环境。**fromAddress** 字段为必须字段，你可以选择的指定 **ccAddresses** 和 **bccAddresses**。

可以使用 EmailTemplateAttachment 实体去添加附件到 EmailTemplate 上。邮件中使用的文件名必须通过 **filename** 字段进行指定。界面渲染的附件本身通过 **attachmentLocation** 字段进行指定。**screenRenderMode** 字段传递到 ScreenRender 去指定界面的输出类型。它同样用于决定 MIME/内容类型。如果邮件内容为空，那么 **attachmentLocation** 将会无界面呈现的被发送，同时他的 MIME 类型基于附件的扩展名。这也适用于通过设置 **screenRenderMode** 为 xsl-fo，去生成 XSL:FO 并转变为 PDF 附在邮件上。

一旦 EmailServer 和 EmailTemplate 被定义完成，你就可以使用 org.moqui.impl.EmailServices.**send#EmailTemplate** 服务发送邮件了。当调用这个服务时，传递 **emailTemplateId** 参数去指定 EmailTemplate。就像上面提及的，EmailServer 将会基于 EmailTemplate.**emailServerId** 字段去确定。

接收邮件信息的地址通过 **toAddresses** 参数进行传递，这个参数是个普通的 String 字符串并可以有多个逗号分隔的地址。这些参数用于在独立于服务的上下文中渲染邮件界面并通过 **bodyParameters** 输入参数进行传递。默认的，**send#EmailTemplate** 服务保存 EmailMessage 实体的输出信息记录的细节。可以通过传递 **createEmailMessage** 参数值为 false 去禁用保存。输出参数是 **messageId**（存放在 Message-ID 邮件标头字段中的值）以及 **emailMessageId**（如果创建了一条 EmailMessage 记录）。

EmailMessage 实体用于传入和传出两者的邮件信息。对于发送的信息使用

send#EmailTemplate 服务，状态（**statusId**）起始于 Sent（实际上先设置为 Ready，发送邮件，然后设置其为 Sent）并可能会变为 Viewed，如果这里基于一个图像请求（通常将 **emailMessageId** 作为一个参数或路径元素）追踪去打开消息。如果邮件消息无法投递退回，状态将变为 Bounced。

一个邮件消息（EmailMessage）同样也可以手动的发送而不是通过一个模板，这种情况下状态应当起始于 Draft。一旦用户完成了消息编辑，状态应当变为 Ready，然后当其真的被发送了，状态变为 Sent。传入的消息起始于 Received 状态，并可以在邮件被打开之后改变为 Viewed 状态。

对于邮件线程来说，EmailMessage 实体有个 **rootEmailMessageId** 字段用于原始信息来将同线程下的所有邮件信息分组到一起，以及 **parentEmailMessageId** 字段用于标识当前邮件消息立即回复的那个邮件消息。

接收邮件遵循一个非常不同的路径。org.moqui.impl.EmailServices.**poll#EmailServer** 服务轮询一个基于 EmailServer 实体设置的 IMAP 或 POP3 邮箱。它只有单个输入参数，**emailServerId**。通常这个服务作为一个调度服务运行。

对邮箱中找到的并未标记已阅的每个邮件消息来说，这个服务可以调用邮件 ECA（EMECA）规则。这些规则与实体和服务 ECA 规则相同，只是这里没有特殊的触发器，仅有接收一个邮件。条件仅可用于运行动作在发往一个特定的地址，或像是标记主题，或者其他期望的条件场景中。

条件及动作的上下文将包含一个带有邮件所有标头（既可以是 String，也可以是一个 String 的 List，如果这里有多个标头）的 **headers** Map，以及一个 **fields** Map，包含：**toList**, **ccList**, **bccList**, **from**, **subject**, **sentDate**, **receivedDate**, **bodyPartList**。***List** 字段是一个 String 的 List，***Date** 字段是一个 java.util.Date 对象。你可以实现 org.moqui.EmailServices.**process#EmailEca** 接口为直接调用服务去安装上下文。

动作和服务调用可以和传入邮件一起做任何事情。保存传入的邮件信息你可以使用 org.moqui.impl.EmailServices.**save#EcaEmailMessage** 服务。

八、 系统接口

为了支持用户接口，Moqui 框架支持各种其他系统接口的选项。这里有基于标准的选项和方式去构建更多的客制化系统接口。

数据和逻辑层接口

系统接口通常可以分为两个主要的类别：支持过程中的一个步骤和传输数据（通常用于保持另一个系统的数据更新）。对于大部分的系统集成来说，一个过程级别的接口更为灵活并更关注于系统中特定的部分，而不是传输所有的数据。有时保持系统间的数据一致性是集成的本质需求或唯一可用的选项，然后数据层集成就是处理这方面的需求。Moqui 有工具用于逻辑/过程以及数据层两者的系统接口。

触发呼出消息最好的方式是通过 ECA（event-condition-action）规则，无论是用于逻辑层接口的服务 ECA（SECA）规则，还是用于数据层接口的实体 ECA（EECA）规则。查看 ***Service ECA Rules*** 和 ***Entity ECA Rules*** 部分获得如何去定义它们的细节。

所有的 ECA 规则调用动作，特别是一个或多个 service-call 动作，并且这些动作将调用出去，无论什么样的系统接口需要。这种方式可以通过客制化代码或简单的调用一个已存在的本地或远程服务去完成。后面的部分会描述 Moqui 中指定的可用工具，同时你可以使用客制化代码去实现任何接口和使用任何你需要的额外类库。

XML，CSV 和简单文本处理

使用 Moqui 框架你可以有很多种生产和消费 XML，CSV，JSON 以及其它文本数据的途径。

Groovy 中用于生产和消费 XML 的良好 API 有：

- groovy.util.Node: 这个 Groovy 类用于表示一个带有属性和子节点的树节点。对于 XML 数据来说，每个元素都表现为一个节点 Node。
- groovy.util.XmlNodePrinter: 打印源于整棵树节点 Node 对象的 XML 文本。
- groovy.util.XmlParser: 读取 XML 文本到一棵树节点 Node 对象中。
- groovy.util.XmlSlurper: 读取 XML 文本到一个 GPathResult 对象中，它使用类似 XPath 表达式的语法在 Groovy 中用于抽取一个 XML 元素树的指定部分。
- groovy.xml.MarkupBuilder: 提供一个 Groovy DSL（领域特定语言）用于编码，其有个类似 XML 输出的结构。对于显示的创建脚本和相对于更动态的构建 XML 树来说最有用。

Moqui 中还使用了很多其他的 Java 编写的 XML 类库，例如 dom4j 和 JDOM。如果你选择使用它们，只要在 Gradle 编译路径下包含 JAR 文件，就可以进行编码了。

对于 CSV 文件，Moqui 使用了 Apache Commons CSV 类库，并且就像操作 XML 文件一样其他类库也可以被使用。你可以在 org.moqui.impl.entity.EntityDataLoaderImpl.EntityCsvHandler 类中查看 Moqui 如何去使用它的。

在 Moqui 框架中，用于和导出数据的主要工具是 XML 表单，特别是列表表单。界面 XML 和表单可被渲染成包括 XML，CSV 以及纯文本等各种形式。要实现它，不管是在界面动作中还是通过一个请求参数在 web 请求中，只要设置上下文中的 **renderMode** 字段即可。它要和 Moqui XML 配置文件中的 screen-facade.screen-text-output.**type** 属性相匹配，同时它可以被

设置为任何已定义的值，包括 Moqui 默认的这些（csv, html, text, xml, xsl-fo）或任何你在你的运行时 Moqui XML 配置文件中定义的那些。

XML 表单可能会安装了分页（这是默认项）。设置 **pageNoLimit** 字段为 true，可以导出（或其他的原因）所有的结果而不是分页的结果。在有些情况下，特别是对于 XML 文件来说，你不想要渲染任何通常用于布置最终呈现界面的父界面。对于 CSV 文件其他界面元素通常都会忽略。可以设置 **lastStandalone** 字段为 true，这意味着最后的界面会独立的呈现并在界面路径中不包含父界面。这些可以作为请求参数，设置在 web 请求的界面动作中。

就像其他界面 XML 和 XML 表单输出形式一样，FTL 宏定义模板用于生成输出并可以通过包含和覆盖/添加来进行自定义。通过这种方式你可以在 Moqui 中为特定的界面（包括子界面，整个应用或应用的截面等）或任何东西进行自定义的输出。

对于拥有 CSV，XML 以及 XSL-FO（PDF）输出选项的界面和表单的案例细节请参看 ***List Form View/Export Example*** 部分。

Web 服务

XML-RPC 和 JSON-RPC

Moqui 有提供和消费 XML-RPC 及 JSON-RPC 服务的工具。任何服务门面的服务都可以通过设置 service.**allow-remote** 属性为 true 来曝露为一个远程可调用的服务。

web 门面有接收这些 RPC 调用的方法： ec.web.**handleXmlRpcServiceCall**() 和 ec.web.**handleJsonRpcServiceCall**()。在开箱即用的 webroot 组件中有一个 rpc.xml 界面，它包含了 **xml** 和 **json** 转换用于调用这些方法。通过这种设置，远程服务调用的 URL 路径为/rpc/xml 和/rpc/json。

下面是个 JSON-RPC 服务调用的例子，它使用了 curl 作为客户端。例子中通过名称，类型以及状态参数去调用 org.moqui.example.ExampleServices.**createExample** 服务。它同样传递了用户名和密码用于运行服务前的授权（遵循用于所有服务门面的服务调用的模式）。

id 字段一般总是为 1。这个 JSON-RPC 字段用于多消息请求，请求中的每个消息都会有个不同的 **id** 值并且在响应中这个值用于 **id** 字段。为了实现它，JSON 字符串需要有一个外部的列表去容纳这些分隔的消息，就像这个例子中的一样。

```
curl -X POST -H "Content-Type: application/json" \
    --data   '{"jsonrpc":"2.0",
"method":"org.moqui.example.ExampleServices.createExample",   "id":1,
"params": {   "authUsername":"john.doe",   "authPassword":"moqui",
"exampleName":"JSON-RPC Test 1",   "exampleTypeEnumId":"EXT_MADE_UP",
"statusId":"EXST_IN_DESIGN"   }   }' \
    http://localhost:8080/rpc/json
```

当你运行它，你会得到一个如下的响应（**exampleId** 值将会发生变化）：

```
{ "jsonrpc":"2.0", "id":1, "result": { "exampleId":"100050" } }
```

Moqui 中 JSON-RPC 的实现遵循 JSON-RPC 2.0 的可用规范，位于：http://www.jsonrpc.org/specification。

XML-RPC 请求遵循简单的模式。Moqui 使用的是实现了 XML-PRC 可用规范

(http://xmlrpc.scripting.com/spec.html)的 Apache XML-RPC 库(http://ws.apache.org/xmlrpc/)。

尽管你可以直接使用一个类库(或类似于 RemoteJsonRpcServiceRunner.groovy 中的自定义 JSON 处理代码)通过编码的方式去调用远程的 XML-RPC 和 JSON-RPC 服务，但最简单的方式调用远程服务是使用一个代理服务定义。你需要这么做：

- 定义一个服务
- service.**type** 属性使用 remote-xml-rpc 或 remote-json-rpc
- 设置 service.**location** 为 RPC 服务器的 URL 和路径(例如 http://localhost:8080/rpc/json)，或是 Moqui XML 配置文件中匹配服务位置的值(如 service-facade.service-location.**name**)；这里有两种开箱即用的用于调用远程服务目的的服务位置：main-xml 和 main-json；这些以及额外期望的设置可在运行时的 Moqui XML 配置文件中进行配置，并然后在你的服务位置中使用以简化配置，特别是当你有不同的 URLs 用于测试以及生产环境的时候
- 设置 service.**method** 为调用的远程服务的名称；在 JSON-RPC 中它映射到 **method** 字段；在 XML-RPC 中它映射到 methodName 元素；当调用另一个 Moqui 服务器时，它是即将被调用的服务名称
- 服务可以使用参数去定义匹配远程服务定义，或设置为不校验输入；你同样也可以为类型转换去定义默认的和指定类型的参数，类型转换会在远程服务调用之前完成

当你本地进行服务调用时，服务门面将会调用远程服务并返回结果。换句话说，你调用了一个被配置成代理远程服务的本地服务。

发送和接收简单 JSON

有时一个 API 规范要求特定的 JSON 结构或是某种 JSON-RPC 封装的结构。在 web 门面中有些特性可令其变得更加简单。

当接收到一个 HTTP 请求(实际上当 web 门面初始化完成后)时，如果请求的 **Content-Type**(MIME type)是 application/json，它将会解析请求体中的 JSON 字符串，并如果外层元素是一个 Map(JSON 中的一个对象)，那么这个 Map 中的条目将会被添加到 web 参数(ec.web.parameters)中，同时 web 参数会通过界面呈现或界面转换的运行，自动的加到上下文(ec.context)中。如果外层元素是一个 List 列表(JSON 中的一个数组)，那么它会被放到一个**_requestBodyJsonList** web 参数中，并再次像前面的那样放到上下文中。

这使得我们可以很容易的从 web 请求中获取到 JSON 数据。这也解决了在 web 门面自动的查找请求体中的多部分内容(这也是 web 门面总是做得事情)之后获得请求体的问题，因为 Servlet 容器可能不允许在此之后再次读取请求体。

对于一个 JSON 响应来说，你可以通过在 HttpServletResponse 上进行各种设置和使用 Groovy JsonBuilder 去生成 JSON 文本，来手动的将响应放在一起。为方便起见，ec.web.**sendJsonResponse**(Object responseObj)方法为你实现了所有的需求。

在另一个方面来说，一个接收和响应都是通过 JSON 的 URL 请求，这里有特殊的工具因为 Groovy 和其他应用程序使得其十分的简单。举个例子，下面是个实际代码的变种用于远程调用 JSON-RPC 服务：

```
Map jsonRequestMap = [ jsonrpc:"2.0",  id:1,  method:method,  params:parameters ]
JsonBuilder jb = new JsonBuilder( )
jb.call( jsonRequestMap )
String jsonResponse = StupidWebUtilities.simpleHttpStringRequest( location,
        jb.toString( ), "application/json" )
Object jsonObj = new JsonSlurper( ).parseText( jsonResponse )
```

它使用了 JsonBuilder 和 JsonSlurper 这两个 Groovy 类，并使用了 Apache HTTP Client 类库中的 StupidWebUtilities.**simpleHttpStringRequest**()方法。

RESTful 接口

RESTful 服务使用一个 URL 模式和请求方法去定义服务，而不是如 JSON-RPC 和 XML-RPC 的方法名称。通常的想法是使用一条通过带有类型（如实体或物理表）的记录作为一个路径元素，同时记录的 ID 作为一个或多个路径元素（通常简单起见，如单个主键字段）的 URL 的记录去呈现。

当这条记录作为一个 web 资源进行交互时，HTTP 请求方法通过这条记录指定了去做什么。这非常类似于 Moqui 实体自动服务中的新增，更新以及删除的服务动词。GET 方法通常用于记录的查找。POST 方法通常映射为新建记录。PUT 方法通常映射为记录的更新。DELETE 方法很明显，一个删除操作。

举个例子，例如下面这个，查看 ExampleApp.xml 文件。

为支持 RESTful web services，我们在一个基于 web 的应用中需要一种途径将 HTTP 请求方法转变为敏感的形式去运行。在 Moqui 框架中使用 transition.**method** 属性去处理这个问题，如下所示：

```
<transition  name="ExampleEntity"  method="put">
    <path-parameter  name="exampleId"/>
    <service-call  name="org.moqui.example.ExampleServices.updateExample"
                   in-map="ec.web.parameters"  web-send-json-response="true"/>
    <default-response  type="none"/>
</transition>
```

为了测试这个转换使用一个 curl 命令行，例如更新 Example 实体中 **exampleId** 为 100010 的 **exampleName** 字段：

```
curl -X PUT -H "Content-Type: application/json" \
    -H "Authorization: Basic am9obi5kb2U6bW9xdWk=" \
    --data '{  "exampleName" : "REST Test - Rev 2"  }' \
    http://.../apps/example/ExampleEntity/100010
```

这个例子中有些重要的事情需要注意一下，以便可以更简单的去创建 Moqui 内部服务的 REST 封装：

- 使用基本的 HTTP 认证（john.doe/moqui），Moqui 框架会自动识别并用于身份认证
- 使用自动映射参数的 JSON 输入体（JSON 字符串必须有一个 Map 根对象）
- **exampleId** 字段作为路径的一部分被传递，并使用 path-parameter 元素作为一个普通的参数被对待
- 使用 ec.web.parameters Map 作为入参 **in-map**，去显式的传递 web 参数到服务上（同样可以使用也包含了 web 参数的 ec.context 中的上下文条目，但这种方式更显式和拘束）
- 使用方便的 service-call.web-send-json-response 属性和一个 none 类型的响应去发送一个 JSON 响应

在 ExampleApp.xml 文件中还有各种其他的例子用于处理 RESTful 服务请求。

通过 Apache Camel 的企业级集成

Apache Camel（http://camel.apache.org）是一个用于路由和处理消息的工具，企业级集成模式的工具在这里有描述（同时这个站点的其他网页里有很多关于 EIP 的其他很好的信息）：http://www.eaipatterns.com/toc.html

Moqui 框架有一个 Camel 的消息终端（MoquiServiceEndpoint）用于绑定到服务门面。它允许服务(通过 **type**=camel 设置)使用 MoquiServiceConsumer，作为一条消息发送到 Camel 上来发送服务调用。终端同样包含了一个消息生产者（MoquiServiceProducer），它作为一个 moquiservice 在 Camel 路由字符串中可用。

下面是源自 ExampleServices.xml 文件中的一些 Camel 服务的例子：

```
<service  verb="localCamelExample"  type="camel"
    location="moquiservice:org.moqui.example.ExampleServices.targetCamelExample">
  <in-parameters><parameter  name="testInput"/></in-parameters>
  <out-parameters><parameter  name="testOutput"/></out-parameters>
</service>
<service  verb="targetCamelExample">
  <in-parameters><parameter  name="testInput"/></in-parameters>
  <out-parameters><parameter  name="testOutput"/></out-parameters>
  <actions>
    <set  field="testOutput"  value="Here's the input: ${testInput}"/>
    <log  level="warn"  message="targetCamelExample testOutput: ${result.testOutput}"/>
  </actions>
</service>
```

当你调用 **localCamelExample** 服务时，它通过 Apache Camel 去调用 **targetCamelExample** 服务。这是一个非常简单的使用 Camel 服务的例子。为了获得你能通过 Camel 做哪些事情的一个全貌，Camel 组件是一个很好的开端：http://camel.apache.org/components.html

通常的想法是你可以：

- 从广泛的各种来源中获得消息信息（文件查询，传入的 HTTP 请求，JMS 消息以及很多其他的）
- 消息转换（支持格式包括 XML，CSV，JSON，EDI 等）
- 运行自定义表达式（甚至在 Groovy 中）
- 分离，合并，路由，过滤，充实或应用于任何其他的 EIP 工具
- 发送消息到终端

Camel 是一个非常灵活的并特性充足的工具，所以我这里推荐下面这些书，而不是尝试去用文档更多的证明这个问题：

- Bilgin Ibryam 编写的《Apache Camel 即时消息路由》（Instant Apache Camel Message Routing）
  - http://www.packtpub.com/apache-camel-message-routing/book
  - 这本书是一个快速入门，你可以很快的过一遍其中大量很酷的通过 Camel 做的东西。
- Scott Cranton 和 Jakub Korab 编写的《Apache Camel 开发人员手册》（Apache Camel Developer's Cookbook）
  - http://www.packtpub.com/apache-camel-developers-cookbook/book
  - 这本书有成百的使用 Camel 的小秘诀和实例

- Claus Ibsen 和 Jonathan Anstey 编写的《Camel 实战》（Camel in Action）
  - http://manning.com/ibsen/
  - 这是 Apache Camel 的经典读物。它覆盖了普通概念，各种内部细节，如何去应用各种 EIPs 以及很多组件的总结。这本书的网站也有一堆有用的在线资源的链接。

九、 安全机制

身份认证

用户的身份认证代码起始于一个 UserFacade.**loginUser**()方法的调用。这个方法调用实际是通过 Apache Shiro 去进行身份认证的。可执行上下文工厂（ExecutionContextFactoryImpl）内部调用了 Shiro 的 SecurityManager，主要的认证代码如下所示：

```
UsernamePasswordToken token = new UsernamePasswordToken( username, password )
Subject currentUser = eci.getEcfi( ).getSecurityManager( )
    .createSubject( new DefaultSubjectContext( ) )
currentUser.login( token )
```

Shiro 默认使用 MoquiShiroRealm 进行配置，所以它将会终止于调用 MoquiShiroRealm.**getAuthenticationInfo**()方法，这个方法使用 moqui.security.UserAccount 实体并处理类似禁用账户，保持追踪失败登录尝试等进行身份认证。这里有源于 shiro.ini 文件的配置行：

```
moquiRealm = org.moqui.impl.MoquiShiroRealm
securityManager.realms = $moquiRealm
```

Shiro 可用于其他认证领域的配置，例如 Shiro 自带的 CasRealm, JdbcRealm 或 JndiLdapRealm 类。你同样可实现你自己的或甚至修改 MoquiShiroRealm 类去更好的满足你的需求。Shiro 有文档用于你去编写自己的领域，并且这些类都有配置文档，例如下面的使用一个 LDAP 服务器，用于 JndiLdapRealm 的 JavaDoc：

http://shiro.apache.org/static/1.2.3/apidocs/org/apache/shiro/realm/ldap/JndiLdapRealm.html

回到默认使用的 MoquiShiroRealm，下面是源于 MoquiDefaultConf.xml 文件的默认配置，并可以在你运行时的 Moqui XML 配置文件中进行覆盖：

```
<user-facade>
    <password encrypt-hash-type="SHA-256"  min-length="6"  min-digits="1"
        min-others="1"  history-limit="5"  change-weeks="26"
        email-require-change="true"  email-expire-hours="48"/>
    <login max-failures="3"  disable-minutes="5"  history-store="true"
        history-incorrect-password="true"/>
</user-facade>
```

login 元素配置了禁用用户帐户 UserAccount (**max-failures**)前的最大登录失败数，当达到最大失败数时禁用账户的时间（**disable-minutes**），在 UserLoginHistory 实体中是否保存一个尝试登录的历史（**history-store**）以及是否在历史中持久化错误的密码（**history-incorrect-password**）。

password 元素用于配置密码约束，当创建账户（org.moqui.impl.UserServices.**create#UserAccount**）或更新密码（org.moqui.impl.UserServices.**update#Password**）时进行校验。

密码设置包含了用于持久化密码之前以及和输入密码比较前的哈希算法(**encrypt-hash-type**；MD5, SHA, SHA-256, SHA-384, SHA512)，最小密码长度（**min-length**），密码中数字字符最小数量（**min-digits**），密码中数字或字母之外的字符最小数量（**min-others**），密码修改时为避免使用相同的密码而记住的老密码的数量（**history-limit**），以及几周之后强制要求修改密码（**change-weeks**）。

忘记密码重置的主要途径是通过包含了一个随机生成密码的电子邮件去处理。**email-require-change** 属性指定首次登陆时是否需要修改来源于邮件临时的密码，**email-expire-hours** 属性指定了邮件中的密码多少小时后将到期。

简单权限

最基本的身份认证（authz）是一个通过编号的明确权限校验。构件授权（在下个章节中介绍）通常更灵活，因为它是配置于构件（界面，服务等）的外部，同时权限是可以继承的，以避免再利用构件（尤其是服务）时出现问题。

校验权限的 API 方法是 ec.user.**hasPermission**(String userPermissionId)方法。如果一个用户是一个拥有权限（UserGroupPermission）的用户组（UserGroup）的成员（UserGroupMember），他就会有权限。**userPermissionId** 可能指向到一条 UserPermission 记录，但是它也可能是任意的文本值，因为用户组权限（UserGroupPermission）没有外键到用户权限（UserPermission），也就是说它们组权限和个人权限没有关联。

构件授权

Moqui 中的构件授权启用外部配置去访问构件，如界面，界面转换，服务，甚至实体。通过这种方式，无需为每个构件去添加代码或者配置，去进行权限校验或者看当前用户是否可以访问构件。

构件运行堆栈和历史

构件运行门面 ArtifactExecutionFacade 用于在每个构件执行时，框架的所有部分都可以保持追踪构件。它维护了一个当前运行中的构件堆栈，当每个构件开始运行时压入栈中（使用其中的一个 **push**()方法），同时运行完成时弹出堆栈（通过 **pop**()方法）。随着每个构件的入栈，它同样会被添加到当前运行上下文 ExecutionContext（例如单个 web 请求，远程服务调用等）里的所有构件都使用的历史中。

使用 ArtifactExecutionInfo **peek**() 方法去获得堆栈顶部的构件信息，Deque<ArtifactExecutionInfo> **getStack**() 获得当前的整个堆栈，List<ArtifactExecutionInfo> **getHistory**()获得所有已执行完成的构件历史。

因为授权记录是可继承的，所以这对于构件授权来说十分重要。如果一个构件授权被配置为继承的，那么不仅这个构件本身被授权，而且它所使用的任何构件都将被授权。

想象一下，一个系统包含了成百的界面和转换，几千个服务以及几百个实体。为其中的每一个进行授权配置将需要付出巨大的努力，不管是初始化设置还是随时维护授权。这将很容易发生错误，无论是错误的允许和禁止访问构件，导致敏感数据或功能的曝露，还是用户在其工作中尝试去执行关键操作发生的运行时错误。

解决方案是授权继承。通过这种方式，你可以安装它来访问整个应用或应用的一部分。单个界面的权限配置将会被所有的子界面，转换，服务以及实体所继承。为了限制部分敏感的服务和实体域，可以使用一个拒绝授权去覆盖继承的授权，这需要对那些构件进行特殊的认证。通过这种方式，你就拥有了一个可灵活组合的，简单的以及细粒度控制的敏感的资源。

这也同样用于追踪每个构件的性能指标。参阅 ***Artifact Execution Runtime Profiling*** 章节

部分获得详情。

构件授权

构件授权配置的第一步是创建一个构件组。它为每个构件都包含了一个 ArtifactGroup 记录和 ArtifactGroupMember 记录，或在分组中使用构件名称模式。

举个例子，这里是带有根界面（(ExampleApp.xml)的 Example 应用作为成员的构件分组：

```
<moqui.security.ArtifactGroup  artifactGroupId="EXAMPLE_APP"
    description="Example App (via root screen)"/>
<moqui.security.ArtifactGroupMember  artifactGroupId="EXAMPLE_APP"
    artifactTypeEnumId="AT_XML_SCREEN"  inheritAuthz="Y"
artifactName="component://example/screen/ExampleApp.xml"/>
```

本例中 **artifactName** 属性值是界面的路径位置。它同样可以是构件名称（通过设置 **nameIsPattern**="Y"）的模式，当所有的服务或实体都在一个包中的时候特别有用。这里有个这样的例子，所有的服务都在 org.moqui.example 包中，或更具体的说，所有的服务全名都匹配"org\.moqui\.example\..*"这个正则表达式：

```
<moqui.security.ArtifactGroupMember artifactGroupId="EXAMPLE_APP"
    artifactName="org\.moqui\.example\..*"  nameIsPattern="Y"
    artifactTypeEnumId="AT_SERVICE"  inheritAuthz="Y"/>
```

授权配置的第二步是为构件组带上一条 ArtifactAuthz 记录。下面是一个记录的例子，它给 ADMIN 用户组一直（AUTHZT_ALWAYS）可以访问 EXAMPLE_APP 构件组中构件的所有动作（AUTHZA_ALL）的安装配置。

```
<moqui.security.ArtifactAuthz  artifactAuthzId="EXAMPLE_AUTHZ_ALL"
    userGroupId="ADMIN"  artifactGroupId="EXAMPLE_APP"
    authzTypeEnumId="AUTHZT_ALWAYS"  authzActionEnumId="AUTHZA_ALL"/>
```

授权的永久类型（**authzTypeEnumId**）覆盖拒绝（AUTHZT_DENY）授权，有别于允许授权（AUTHZT_ALLOW）被拒绝所覆盖。授权动作（**authzActionEnumId**）的其他选项包括：查看(AUTHZA_VIEW)，新建(AUTHZA_CREATE)，更新(AUTHZA_UPDATE)，删除(AUTHZA_DELETE)以及所有（AUTHZA_ALL)。

这里举的例子是一条仅同意查看授权的记录，为 EXAMPLE_VIEWER 用户组设置使用相同允许类型（同样可以为拒绝）的构件组：

```
<moqui.security.ArtifactAuthz  artifactAuthzId="EXAMPLE_AUTHZ_VW"
    userGroupId="EXAMPLE_VIEWER"  artifactGroupId="EXAMPLE_APP"
    authzTypeEnumId="AUTHZT_ALLOW"  authzActionEnumId="AUTHZA_VIEW"/>
```

实体构件授权同样可以被限制为特定的 ArtifactAuthzRecord 实体记录。它可以使用一个视图实体（**viewEntityName**）来关联当前登录用户的 **userId** 以及想要的记录。如果 **userId** 字段的名称不是 **userId**，则通过 **userIdField** 字段来指定它的名字。记录级授权的检查是通过在视图实体上进行一个带有当前 **userId** 以及实体主键字段的查询操作来完成的。你可以通过 **filterByDate** 属性或使用 ArtifactAuthzRecordCond 记录来添加特定的条件到 view-entity 定义

上，用于添加这个查询的约束。

如果当框架使用的构件认证失败时，则新建一条相关细节的 ArtifactAuthzFailure 记录。

构件伺服器

构件伺服器（Artifact Tarpit）用于大量的构件访问时，进行速度限制。这里有个构件组的例子，适用于所有的界面，同时如果这里每 60 秒（**maxHitsDuration**）内持续有超过 120 次点击（**maxHitsCount**），那么 ArtifactTarpit 将会限制所有的用户访问每个界面的时间为 60 秒（**tarpitDuration**）。

```
<moqui.security.ArtifactGroup    artifactGroupId="ALL_SCREENS"
    description="All Screens"/>
<moqui.security.ArtifactGroupMember    artifactGroupId="ALL_SCREENS"
    artifactName=".*"    nameIsPattern="Y"
    artifactTypeEnumId="AT_XML_SCREEN"/>
<moqui.security.ArtifactTarpit    userGroupId="ALL_USERS"
    artifactGroupId="ALL_SCREENS"    maxHitsCount="120"
    maxHitsDuration="60"    tarpitDuration="60"/>
```

当一个特定的用户（**userId**）超过了某个特定构件（**artifactName**）或特定类型（**artifactTypeEnumId**）配置的速率限制，框架将创建一条 ArtifactTarpitLock 记录，用于限制这个用户访问那个构件，直到一个明确的日期/时间（**releaseDateTime**）为止。

十、 性能

性能指标

构件点击统计

Moqui 框架根据 Moqui XML 配置文件中 server-stats.artifact-stats 元素的配置来保持构件使用（点击）和时间的统计信息。下面是默认的配置（在 MoquiDefaultConf.xml 文件中），你可以在运行时的配置文件中覆盖它们。默认的运行时开发配置文件（MoquiDevConf.xml）设置的记录甚至比这个还要多。

```
<server-stats  bin-length-seconds="900"  visit-enabled="true"  visitor-enabled="true">
    <artifact-stats  type="screen"  persist-bin="true"  persist-hit="true"/>
    <artifact-stats  type="screen-content"  persist-bin="true"  persist-hit="true"/>
    <artifact-stats  type="transition"  persist-bin="true"  persist-hit="true"/>
    <artifact-stats  type="service"  persist-bin="true"  persist-hit="false"/>
    <artifact-stats  type="entity"  persist-bin="false"/>
</server-stats>
```

这些设置为界面 screen，界面内容 screen-content (界面之下的内容)以及界面转换 transition 的点击都新建了一条 ArtifactHit 记录。它们同样新建了 ArtifactHitBin 记录用于那些加上了服务 service 调用的构件。

这里有一对 ArtifactHit 记录的例子，第一个是 FindExample.xml 界面的点击，第二个是工具应用的 DataExport.xml 界面中 EntityExport.xml 转换的点击。EntityExport.xml 转换的点击带有记录在 **parameterString** 属性中的参数。

```
<moqui.server.ArtifactHit  hitId="100030"  visitId="100000"
    userId="EX_JOHN_DOE"  artifactType="screen"  artifactSubType="text/html"
    artifactName="component://example/screen/ExampleApp/Example/FindExample.xml"
    startDateTime="1406670788608"  runningTimeMillis="1,499"  wasError="N"
    requestUrl="http://localhost:8080/apps/example/Example/FindExample"
    serverIpAddress="172.16.7.38"  serverHostName="DEJCMBA3.local"
    lastUpdatedStamp="1406670790120"/>

<moqui.server.ArtifactHit  hitId="100037"  visitId="100001"
    userId="EX_JOHN_DOE"  artifactType="transition"
    artifactName="component://tools/screen/Tools/Entity/DataExport.xml#EntityExport.xml"
    parameterString="moquiFormName=ExportData,output=file,filterMap=
            [artifactType:"screen"],entityNames=moqui.server.ArtifactHit"
    startDateTime="1406674728129"  runningTimeMillis="45"  wasError="N"
    requestUrl="http://localhost:8080/apps/tools/Entity/DataExport/EntityExport.xml"
    serverIpAddress="172.16.7.38"  serverHostName="DEJCMBA3.local"
    lastUpdatedStamp="1406674728195"/>
```

在一个 web 应用中，对每个会话都有个访问 Visit 记录，它有会话的明细并且通过 **visitId**

字段和ArtifactHit记录绑定在一起。一旦用户登录系统，Visit记录就会保持追踪登录的**userId**，但是甚至在登录之前，访问使用服务上追踪的保存在 Visitor 记录上的 **visitorId** 字段绑定在一起，同时通过 cookie 在浏览器/客户端上与会话绑定在一起，甚至这个会话上没有用户登录。

```
<moqui.server.Visit visitId="100000"  visitorId="100000"
    userId="EX_JOHN_DOE"  sessionId="749389362bac39c39de3c77769b9b485"
    serverIpAddress="172.16.7.38"  serverHostName="DEJCMBA3.local"
    webappName="ROOT"  initialLocale="en_US"
    initialRequest="http://localhost:8080/"  initialUserAgent="Mozilla/5.0
        (Macintosh; Intel Mac OS X 10_9_4) AppleWebKit/537.77.4 (KHTML,
        like Gecko) Version/7.0.5 Safari/537.77.4"
    clientIpAddress="0:0:0:0:0:0:0:1"  clientHostName="0:0:0:0:0:0:0:1"
    fromDate="1406670784083"  lastUpdatedStamp="1406670784396"/>
<moqui.server.Visitor visitorId="100000"  createdDate="1406670784353"
    lastUpdatedStamp="1406670784363"/>
```

每次点击构件都新建一条记录会有性能影响，并且在繁忙的服务上，数据库的大小会变得非常庞大。这种情况可以使用一个低延迟的插入数据库如 OrientDB 或其他的 NoSQL 数据库来得以缓解。如果你就是想要每隔一个时间段统计性能，并且不需要个体的点击记录进行审计或明细分析，ArtifactHitBin 记录就可以实现这个小花招。

这些记录有在一个时间段内构件点击的概要，在 **binStartDateTime** 和 **binEndDateTime** 之间。这个容器 bin 的长度通过 server-stats.**bin-length-seconds** 属性进行配置并默认为 900 秒（15 分钟）。

这里有个 **create#moqui.entity.EntityAuditLog** 服务点击 bin 的例子。在这个例子中服务被点击/使用了 77 次，总计（累计）运行时间 252ms，这意味着构件在 bin 中平均运行时间为 3.27ms。

```
<moqui.server.ArtifactHitBin  hitBinId="100010"  artifactType="service"
    artifactSubType="entity-implicit"
    artifactName="create#moqui.entity.EntityAuditLog"
    serverIpAddress="172.16.7.38"  serverHostName="DEJCMBA3.local"
    binStartDateTime="1406268616369"  binEndDateTime="1406268636249"
    hitCount="77"  totalTimeMillis="252"  minTimeMillis="1"
    maxTimeMillis="61"  lastUpdatedStamp="1406268636290"/>
```

这些记录可以直接通过工具应用的 **Artifact Bins** 和 **Artifact Summary** 界面从数据库中使用。

构件执行运行时分析

Java 分析工具如 JProfiler 是用于分析 Java 方法性能的伟大工具，但是它对于 Moqui 的构件如界面，转换，服务和实体却无从知晓。Moqui 构件执行门面（Moqui Artifact Execution Facade）在构件运行时，在内存中对每个构件实例（每个 ExecutionContext，例如一个 web 请求等）的性能细节进行追踪。

每个构件运行时创建的 ArtifactExecutionInfo 对象用于保存数据，同时它们会被压入运行堆栈中并在运行历史中保持。你可以使用 ec.artifactExecution.**getStack**() 和 ec.artifactExecution.**getHistory**()方法去访问它们。

从 ArtifactExecutionInfo 实例中你可以获得它自身的运行时（(long **getRunningTime**())，调用它的构件（ArtifactExecutionInfo **getParent**())，它调用的构件（List<ArtifactExecutionInfo> **getChildList**())，由这个构件调用的所有构件的运行时间（long **getChildrenRunningTime**())，以及基于此的构件自身的运行时间（long **getThisRunningTime**()，它的值为 **getRunningTime**() - **getChildrenRunningTime**())。你同样可以通过 **print**(Writer writer, int level, boolean children)方法去打印这些当前构件信息的统计报告，并可选的递归获得其孩子信息。

对于复杂的代码部分，比如一个订单的订购，要处理很多的服务调用从而产生大量的数据。为了更容易的去追踪这个常用的部分，可以使用 ArtifactExecutionInfoImpl 类的方法去生成一个热点的列表：

```
static List<Map> hotSpotByTime(List<ArtifactExecutionInfoImpl> aeiiList, boolean ownTime, String orderBy)
```

这个方法贯穿执行历史中的所有 ArtifactExecutionInfoImpl 实例，同时汇总统计为每个构件创建了一个 Map，包含了这些条目：time, timeMin, timeMax, count, name, actionDetail, artifact type 以及 artifact action。

另一个会产生大量数据的场景是多次运行一个进程，以便获得更好的平均统计。在这种情况下你可能会有数以千计的构件执行信息在历史中。为了整合源于多次运行的数据，到单棵包含了每个构件执行及其子孙信息的树上，使用这个方法：

```
List<Map>  consolidateArtifactInfo( List<ArtifactExecutionInfoImpl>  aeiiList )
```

每个 Map 都有这些条目：time，thisTime，childrenTime，count，name，actionDetail，childInfoList，key（它的格式是：name + ":" + typeEnumId + ":" + actionEnumId + ":" + actionDetail），type 和 action。利用这些结果，通过下面的方法你可以打印出这个缩进排版的普通文本（最好用等宽字体去显示）的树：

```
String  printArtifactInfoList( List<Map>  infoList )
```

使用这些方法的一个例子就是 POP 电商应用中的 TestOrders.xml 界面。它通过如下 URL 进行访问使用，同时展示一个订购和装运指定数量订单的业务代码的性能分析结果的界面：

http://localhost:8080/popc/TestOrders?numOrders=10

这里是源于界面动作脚本的一个片段，使用下面的方法去运行测试代码并获得以上描述的性能指标：

```
def artifactHistory = ec.artifactExecution.history
ownHotSpotList = ArtifactExecutionInfoImpl.hotSpotByTime(artifactHistory, true, "-time")
totalHotSpotList = ArtifactExecutionInfoImpl.hotSpotByTime(artifactHistory, false, "-time")
List<Map> consolidatedList = ArtifactExecutionInfoImpl.consolidateArtifactInfo(artifactHistory)
String printedArtifactInfo = ArtifactExecutionInfoImpl.printArtifactInfoList(consolidatedList)
```

下面的例子是 **Artifacts by Own Time** 部分中订购和装运 25 个订单的界面输出表格上的顶部几行：

Time	Time Min	Time Avg	Time Max	Count	Name	Type	Action	Action Detail
1838	0	2.29	25	801	mantle.order.OrderItem	Entity	View	list
1093	0	1.32	26	825	mantle.ledger.account. GlAccountOrgTimePeriod	Entity	Update	
1025	0	1.08	10	950	moqui.entity.EntityAuditLog	Entity	Create	
844	7	11.25	33	75	mantle.product.PriceServices. get#ProductPrice	Service	All	
686	0	3.43	12	200	mantle.order.OrderPart	Entity	Update	

从这些结果我们发现大部分的时间用于 OrderItem 实体的列表展现（find）操作。这里 **place#Order** 和 **ship#OrderPart** 服务运行的事务缓存被禁用了，同时 OrderItem 实体并未使用实体缓存，所以这次运行中查询进行了 801 次。事务缓存是直写式高速缓存，它将缓存写入的记录并从中进行读取。开启了事务缓存，整个订单呈现大概为 0.8 到 1.4 秒（数据源于我笔记本上使用的 Derby 数据库），同时 **Artifacts by Own Time** 的输出看起来很不同：

Time	Time Min	Time Avg	Time Max	Count	Name	Type	Action	Action Detail
3449	72	137.96	222	25	mantle.shipment. ShipmentServices. ship#OrderPart	Service	All	
1284	0	1.60	10	801	mantle.order.OrderItem	Entity	View	list
679	6	9.05	14	75	mantle.product.PriceServices. get#ProductPrice	Service	All	
614	14	24.56	51	25	mantle.order.OrderServices. place#Order	Service	All	
561	0	0.68	5	825	mantle.ledger.account. GlAccountOrgTimePeriod	Entity	View	one

下面是源于 **Consolidated Artifacts Tree** 部分的一些简单输出。它展示了构件整理过的交叉运行的层次结构，同时还展示了每个运行中的上下文中构件及其子父级构件的数据。当解释这些结果时需注意，每个构件的总次数和时间不仅仅是当前构件作为上级构件的儿子去运行所得到的值，而是这个构件的所有运行信息汇总。最主要的价值就是将使用的最繁忙的构件追踪记录下来，同时正确的了解运行时实际做了什么，特别是指定的服务。

输出的每行遵循下面的的形式进行格式化：

```
[${time}:${thisTime}:${childrenTime}][${count}] ${type} ${action} ${actionDetail} ${name}
```

下面的是样例输出，注意一下，构件名称为了更好的格式化输出，被 eclipse 缩短了：

```
[    16:   3: 13][   2] Screen    View             component://webroot/screen/webroot.xml
| [    13:-41: 54][   3] Screen    View             component://PopCommerce/.../PopCommerceRoot.xml
| | [   165:165:   0][126] Entity    View    one     mantle.product.store.ProductStore
| | [     0:-31263:31263][   3] Screen    View             component://PopCommerce/.../TestOrders.xml
| | | [     3:   3:   0][   3] Entity    View    one     moqui.security.UserAccount
| | | [     5:   5:   0][   1] Entity    View    one     moqui.server.Visit
| | | [     6:   1:   5][   1] Service   Create          create#moqui.security.UserLoginHistory
| | | | [     5:   5:   0][   1] Entity    Create          moqui.security.UserLoginHistory
| | | [ 4700:269:4431][ 75] Service   All             ...OrderServices.add#OrderProductQuantity
| | | | [   632:632:   0][300] Entity    View    list    mantle.order.OrderPart
| | | | [   497:497:   0][375] Entity    View    one     mantle.order.OrderPart
| | | | [   165:165:   0][126] Entity    View    one     mantle.product.store.ProductStore
| | | | [   195:195:   0][ 25] Entity    View    list    mantle.order.OrderHeaderAndPart
| | | | [   328: 21:307][ 25] Service   Create          mantle.order.OrderServices.create#Order
```

```
| | | | | [    146: 12:134][ 25] Service    Create            create#mantle.order.OrderHeader
| | | | | | [    134: 97: 37][ 25] Entity     Create            mantle.order.OrderHeader
| | | | | | | [ 1564:406:1158][950] Service    Create            create#moqui.entity.EntityAuditLog
| | | | | | | | [      83: 83:    0][ 30] Entity     View      one     moqui.entity.SequenceValueItem
| | | | | | | | [      90: 90:    0][ 30] Entity     Update            moqui.entity.SequenceValueItem
| | | | | | | | [ 1025:1025:    0][950] Entity     Create            moqui.entity.EntityAuditLog
| | | | | [    161: 11:150][ 25] Service    Create            create#mantle.order.OrderPart
| | | | | | [    632:632:    0][300] Entity     View      list    mantle.order.OrderPart
| | | | | | [    134: 99: 35][ 25] Entity     Create            mantle.order.OrderPart
| | | | | | | [ 1564:406:1158][950] Service    Create            create#moqui.entity.EntityAuditLog
| | | | | | | | [      83: 83:    0][ 30] Entity     View      one     moqui.entity.SequenceValueItem
| | | | | | | | [      90: 90:    0][ 30] Entity     Update            moqui.entity.SequenceValueItem
| | | | | | | | [ 1025:1025:    0][950] Entity     Create            moqui.entity.EntityAuditLog
| | | | [ 1838:1838:    0][801] Entity     View      list    mantle.order.OrderItem
| | | | [    882:844: 38][ 75] Service    All               ...PriceServices.get#ProductPrice
| | | | | [      38: 38:    0][150] Entity     View      list    mantle.product.ProductPrice
| | | | [ 2324: 83:2241][ 75] Service    Create            ...OrderServices.create#OrderItem
| | | | | [    430:430:    0][575] Entity     View      one     mantle.product.Product
| | | | | [ 2747: 64:2683][100] Service    Create            create#mantle.order.OrderItem
| | | | | | [ 1838:1838:    0][801] Entity     View      list    mantle.order.OrderItem
| | | | | | [ 2482:384:2098][100] Entity     Create            mantle.order.OrderItem
| | | | | | | [ 1564:406:1158][950] Service    Create            create#moqui.entity.EntityAuditLog
| | | | | | | | [      83: 83:    0][ 30] Entity     View      one     moqui.entity.SequenceValueItem
| | | | | | | | [      90: 90:    0][ 30] Entity     Update            moqui.entity.SequenceValueItem
| | | | | | | | [ 1025:1025:    0][950] Entity     Create            moqui.entity.EntityAuditLog
| | | | | | | [ 1784: 89:1695][100] Service    Update            ...OrderServices.update#OrderPartTotal
| | | | | | | | [ 1838:1838:    0][801] Entity     View      list    mantle.order.OrderItem
| | | | | | | | [    322:127:195][250] Service    All               ...OrderServices.get#OrderItemTotal
| | | | | | | | | [ 1838:1838:    0][801] Entity     View      list    mantle.order.OrderItem
| | | | | | | | [    497:497:    0][375] Entity     View      one     mantle.order.OrderPart
| | | | | | | | [ 1204:686:518][200] Entity     Update            mantle.order.OrderPart
| | | | | | | | | [    224:224:    0][200] Entity     View      refresh mantle.order.OrderPart
| | | | | | | | | [ 1564:406:1158][950] Service    Create            create#...EntityAuditLog
| | | | | | | | | | [      83: 83:    0][ 30] Entity     View      one     moqui.entity.SequenceValueItem
| | | | | | | | | | [      90: 90:    0][ 30] Entity     Update            moqui.entity.SequenceValueItem
| | | | | | | | | | [ 1025:1025:    0][950] Entity     Create            moqui.entity.EntityAuditLog
| | | | | | | | [    629: 56:573][100] Service    Update            ...update#OrderHeaderTotal
| | | | | | | | | [    632:632:    0][300] Entity     View      list    mantle.order.OrderPart
| | | | | | | | | [    349:349:    0][450] Entity     View      one     mantle.order.OrderHeader
| | | | | | | | | [    884:592:292][175] Entity     Update            mantle.order.OrderHeader
| | | | | | | | | | [    181:181:    0][175] Entity     View      refresh mantle.order.OrderHeader
| | | | | | | | | | [ 1564:406:1158][950] Service    Create            create#...EntityAuditLog
| | | | | | | | | | | [      83: 83:    0][ 30] Entity     View      one     ...SequenceValueItem
| | | | | | | | | | | [      90: 90:    0][ 30] Entity     Update            ...SequenceValueItem
```

性能优化

一旦一个构件或定义的代码块占用了大量的执行时间，下一步就是回顾检查下并看看其中的哪一部分可以被改进。有时操作就是占用时间并无法为它做更多的事情。哪怕在这些情况下，某些部分可以为异步或使用其他的方式来至少减少对用户或系统资源的影响。

典型的最慢的操作是涉及数据库或文件访问，在内存中进行缓存有助于提升很多性能。Moqui 缓存门面（Moqui Cache Facade）用于框架的很多部分，同时你可以按照需要通过编码直接使用它去进行缓存。Moqui 默认使用 ehcache 进行实际的缓存，并通过 Moqui XML 配置文件进行设置。其他 ehcache 特定的缓存配置可以使用它自己的文件（主要是 ehcache.xml）。在一个应用服务集群中格外需要如此设置一个分布式的缓存。

在运行时开发配置（MoquiDevConf.xml）中，比如实体，服务定义，界面 XML，脚本以及模板构件的缓存都有很短的超时，这样它们会经常的重载，以便于改变文件之后进行测试。在生产配置（MoquiProductionConf.xml）中，为了最优性能全部都使用了缓存。当做性能测试时，确认运行时所有的缓存都开启使用了，例如生产环境的设置，这样测试的数字在生产环境中不会非常缓慢，发生偏移。

资源门面用了很多的缓存。**getLocationText**(String location, boolean cache)方法通过设置缓存参数为 true 去使用 resource.text.location 缓存。其他的缓存总是以编译的形式用在脚本和模板中（可能通过脚本解释器或模板渲染器编译），并甚至通过资源门面用于缓存 Groovy 表达式和字符串表达式。正如上面提到的这些缓存永不“禁用”，但为了有助于运行时的重载，最简单的办法就是对期望的缓存使用一个超时时间。

另一个常用缓存就是通过实体门面管理的实体缓存。缓存用于个别的记录，结果列表以及结果计数。当记录通过实体门面进行新建，修改或删除时，这些缓存会自动的被清理。不管是相对于单表简单的实体还是视图实体都可以被缓存，并且两者都有自动缓存清理工作。为了令缓存清理更有效率，这里默认使用了反向关联的缓存通过实体名称以及记录的主键值去查找缓存的条目。在其他场景（例如新建记录时），它必须在缓存条目上进行一个条件扫描去找到匹配的条目进行清理，特别是列表和计数缓存。查看 ***Data and Resources*** 章节获得更多详细信息。

除了实体读缓存，还有直写式的每事务的缓存，它可以使用 service.**transaction** 属性设置为 cache 或 force-cache 来开启。在 TransactionCache.groovy 文件中有其相关实现。

事务的基本想法是当新建，修改或者删除一条记录时，记录就放在一个事务的对象中保存而不是实际的写入到数据库中。当事务提交但在实际提交之前，将变化的数据写入到数据库中。当进行查询操作时直接使用缓存中的数据值，或者添加数据库中的查询结果到缓存中的值。

它足够智能地知道当进行带约束条件的查询时，只需要匹配 TX 缓存（通过事务进行创建）中的数据值，而根本无需去数据库中进行查找，并且查询全部在内存中进行处理。举个例子，如果你新建一条订单头 OrderHeader 记录以及各种订单条目 OrderItem 记录，那么通过 **orderId** 字段查询所有的订单条目 OrderItem 记录时，它将会查看订单头 OrderHeader 记录是否通过事务缓存进行了创建，并且如果已创建，就会从 TX 缓存中取到订单条目 OrderItem 记录而根本不用去数据库中进行查询了。

对于实体查询操作来说，另一个有价值的工具就是自动最小化的视图实体。当你在一个大的视图实体比如说 FindPartyView 实体上进行查询时，只需要明确选择的指定字段以及限制这些就是你需要的字段即可。实体门面将会自动的查看选中的字段，在过滤条件以及在结

果的排序中使用它们，并且只包含别名的字段以及成员实体必须的字段。通过这种方式，我们就无需使用一个动态视图实体（EntityDynamicView）去有条件的添加成员实体以及别名字段了。回到 FindPartyView 的例子，**find#Party** 服务（在 findParty.groovy 中实现）使用了这种方式，通过极少的代码提供了大量的操作。

一般的准则是，除了绝对的必要，否则不要让数据库进行大量的表数据的查询。将过多的成员实体加入到一个视图实体中是一个很戏剧化，搞笑的形式，因为创建庞大的临时表是一个很昂贵的运算操作。

按照这些原则，另一个常用的方案是查询可能会返回一个非常庞大的结果，然后展示结果时每次只显示一页（比如 20 条记录）。你最好不要选择所有的数据进行呈现，由于主查询中的每条记录都会使得临时表的关联越来越大，并且你在要求数据库去为所有的记录获取值，而不是仅仅你需要展现的 20 条记录。一个更好的方式就是查询一个或几个足以识别记录的字段，然后查询数据进行显示，因为键值是个独立的查询。这样通常会执行的更快，但在一些稀少的情况下不是，因此你可以在你的特殊情境下去测试它们，同时使用实际的数据去变化的查询来看看哪种方式性能最好，这种方式会比较好。

在大量生产环境的电子商务和 ERP 系统中，另一个常见的问题是同步以及锁定延时。这些问题发生在应用服务使用了 Java 的同步机制，或数据库中的锁以及锁等待。你可能也会发现死锁，但那是另一个问题（例如不同的执行）。真正的找到这些问题的唯一途径就是使用负载测试，尤其尽可能的使用相同的资源去进行负载测试，比如像一群相同产品的订单尽可能的接近同一时间去进行处理。

这里有一些途径去优化它们。在 Java 同步机制层面使用非阻塞算法并且数据结构可能会产生巨大的差异，同时很多的类库都这样处理。Brian Goetz 编著的《Java 并发编程实战》（Java Concurrency in Practice）是关于这个话题很好的一本书。

除了这些基本的东西要记住，还有无数优化性能的途径。对于真正重要的通过合理约束的代码，特别是高度使用或通常执行敏感功能的代码，唯一约束其能否更快速运行的问题在于你投入了多少时间和努力在性能测试和优化上。

有时这涉及到象征性的创造并使用非常不同的架构和工具去处理事情，包括像是大量的用户，非常巨大的数据，数据零散在很多的地方等等。对于其中的一些问题，分布式处理或数据存储工具比如 Hadoop 和 OrientDB（以及无数其他当前真正可用的数据库）也许就是你所需要的，甚至使用它们需要更多的努力，但这对于十分特殊的功能来说，这也是十分有意义的。

当使用工具比如 JProfiler 去进行 Java 性能分析时，你通常查看不同类型的事情对于性能的影响多于查看 Moqui 构件运行的性能数据。这里有很多不同的工具和参考用于优化 Java 方法（以及内存使用的类）超过上面提及的那些，可以更好的去优化业务逻辑在一个更高的水平上。

十一、 工具应用

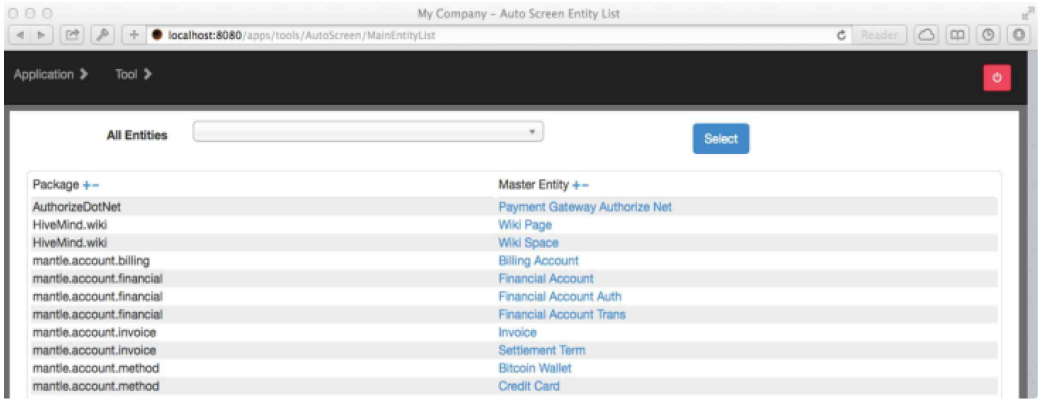
工具应用是默认的 Moqui 运行时的一部分，同时位于 moqui/runtime/component/tools 组件中。它有技术管理界面用于构建在 Moqui 框架上的系统，例如查看和编辑数据，运行服务，管理任务，管理缓存以及查看服务使用的统计。

自动界面

自动界面基于实体定义，同时使用一个由 XML 表单生成的默认表单，其中的自动表单字段基于给定实体的字段所组成。这些界面用于实体记录的查找和创建，编辑已存在的数值以及查看关联的记录。

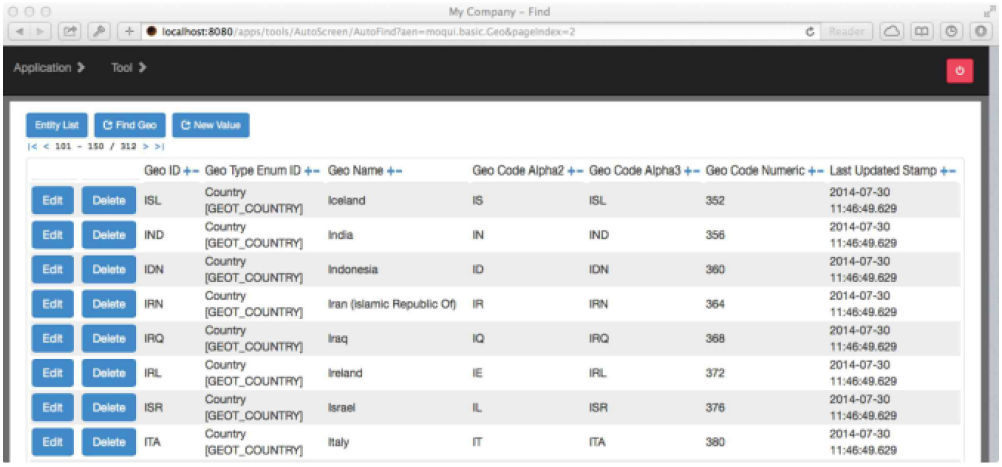
实体列表

自动界面中的主要实体列表有一个可以选择所有的实体加上主实体列表的下拉框在顶部。主实体是有依赖实体的实体并最有用于查找和浏览一组选项卡的依赖和相关实体，尽管任何实体都可用于自动界面。选择一个实体去它的查询页面。

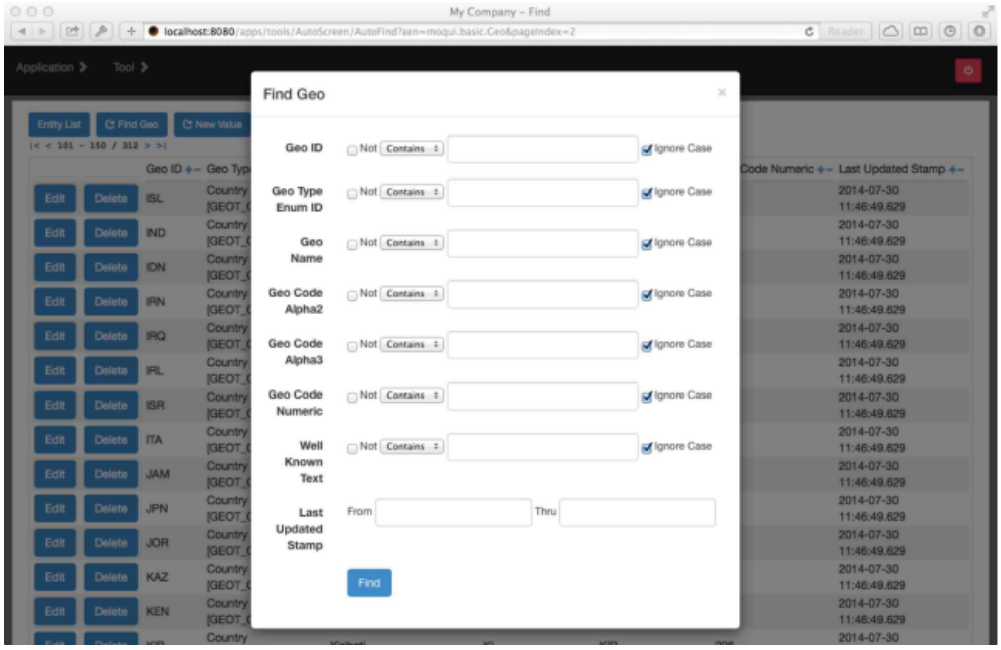


查询实体

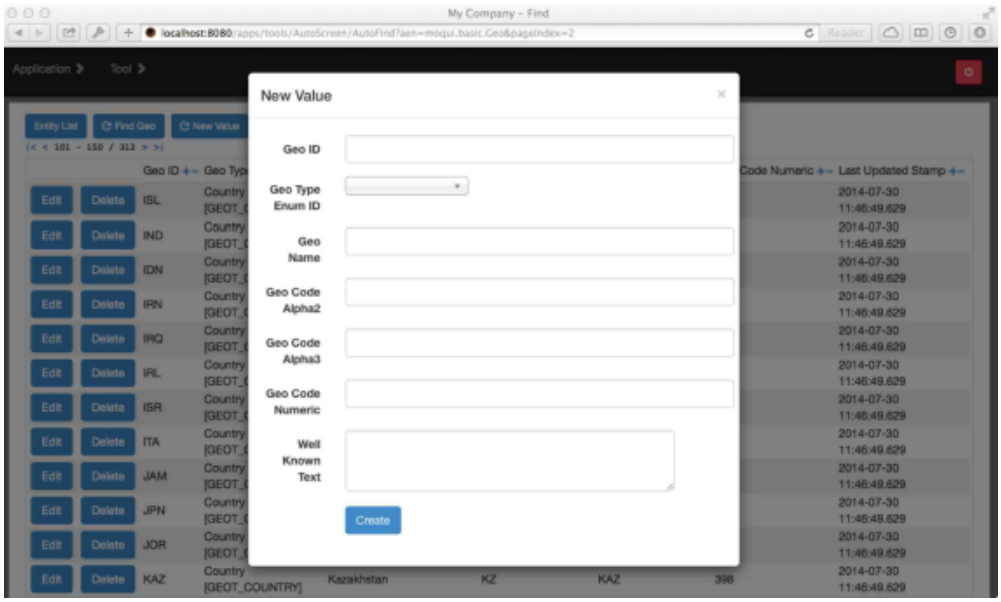
查询界面为选中的实体有一个分页的记录列表，同时每条记录都有编辑和删除按钮，编辑按钮会前往编辑实体的界面。在列表表单中，表格有基于实体字段的自动呈现的字段。实体列表按钮返回主实体列表以及所有的实体。查询按钮弹出一个带有实体字段过滤输入的表单，同时新建值按钮弹出表单去创建一条新的记录。



下面的是 Geo 实体的查询弹出表单。

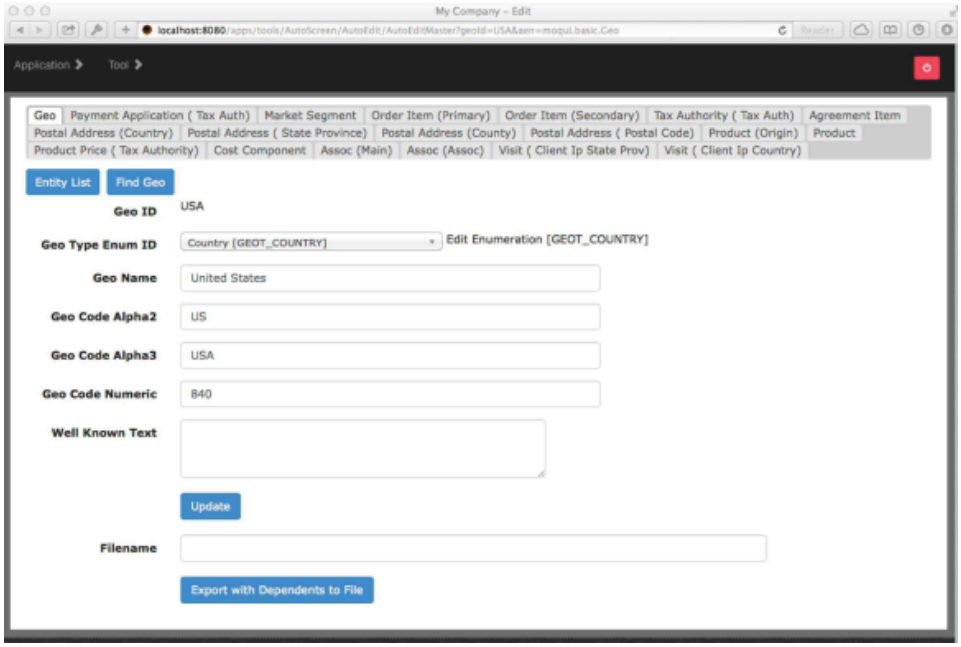


这是 Geo 实体的新建弹出表单。



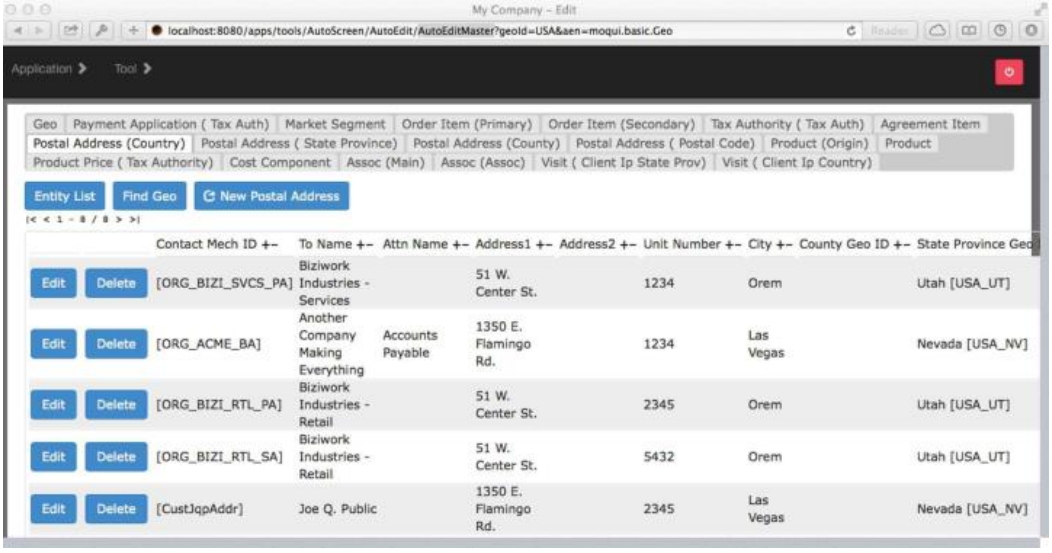
编辑实体

实体编辑界面有选项卡用于当前实体以及所有关联的实体。它有一个基于实体定义自动生成的编辑界面（单个表单），包括外键到其他记录字段的下拉框。这里同样在底部还有个简单的表单用于导出实体及其依赖记录的数据到一个文件中（类似实体导出界面）。这里的是 USA Geo 记录案例：



编辑关联

当你在编辑界面中点击一个关联实体的选项卡时，你会获得一个关联记录的列表，就像实体查询界面一样每行都有编辑和删除链接。它是一个基于实体字段自动生成的列表表单。类似查询界面，你同样可以获得实体列表，查询以及新建值按钮。下面展示的是 Geo（USA）关联的邮政地址记录。

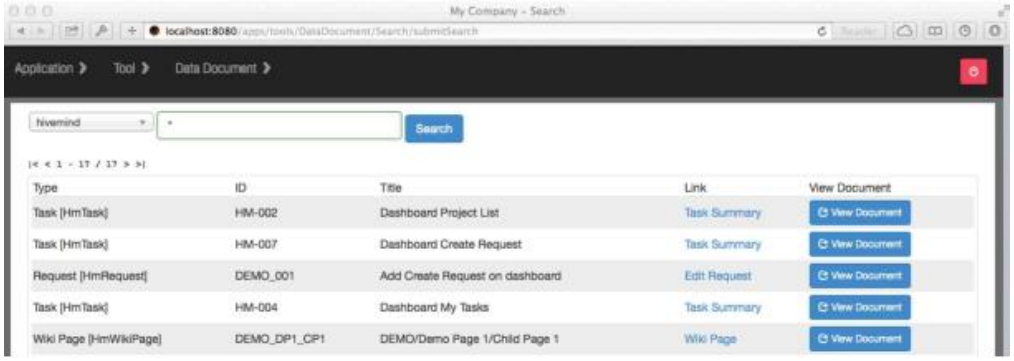


数据文档

实体数据文档在 ***Data and Resources*** 章节的 ***Data Document*** 部分中已涵盖其内容。工具应用中的这些界面允许你去检索文档，为定义的数据供给去建立文档索引，以及导出数据文档到 JSON 文件中。

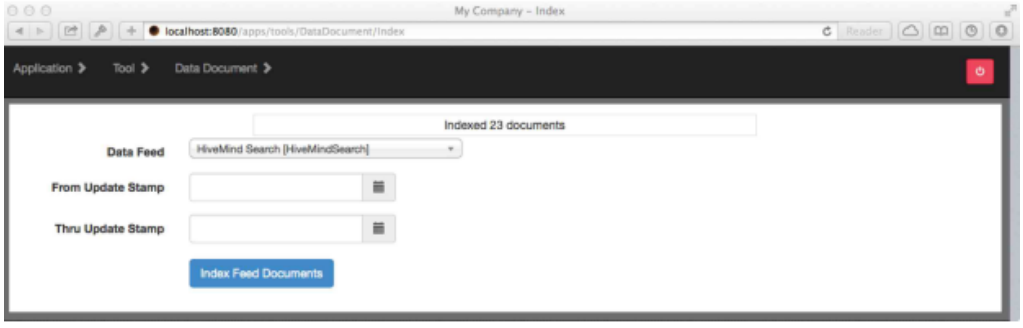
检索

使用检索界面按照索引去查找数据文档，例如本例中的 hivemind 索引。基于 DataDocumentLink 记录的链接将会前往一个关联相应应用文档的界面。查看文档按钮将弹出一个 JSON 文本格式的完整文档以及一个打印文档扁平地图的窗口。



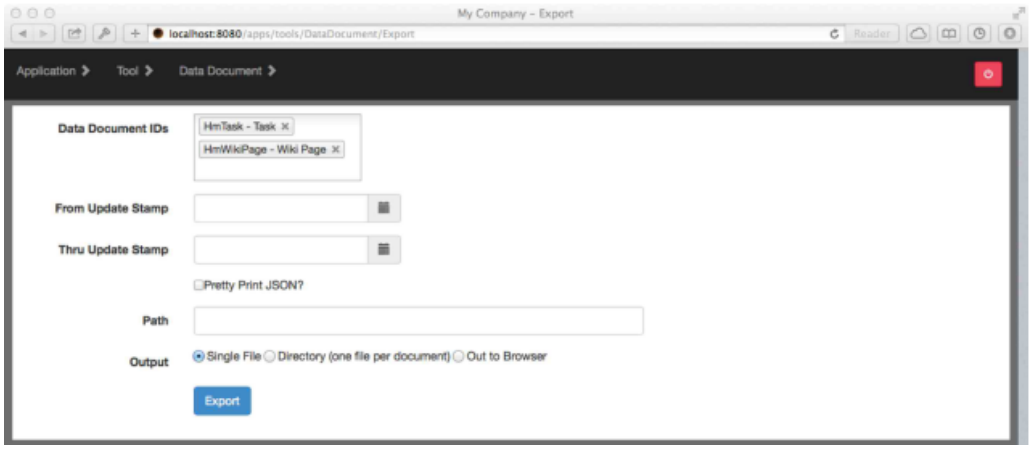
索引

在数据文档索引界面中你可以选择一个数据供给，并可选的去指定起始和截止时间戳，结合实体门面自动添加的 **lastUpdateStamp** 字段去约束文档，然后为所有关联这个数据供给的数据文档去建立索引。



导出

这个界面通过指定的 IDs 以及在起始/截止时间范围内的 **lastUpdateStamp** 字段，导出数据文档到单个文件，一目录的文档文件，或输出到浏览器上。

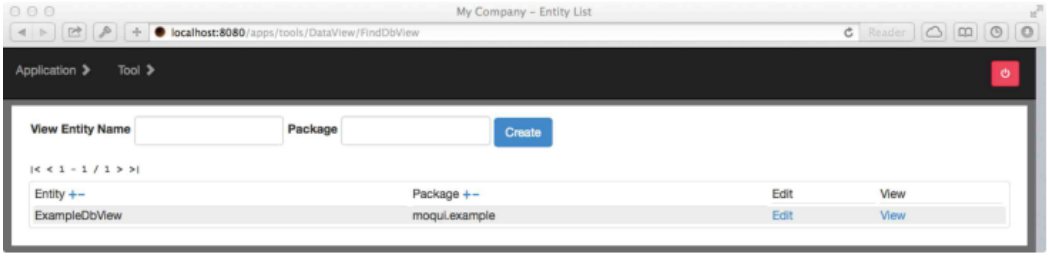


数据视图

数据视图界面用于定义一个简单的视图实体保存在数据库中（使用 DbViewEntity 及其关联实体），然后查看结果并导出它们到一个 CSV 文件。这些界面是个简单用于专门的报表及数据导出的表单，它利用主实体和依赖实体的概念，并通过可选功能去允许主实体上容易的别名字段及所有直接关联的依赖实体进行导出。更详尽的 DB 视图实体可以通过这些界面去定义和查看/导出，但是编辑 DB 视图界面仅支持一个主实体和多个直接关联的实体。

查询 DB 视图

查询界面顶部有个表单去创建一个 DbViewEntity，然后表格中展现所有已存在的 DB 视图实体，同时链接用于编辑和查看它们。



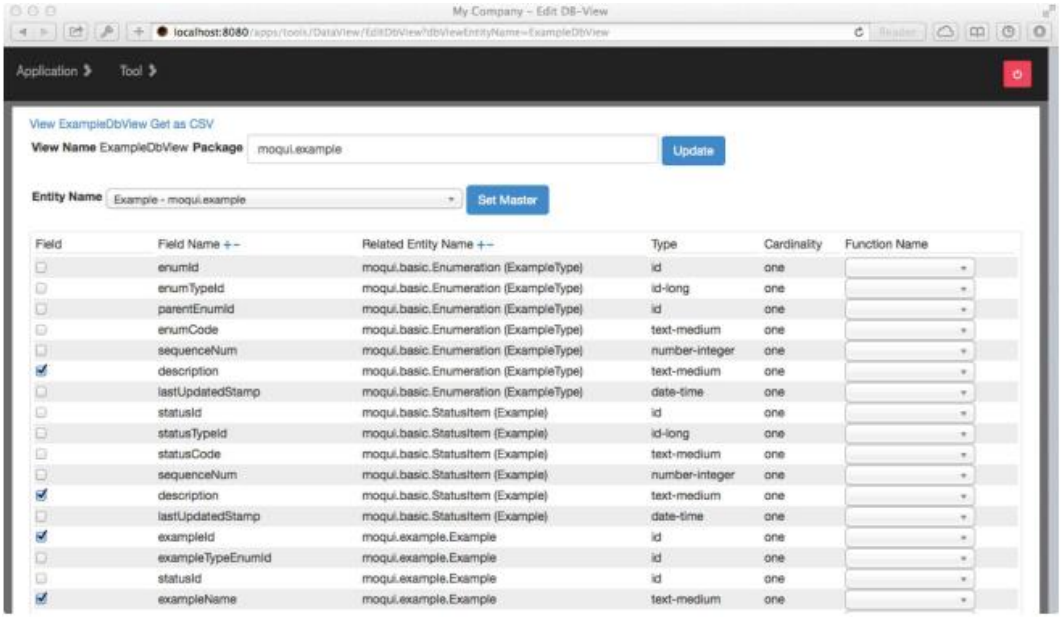
编辑 DB 视图

编辑一个 DB 视图实体的界面顶部有一个表单用于改变实体所在的包。注意定义在 DbViewEntity 中的视图实体可被用在实体门面中，就像其他任何实体或视图实体一样。

界面再往下是一个设置主实体（或视图中主要的、所有其他的实体都将关联的实体）的

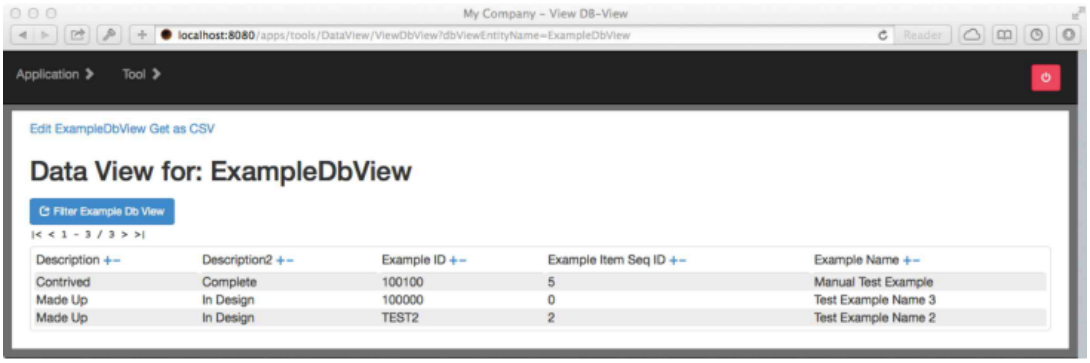
表单。一旦设置了主实体，下面的列表表单将展示所有的实体字段以及直接关联的实体。下面截屏中展示的主实体是 Example 实体及其字段，还有 ExampleType Enumeration 和 Example StatusItem 实体。这里界面只截取了部分下来，如果你查看完整的界面，你将会看到更进一步的字段关联 ExampleContent，ExampleFeatureAppl 以及 ExampleItem 实体（它们都是多的基数）。

视图中选择的字段包括 Enumeration.**description** 和 StatusItem.**description** 字段，源于 Example 实体（主实体）的 **exampleId** 和 **exampleName** 字段，以及界面下面的带有一个合计函数（count）功能去计算条目合计的 ExampleItem.**exampleItemSeqId** 字段。



查看 DB 视图

这个界面展示了查询定义的 DB 视图实体的结果，按需进行分页，并有一个过滤按钮用于弹出一个视图实体上的字段过滤选项的表单（使用了单个表单中的默认自动字段）。这里还有一个链接回到编辑 DB 视图界面，以及一个获得结果 CSV 文件的链接。



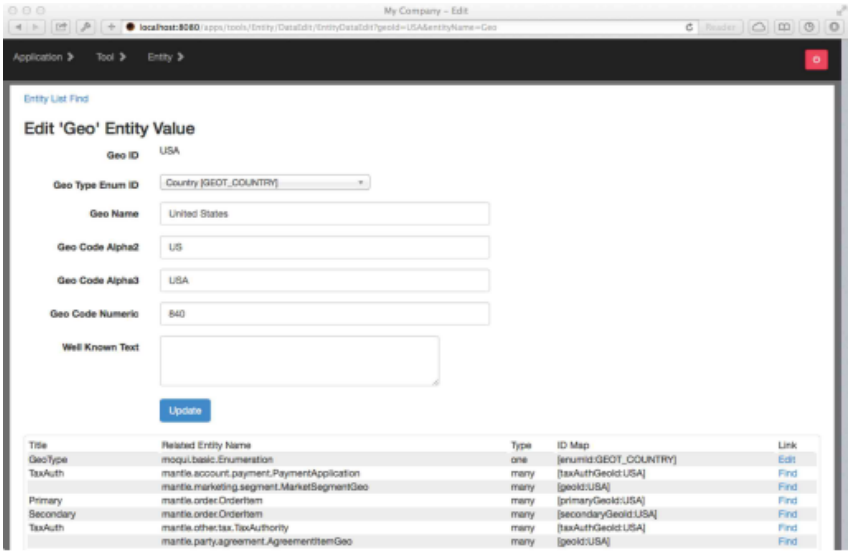
下面是导出源自 ExampleDbView，并与截屏结果相同的 CSV 文件的例子：

```
Description,Description2,Example ID,Example Item Seq ID,Example Name
Contrived,Complete,100100,5,Manual Test Example
Made Up,In Design,100000,0,Test Example Name 3
Made Up,In Design,TEST2,2,Test Example Name 2
```

实体工具

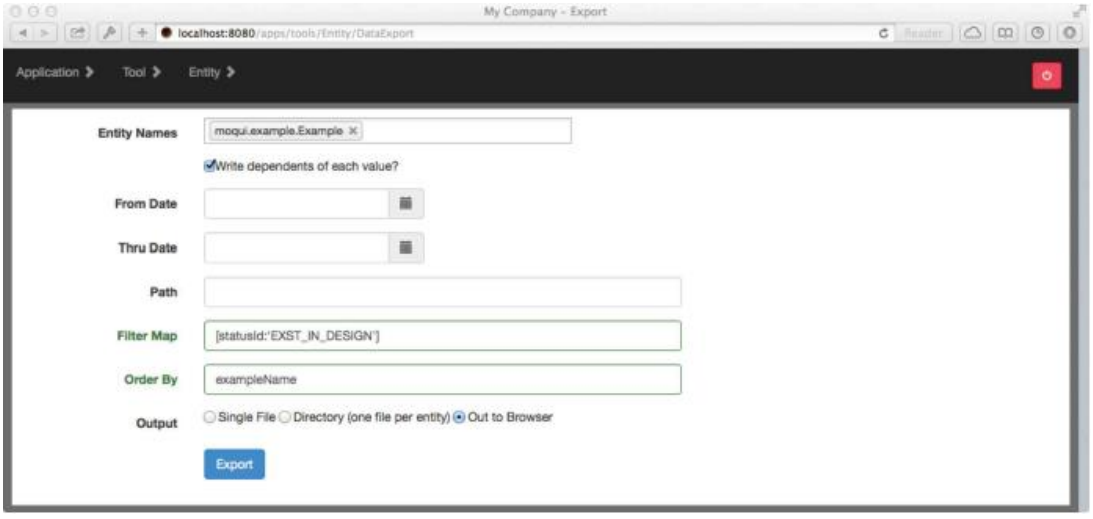
数据编辑

数据编辑界面有点类似自动界面，但没有标签卡集合，作为替代的就如同你这里所见，在实体编辑界面上使用带有链接的关联实体列表，去查找关联当前记录的相关记录。



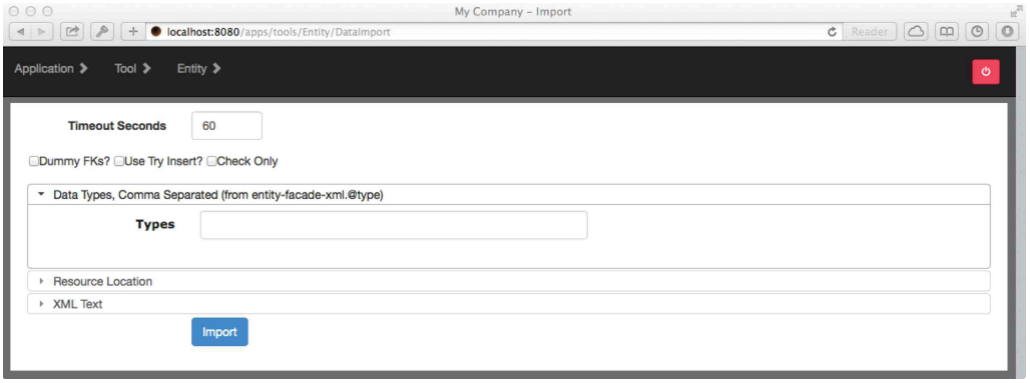
数据导出

这个界面用于导出一个或多个实体 XML 文件的实体数据，或输出到浏览器上。选择一个或多个实体名称，通过起始/截止日期来限制 **lastUpdateStamp**，输出路径或文件名（如果为空则输出到浏览器上），一个可选的 Groovy 语法的 Map 用于过滤（过滤字段仅当匹配实体的字段名称时才应用于实体上，否则忽略），以及可选的逗号分隔的排序字段名称（同样仅匹配字段名称时才适用于实体）。



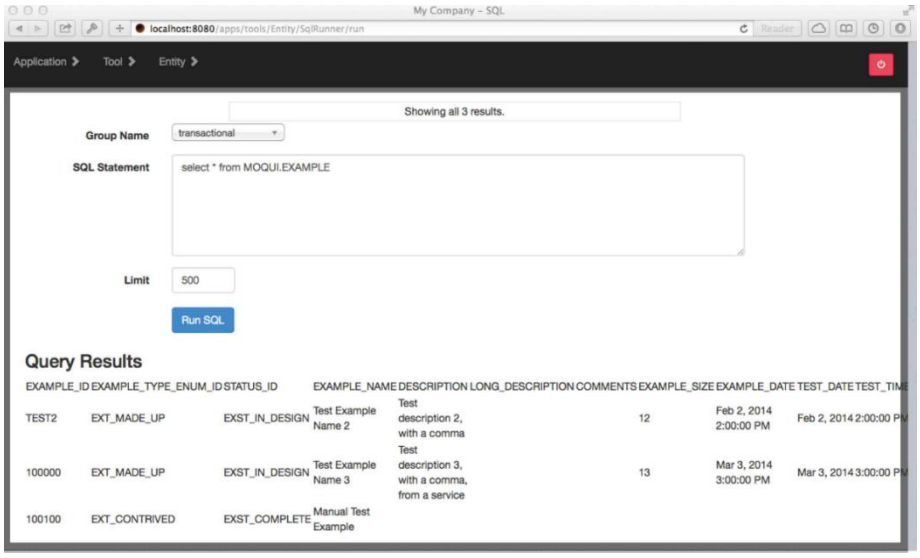
数据导入

使用这个界面从实体 XML 或 CSV 文本中导入数据。这里对于文本本身有 3 个选项：逗号分隔的数据类型（匹配 entity-facade-xml.**type** 属性），一个可以为本地文件名或任何资源门面支持的路径的资源位置，抑或是粘贴到浏览器中文本域里的文本。虚拟外键将检查每条记录的外键，并如果有一条外键对应的记录不存在，则添加一条仅填充主键值的记录。使用尝试插入（Try Insert）意味着原本期望数据是不存在的，同时它将会先尝试插入，并如果失败则进行更新（慢于大部分的更新），而不是插入前检查记录是否存在。检查唯一并不实际加载数据，并替代检查每条记录和报告间的差异。



SQL 执行器

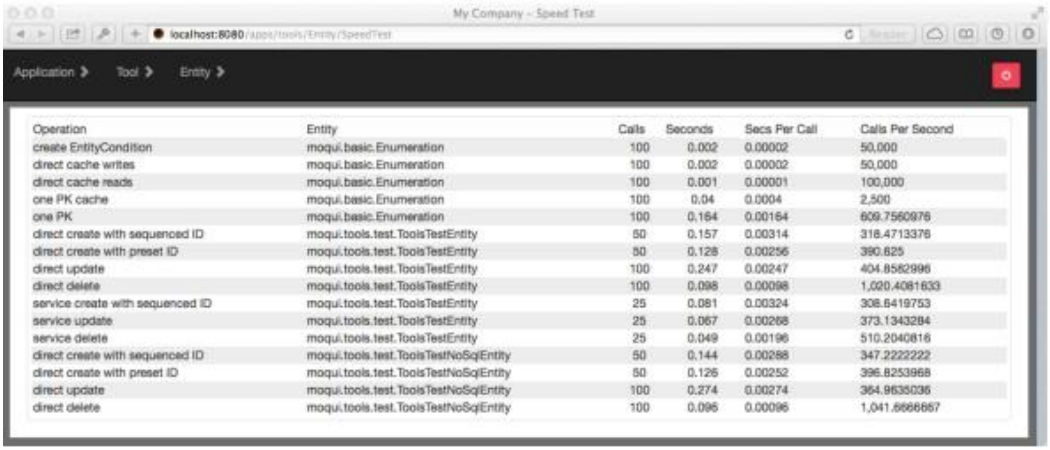
使用这个界面相对于数据库使用给定的实体组来运行任意的 SQL 语句，并显示结果。



速度测试

这个界面运行了一系列的缓存和实体操作去报告定时的结果。它对于比较不同的数据库和不同服务器配置的性能来说是最有用处的。界面接受一个默认为 100（如下所示）的基本

调用参数。注意一下，这个截屏界面在 Derby 数据库中使用了“nosql”实体组及其它所有实体组的默认配置。当使用 OrientDB 或其他一些 NoSQL 数据库时，你会看到相当不同的结果。

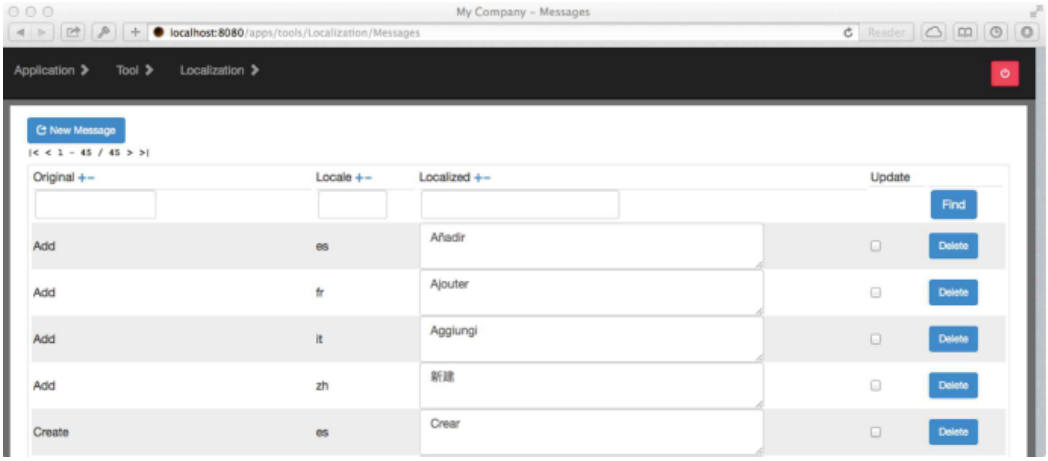


Operation	Entity	Calls	Seconds	Secs Per Call	Calls Per Second
create EntityCondition	moqui.basic.Enumeration	100	0.002	0.00002	50,000
direct cache writes	moqui.basic.Enumeration	100	0.002	0.00002	50,000
direct cache reads	moqui.basic.Enumeration	100	0.001	0.00001	100,000
one PK cache	moqui.basic.Enumeration	100	0.04	0.0004	2,500
one PK	moqui.basic.Enumeration	100	0.164	0.00164	609.7560976
direct create with sequenced ID	moqui.tools.test.ToolsTestEntity	50	0.157	0.00314	318.4713376
direct create with preset ID	moqui.tools.test.ToolsTestEntity	50	0.128	0.00256	390.625
direct update	moqui.tools.test.ToolsTestEntity	100	0.247	0.00247	404.8582996
direct delete	moqui.tools.test.ToolsTestEntity	100	0.098	0.00098	1,020.4081633
service create with sequenced ID	moqui.tools.test.ToolsTestEntity	25	0.081	0.00324	308.6419753
service update	moqui.tools.test.ToolsTestEntity	25	0.067	0.00268	373.1343284
service delete	moqui.tools.test.ToolsTestEntity	25	0.049	0.00196	510.2040816
direct create with sequenced ID	moqui.tools.test.ToolsTestNoSqlEntity	50	0.144	0.00288	347.2222222
direct create with preset ID	moqui.tools.test.ToolsTestNoSqlEntity	50	0.126	0.00252	396.8253968
direct update	moqui.tools.test.ToolsTestNoSqlEntity	100	0.274	0.00274	364.9635036
direct delete	moqui.tools.test.ToolsTestNoSqlEntity	100	0.096	0.00096	1,041.6666667

本地化

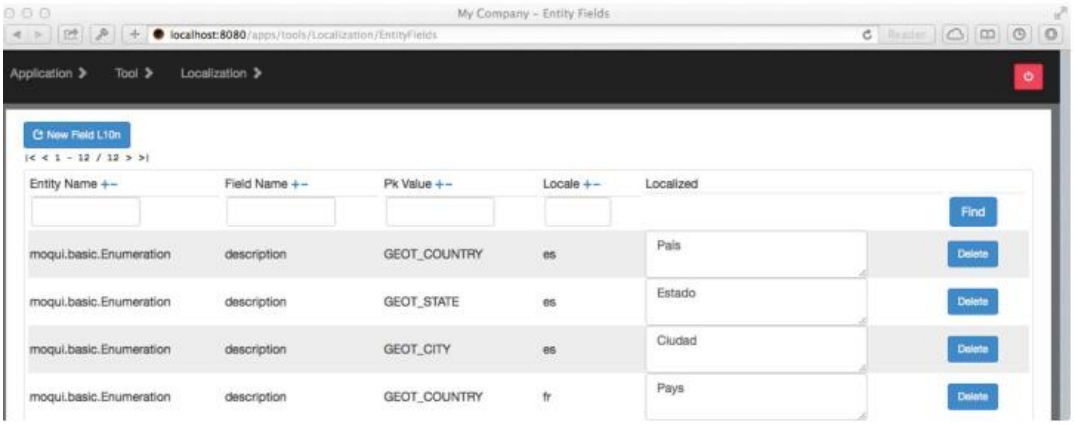
信息

Moqui 使用数据库记录替代 XML 文件的属性去本地化信息和标签。使用这个界面管理的本地化信息会被 L10nFacade.**getLocalizedMessage**( )方法使用，资源门面会在字符串扩展及界面 XML 和表单的标题等呈现之前，轮流地使用这个方法。



实体字段

EntityValue.get( )方法支持本地化任何实体的字段，只要简单的设置 field.enable-localization 属性为 true，同时在这里添加记录即可（记录在 LocalizedEntityField 实体中）。每条记录都有实体名称，要本地化的字段名称，单一字段主键的值（仅有唯一主键字段的实体才可以使用），数值的所在区域以及本地化的数据。

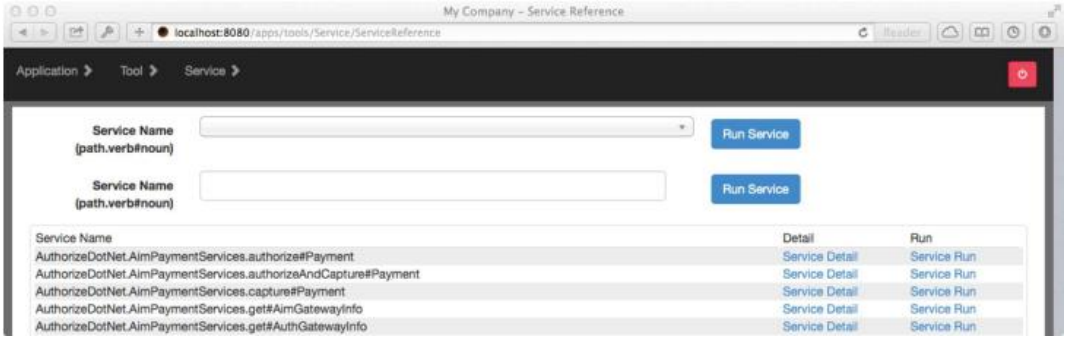


服务

服务参照

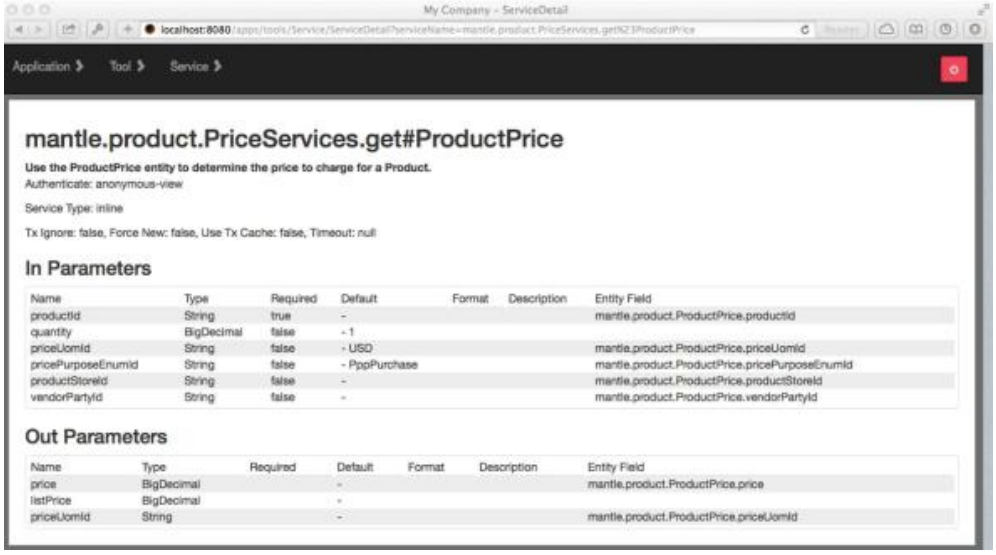
- ***服务列表***

结合服务的引用，你可以看到一个已存在的服务列表，每个服务的明细，同时也可以前往一个界面去运行它们。



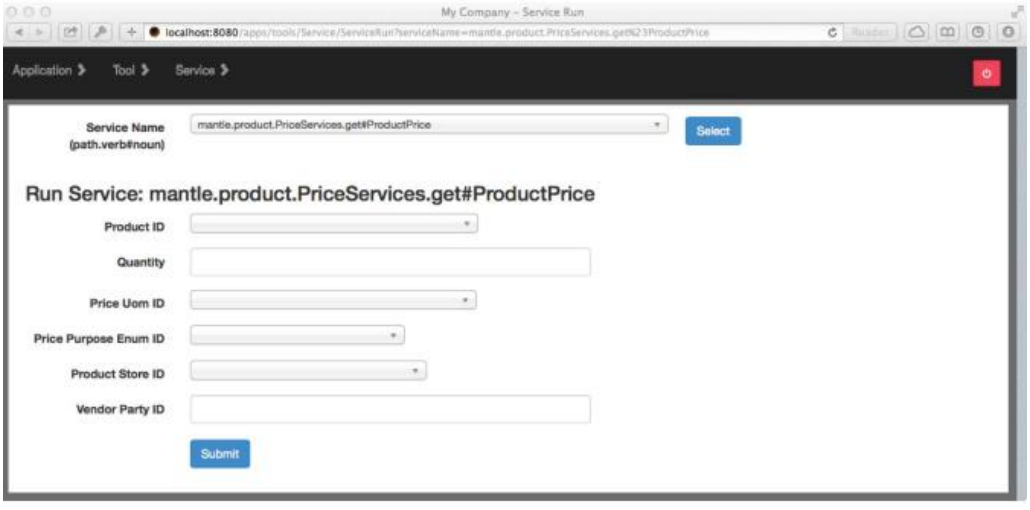
- ***服务明细***

服务的明细界面展示了这个服务的描述和关于服务的一般信息，加上明细的出入参细节。这对于普通引用和当实现接口时查看服务如何在运行时进行扩展等来说，十分有用。



运行服务

服务运行界面展示了一个基于服务定义自动生成字段的单个 XML 表单，当服务的入参关联于实体字段时（为关联的实体选择下拉框的值等），它工作的最好。简单的输入/选择数值并提交去运行这个服务，同时查看结果。

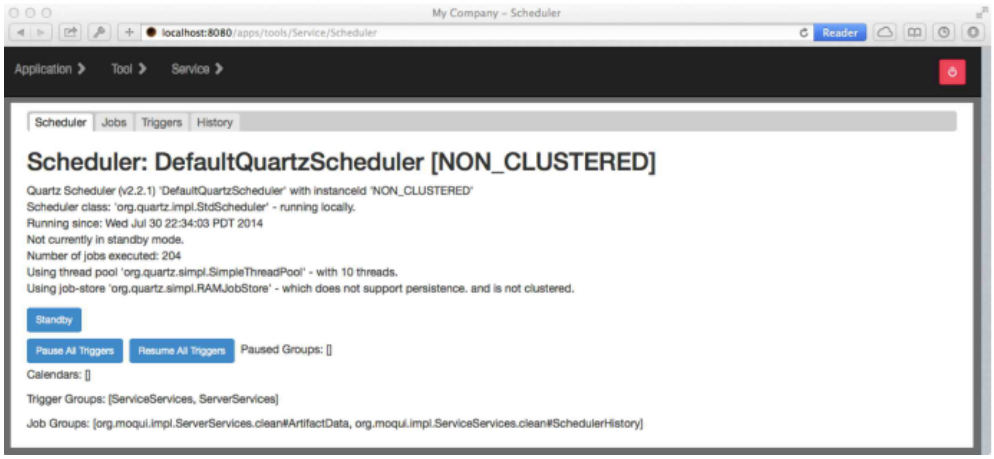


调度器

Moqui 框架使用 Quartz 调度器去运行已排程和异步的服务及任务。这些界面用于查看如暂停、恢复任务及触发器等，关于调度、已排程的任务和执行管理的信息。

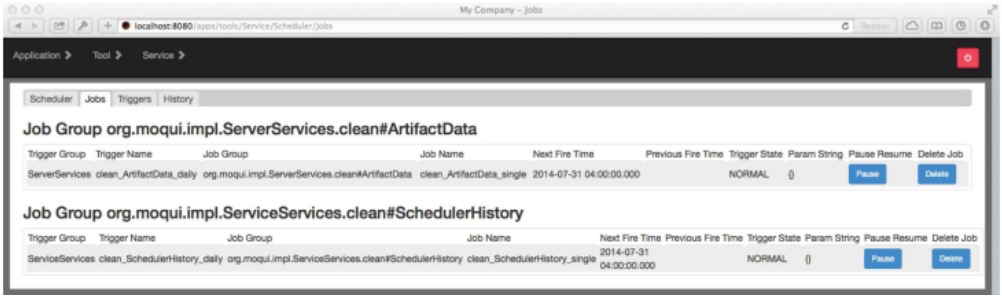
- ***调度状态***

这个界面展示了 Quartz 调度器的状态，同时有按钮用于将整个调度器设置为待命，以及暂停和恢复所有的触发器。



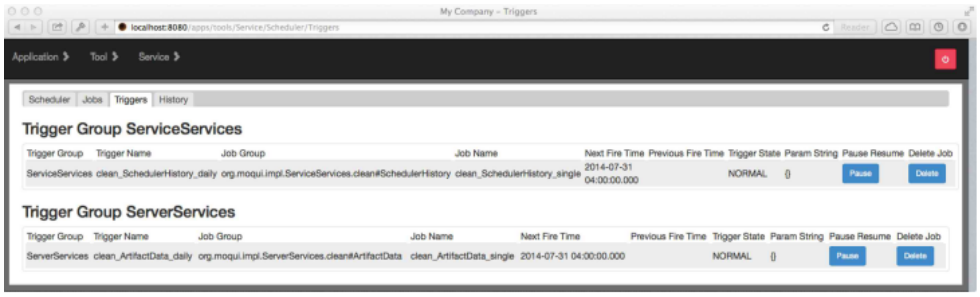
- ***任务***

任务选项卡展示了当前活动的任务，通过与服务同名的 Moqui 服务任务组进行组织。除了任务明细之外，还有额外的按钮用于暂停任务，或当暂停时恢复任务，以及删除任务。当暂停一个任务时，将会暂停所有任务关联的触发器。



- ***触发器***

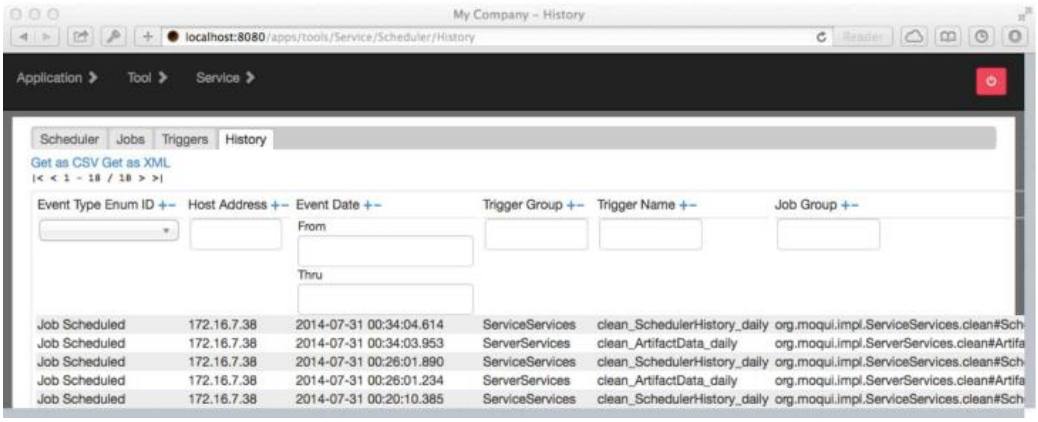
和任务选项卡非常类似，这个选项卡展示了任务关联的触发器并有相同的选项用于暂停/恢复及删除。一个任务可能有多个触发器，通过这个界面你可以暂停/恢复一个任务中明确的触发器同时其他的保持原样。



➢ ***历史***

调度历史选项卡展示了一个包括排程的服务及其他任何你可能运行的自定义任务，的任务运行的历史。这里有链接去获得 CSV 或 XML 文件格式的数据。列表表单的头部有选项去过滤结果，同时这里还有分页，因为可能会有大量的任务。

数据来源于 SchedulerHistory 实体，并通过实现了 Quartz 的 SchedulerListener 接口的 ServiceFacadeImpl.HistorySchedulerListener 类进行管理。

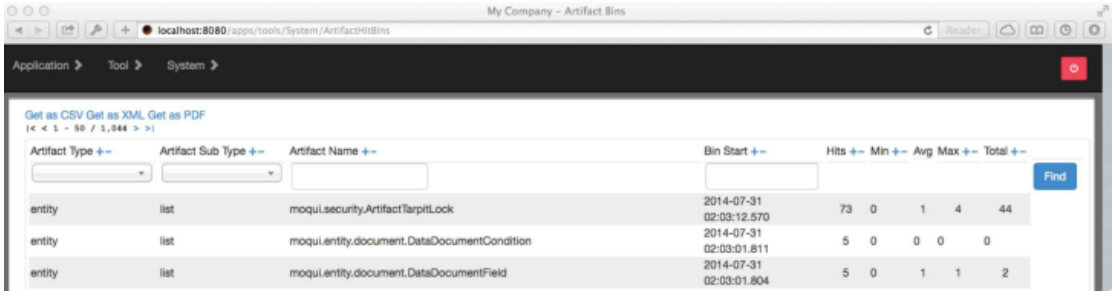


系统信息

构件统计

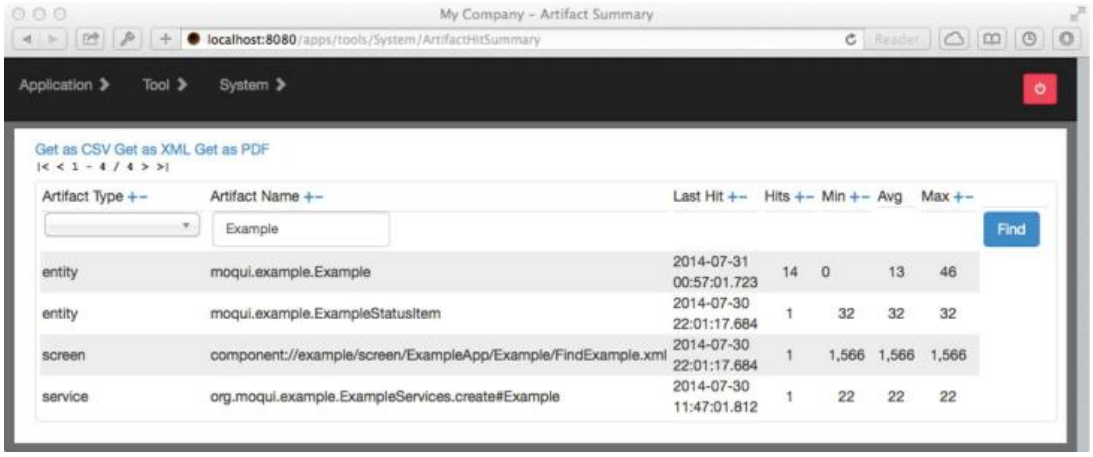
➢ ***点击次数统计***

这个界面展示了源于 ArtifactHitBin 实体的记录，同时有选项用于过滤，排序及导出到 CSV，XML 和 PDF。使用这个界面去查看关于指定构件在指定日期/时间范围内的构件点击数据。



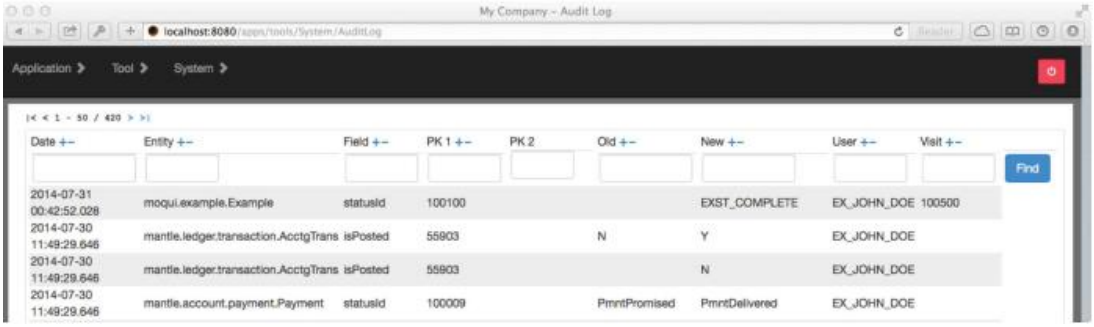
➢ ***构件概要***

构件概要界面基于 ArtifactHitBin 记录并使用 ArtifactHitReport 视图实体，去展示每个构件在所有时间下的一般性能数据。就像点击统计界面一样，这个界面有过滤，排序以及导出选项。下面的截屏仅展示了使用头部过滤表单通过名称过滤出来的“Example”构件的信息。



审计日志

当 field.enable-audit-log 属性设置为 true，实体门面将会追踪 EntityAuditLog 记录的变化。使用这个界面可以查看到这些记录。



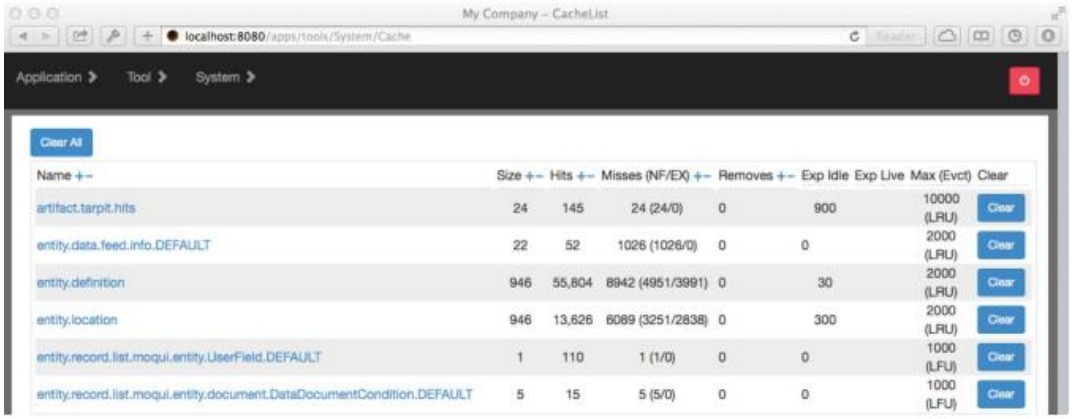
缓存统计

➢ ***缓存列表***

Moqui 缓存门面用于缓存整个系统，包括资源，实体，以及其他各种缓存。使用这个列表去查看每个缓存的明细总结。Size 是缓存中元素的数量。Hits 是成功访问的缓存。Misses 包括普通缓存错过（从缓存中未成功的取到）、指定的未找到（NF）以及过期（EX）错过的合计。Removes 展示了明确从缓存中移除的总数。

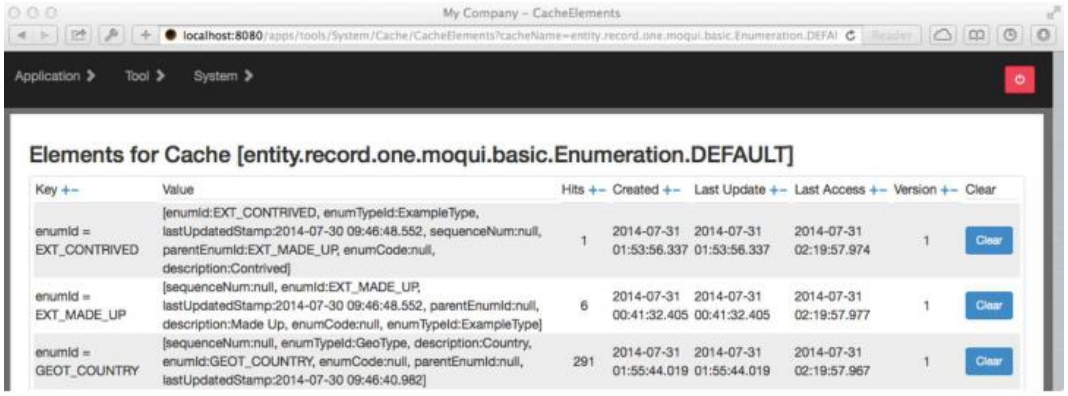
这里有两个过期时间可以进行配置：在闲置了一个明确时间之后的闲置到期时间和自缓存元素创建开始的存活时间。Max (Evct)列展示了每个缓存的最大元素数量（默认为 10,000），同时一旦达到限制将会使用回收算法。每个缓存的清理 Clear 按钮仅清理本缓存，同时顶部的清理所有（Clear All）按钮将清理所有的缓存。点击 Name 列中的数据可以去查看缓存中

的元素。



➢ ***缓存元素***

当你点击缓存的名字时，你将会看到这个界面。它展示了上限为 500 的缓存条目（使用 **displayLimit** 参数去指定一个不同的限制）。界面中有每个缓存元素的细节，加上一个按钮用于从缓存中清理（移除）这个元素。这个截屏是某个实体的一个缓存（Enumeration 实体）。key 和 value 列中展示的文本是通过调用对象上的 **toString()**方法获得的。本例中的 key 是一个 EntityCondition，value 是一个 EntityValue，并且它们都展示成良好的文本，但不是所有的对象都可以这样。

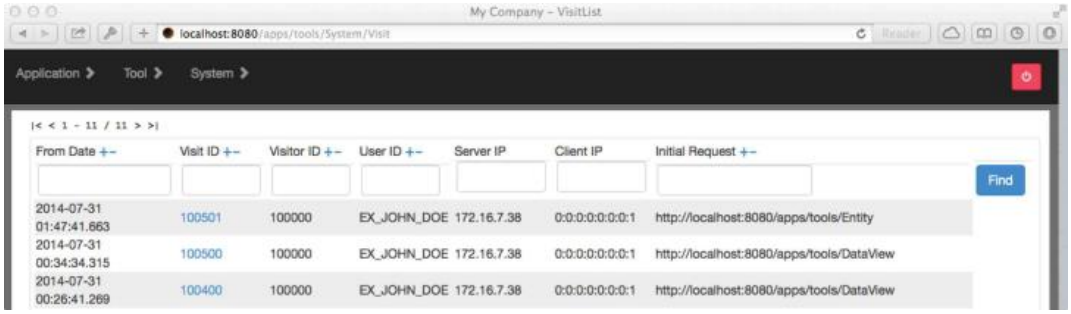


服务访问

Moqui 为每个 web 会话都创建一条 Visit 记录去追踪服务访问并将会话中的构件点击（作为界面的页面请求，内容，转换，服务等）绑定在一起。

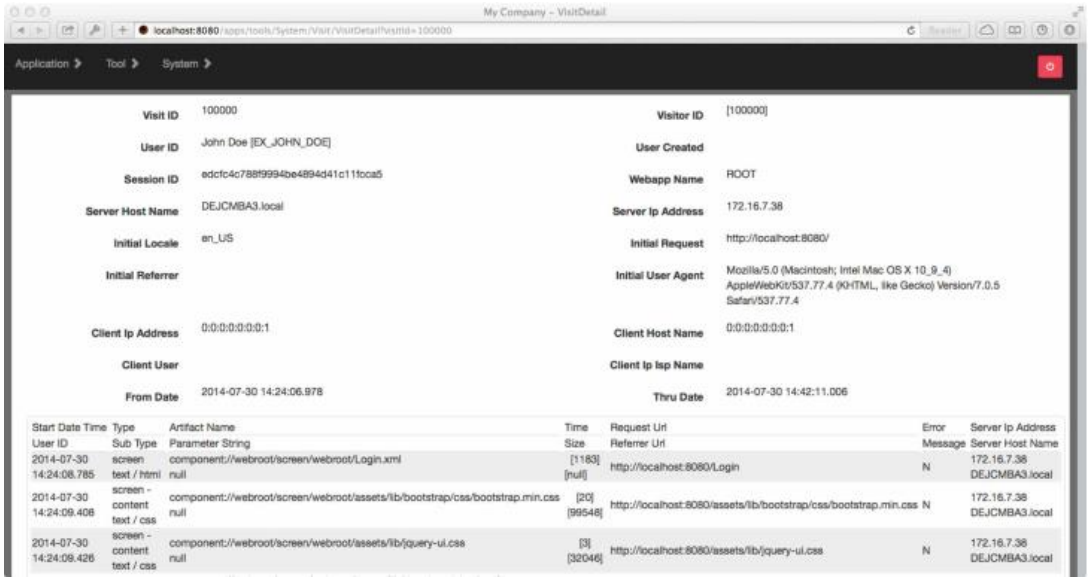
- ***访问列表***

这个界面展示了一个带分页和带过滤，排序记录选项的访问列表，因为随着时间的发展这里会有非常大数量的访问。点击访问 ID 去查看访问的细节。



- ***访问明细***

这个界面展示了访问（会话）的明细。头部的字段常用于一个 HTTP 请求中，加上额外的信息，如访问期间登录的用户 ID（如果一个用户登录了）。它同样展示了访问关联的构件点击（如会话中的页面请求等）。这可用于特定的用户为了安全和服务的目的，去查看一个活动的历史，同时下面 Visit 和 ArtifactHit 中的数据可用于更为普通的为之前目的的分析以及营销。



十二、 地幔业务构件

地幔业务构件是一个独立于 Moqui 框架构建的开源项目。Moqui 框架是一个构建应用的工具集。地幔业务构件是一个企业级应用的基础的低层级的构件库。使用地幔构件最大的好处就是减少了保存、设计和实现的风险，采用了通用的、标准化的业务结构和业务过程，能够保证与其他构建于 Moqui 和地幔的应用的一致性。

地幔有三个部分：通用数据模型（UDM），通用业务服务库（USL）以及通用业务过程库（UBPL）。本章节主要关注于数据模型（UDM）和服务模型（USL）。

UBPL 是一些业务过程的故事以及一些通用的用于企业应用设计的业务需求文档。它们是理解业务概念，参与者以及业务过程的很好的源头，数据模型和服务库都基于 UBPL。它们同样也是个普遍的真实世界业务的起点，并且可以按照需要进行修改。

地幔是一个构建企业自动化应用的基础，它适用于：

- ✧ 企业资源计划（ERP）
- ✧ ERP 项目
- ✧ 专业服务自动化（PSA）
- ✧ 客户关系管理（CRM）
- ✧ 供应链管理（SCM）
- ✧ 生产资源计划（MRP）
- ✧ 企业资产管理（EAM）
- ✧ 零售终端（POS）
- ✧ 电子商务

Moqui 框架和地幔业务构件共同形成了一个完整的应用生态系统。应用系统可以扩展地幔的数据模型并且使用它们自带的服务，使用数据模型和服务意在使本应用系统能够更方便的与其他相同构建方式的应用进行数据和服务的协调工作。

当这样的应用系统被部署在一起的时候，数据就自动共享了。举个例子，你将在所有的电子商务，客户服务，项目管理，会计系统和其他任何类型的系统中使用同一个结构的客户数据。

注意：本章将使用一大批业务构件。如果你运行这些你并不熟悉的构件，你可以浏览翻看、关注下它们并继续使用，而无需担心使用它们。**地幔结构和 UDM** 部分作为数据结构上下文运行在很多的业务构件中。当你使用 **USL 业务过程**部分时，你会通过例子的运行发现业务构件执行的来龙去脉，特别是你花费了一定的时间去阅读了数据结构。

地幔结构和 UDM（通用数据模型）

地幔通用数据模型（UDM）基于 Len Silverston 所著的《数据模型资源手册》卷一和卷二的业务概念。作为这部分的补充，这两本书是组件地幔通用数据模型（UDM）的建模概念的基础。UDM 是对书中的数据模型概念的宽松实现。UDM 中包含了一部分超出书中定义的实体，并完善了它们（例如报价单和销售单）。

UDM 和 USL 都遵从于同样的组织业务构件的设计模式。目录规划和文件结构也同样按照这种模式进行设计。

下面的部分是书中每一块业务的结构和实体的摘要，它们按照字母的排序以便于展示和

参考。当你初次开始理解这些数据模型时，我的建议是按照下面的顺序先去了解下最基本的实体概念：

- ✧ **数据和资源**章节中的**数据模型模式**
- ✧ 当事人 (mantle.party)
- ✧ 联系机制 (mantle.party.contact)
- ✧ 设施 (mantle.facility)
- ✧ 定义 – 产品 (mantle.product)
- ✧ 资产 – 资产 (mantle.product.asset)
- ✧ 会计 – 发票 (mantle.account.invoice)
- ✧ 会计 – 支付 (mantle.account.payment)
- ✧ 工作投入 (mantle.work.effort)
- ✧ 订单 (mantle.order)
- ✧ 采购 (mantle.shipment)

数据模型图中只选择了业务中关键的数据实体结构以及部分关键的字段进行了图解说明。图中并没有完整的罗列出所有的实体和字段。在这些图示中，主实体使用了蓝色的边框，明细实体使用了紫色的边框，同时关联实体使用了绿色的边框。

会计

➢ *会计 - 计费账户 (mantle.account.billing)*

一个计费账户 BillingAccount 是通过聚集发票（Invoice）和支付（Payment）记录，以达到追踪客户(**billToPartyId**)对卖家(**billFromPartyId**)负债情况的目的。财务中负债的结算是使用未支付发票的总额减去已支付的总额。已支付总额可能会大于发票总额，这种情况下财务结算对于客户(**billToPartyId**)就是一个正的负债。计费账户 BillingAccount 可能会有一个信用限额，这里用 **accountLimit** 字段记录，同时这个字段关联的货币类型字段为 **accountLimitUomId**。

BillingAccount 本身作为发票 Invoice 和支付 Payment 记录中账户的“交易”明细非常的简单。计费账户可以使用 BillingAccountParty 来关联联系人，也可以用 BillingAccountTerm 来关联条款。

➢ *会计 - 金融账户(mantle.account.financial)*

金融账户 FinancialAccount 类似于银行账户一样是一个单入口的结算账户。金融账户通过金融账户类型实体（FinancialAccountType，比如：是否可退款（isRefundable），需要 Pin 码（requirePinCode），自动补充设置等）被定义成很多的种类。开箱即用的类型包含了礼品券，商店信用账户，服务信用账户，贷款账户以及银行账户。

金融账户 FinancialAccount 的所有者是一个当事人 Party（**ownerPartyId**），并且也可以是对此账户有结算权责的内部组织部门（**organizationPartyId** 来记录）。其他当事人可能会通过 FinancialAccountParty 来关联此账户。金融账户有一个名称（**finAccountName**），编码（**finAccountCode**），也许有一个 PIN 码（**finAccountPin**）。金融账户可能有一个时间区间（**fromDate**, **thruDate**）的有效期，同时有一个状态标识（**statusId**）区分激活，消极等待补给，人工冻结或者注销。

FinancialAccount 的实际结算 **actualBalance** 是所有金融账户关联的金融账户事务

（FinancialAccountTrans）的总和。金融账户有效结算是实际结算 **actualBalance** **减去**授权的账户（FinancialAccountAuth）的合计。

一笔指定的金额 **amount** 在金融账户事务（FinancialAccountTrans）中可能是存款（Deposit），提款（Withdraw）或者调整（Adjustment），通过 **finAccountTransTypeEnumId** 来记录。金融账户事务由于需要进行批准或者其他原因有个 **statusId** 状态字段去表示创建，批准和撤销状态。同时，**reasonEnumId** 字段记录了金融账户事务的缘由是购买（Purchase），首次开户存款（Initial Deposit），存钱（Replenishment）还是提款（Refund）。

金融账户事务发生在某个确定的时间（**transactionDate**），同时可能登记是在另外个时间（**entryDate**）。事务的执行和初始化都是某个当事人（**performedByPartyId**）发起的，并可能会有相应的注释。同时，它也通常有一个支付 Payment（**paymentId**）或者订单条目 OrderItem（**orderId**, **orderItemSeqId**）与之相关联。

金融账户授权（FinancialAccountAuth）是用于存储一个持续的退款事务的总额 **amount**。授权起始于授权时间（**authorizationDate**），结束于到期时间（**authorizationDate**）。

- ***会计 - 发票 (mantle.account.invoice)***

一个发票 Invoice 或者票据是用于借方（**fromPartyId**）当事人 Party 向贷方（**toPartyId**）当事人 Party 请求的支付 Payment 的明细。有很多种发票类型（**invoiceTypeEnumId**）包括：销售单，退货单，工资单，佣金以及样板。借方和贷方当事人身份决定了发票的方向，所以购买和销售是不可分离的类型，它们都是伴随着当事人以某种形式进行的销售类型的发票。

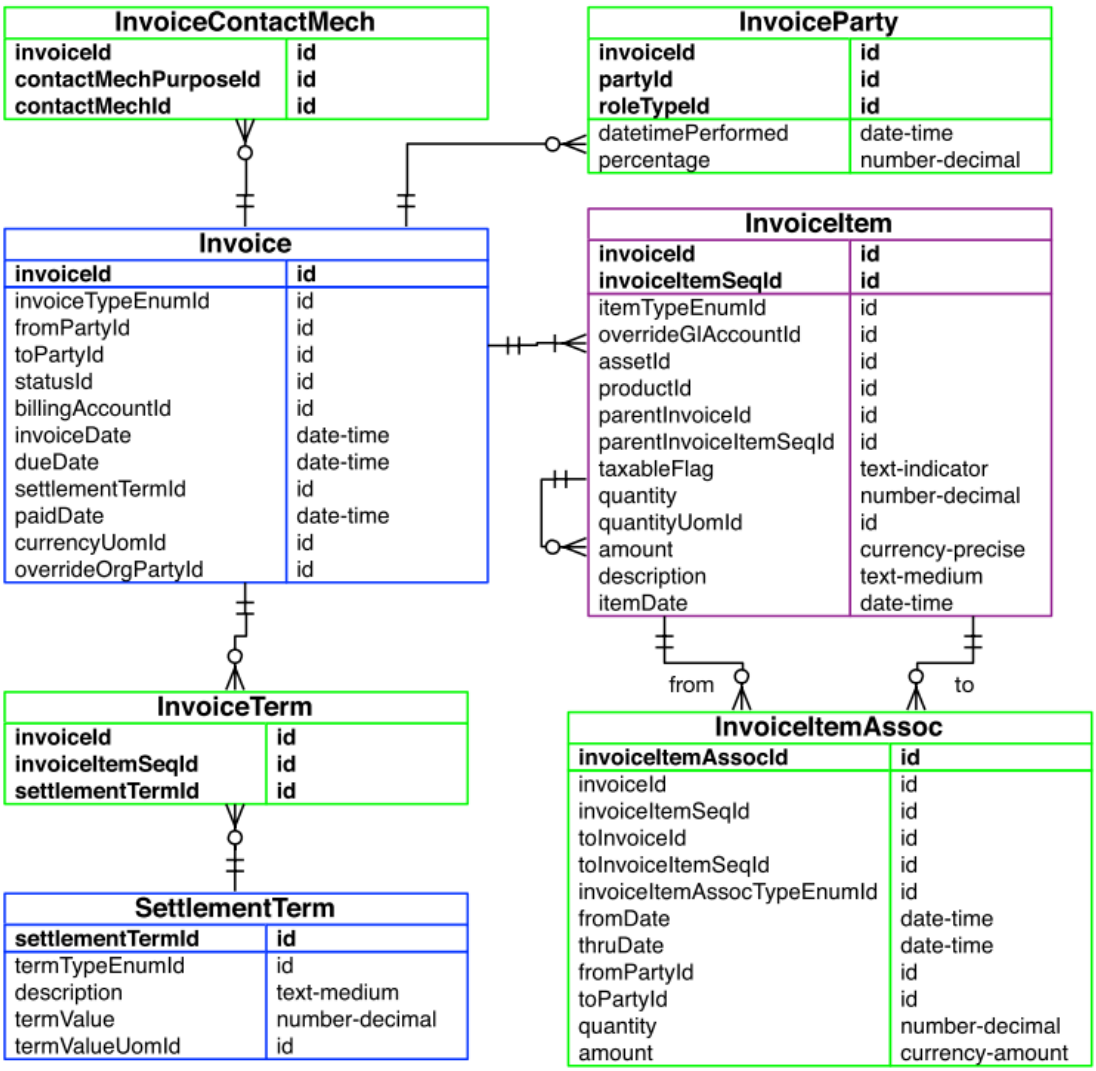
按照发票的方向，内部的组织当事人 Party 有很多种状态类型（**statusId**）。传入的发票状态有传入，接收，核准，已支付，计费通过以及撤销。寄出的发票状态有处理中，最终敲定，发送，收到付款，勾销和撤销。

发票可能会关联一个计费账户 BillingAccount（**billingAccountId**），本章的 ***Account - Billing (mantle.account.billing)***部分有详细描述。货币计量单位（**currencyUomId**）字段标识了发票的总额单位。发票初始化时有个明确的开票时间（**invoiceDate**）和应付款时间（**dueDate**）用于参照发票实际被支付时间（**paidDate**）的历史情况。支付条款 SettlementTerm 中的 **settlementTermId** 字段通常决定了应付款的时间。其他支付条款可能关联发票或者发票中的条款（InvoiceTerm）。

InvoiceContactMech 用于关联发票和联系信息。除了借方和贷方当事人，其他当事人如销售代表或者财务会计人员，可以通过发票当事人（InvoiceParty）关联上一个发票。

发票明细（InvoiceItem）记录用于记录发票的诸如商品，服务，配送，税金，优惠等明细信息。发票栏目使用了类似于订单明细（mantle.order.OrderItem）和退单明细（mantle.order.return.ReturnItem）的数据类型。ItemTypeData.xml 文件定义了这种明细的类型，这里面有很多诸如销售，采购，费用，佣金和工资单等类型。对于销售订单来说，最常用的类型包括产品，登记时间，运费，营业税以及优惠折扣。

就像销售订单明细一样，发票明细也可以通过 **parentInvoiceId** 和 **parentInvoiceItemSeqId** 字段去支持一个层级结构的场景。这适用于类似对某个产品或者服务条目指定税收条目的情况。



发票明细有个描述字段 **description**，并且通常有个产品编号（**productId**），可能还有个资产编号（**assetId**）去关联上产品或者服务的更多详细信息。每条明细记录都有一个数量 **quantity** 以及计量单位（**quantityUomId**），同时有一个单价金额 **amount**。那么，每个发票明细的小计就是：数量×单价（**quantity** * **amount**）。

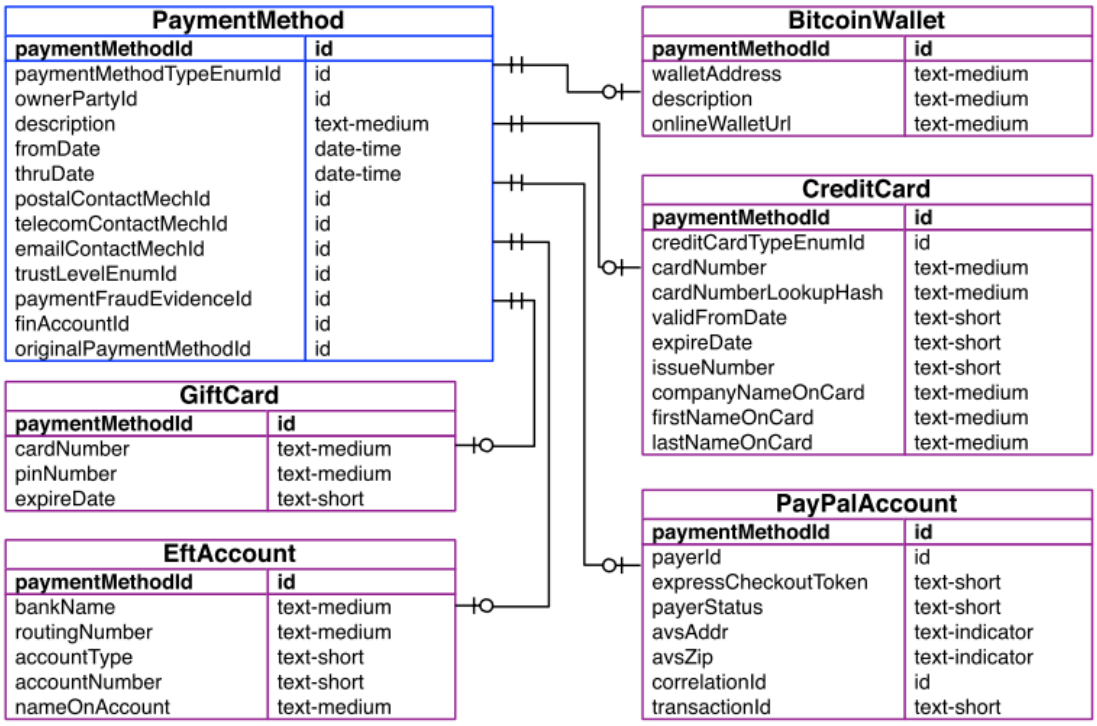
发票明细通过 InvoiceItemAssoc 和其他的明细记录相关联。一个很有用的案例场景是从服务提供商那接到一个消费明细发票，然后可以通过终端机进行支付。

当财务的发票 Invoice 状态转变为出去的票据为最终敲定并且传入的票据为批准时，将会影响并触发总帐过账（GL POSTING），生成财务预制凭证。注意一下，如果一个发票作用于内部组织间的借方和贷方当事人，财务设置将默认对两方都发送票据。如果 **overrideOrgPartyId** 字段数据被填充了，则将使用组织进行发送票据，而不是 **fromPartyId** 或 **toPartyId**，这取决于是否是个内部组织（这种情况适用于借贷双方不是通常的内部组织）。

自动生成发送票据的会计事务（AcctgTrans）将通过一个入口对每个发票明细条目按照其类型发送到总帐会计（GlAccount）中。同时，结算事务入口将按照发票总额发送一个传入的应付款发票和一个传出的应收款发票。

➢ *会计 - 付款方法 (mantle.account.method)*

付款方法（PaymentMethod）是一个用于支付的手段，并对每种支付方式有独立的实体去记录其明细，包括：比特币钱包 BitcoinWallet，信用卡 CreditCard，EFT（Electronic funds transfer，电子资金转帐）账户 EftAccount，礼品券 GiftCard 和贝宝（PayPal）账户 PayPalAccount。如果使用了指定的 **finAccountId** 字段，金融账户 FinancialAccount 也可以使用这些 PaymentMethod。其他的一些支付方式例如现金，支票和汇款单直接用于支付，订单等等无需记录付款方法信息，因为这种支付 Payment 不是通过付款方法进行完成的。



一个付款方法的所有者是某个当事人（**ownerPartyId**），它有一个描述 **description** 并通常有邮政联系方式（**postalContactMechId**），电信联系方式（**telecomContactMechId**）和电子邮件联系方式（**emailContactMechId**）。

付款方法 PaymentMethod 有一个时间有效期（**fromDate**, **thruDate**）。通常 **thruDate** 为空，除非这种付款方式长期不使用或者被改变了。PaymentMethod 和其关联的记录是不可变更的，所以当需要变更原始记录时，需要将 **thruDate** 字段设置上并在修改的明细中添加一条新的记录。新的记录通过 **originalPaymentMethodId** 字段指向关联到原始单据。

付款方法通过信任级别（**trustLevelEnumId**）去进行安全防护，防止欺骗。开箱即用的操作包括了新数据，有效/清理（通过第三方服务），验证（通过外部认证），灰名单和黑名单。如果信任级别设置为灰名单或者是黑名单，则必须通过 **paymentFraudEvidenceId** 字段设置指向一条支付欺诈证据（PaymentFraudEvidence）明细去记录缘由。

礼品券 GiftCard 付款方式通常是通过某个组织进行发放的，这种方式的明细信息通过 GiftCardFulfillment 实体进行追踪记录。

还有某些付款方式，特别是信用卡，通常是通过类似 Authorize.net 和 Cybersource 这样的付款网关自动进行支付。支付过程集成的关联服务有认证，捕获，释放和退款。这些服务通过 PaymentGatewayConfig 进行配置，这些配置使用 ProductStorePaymentGateway 与某个产品商店（ProductStore）进行关联。

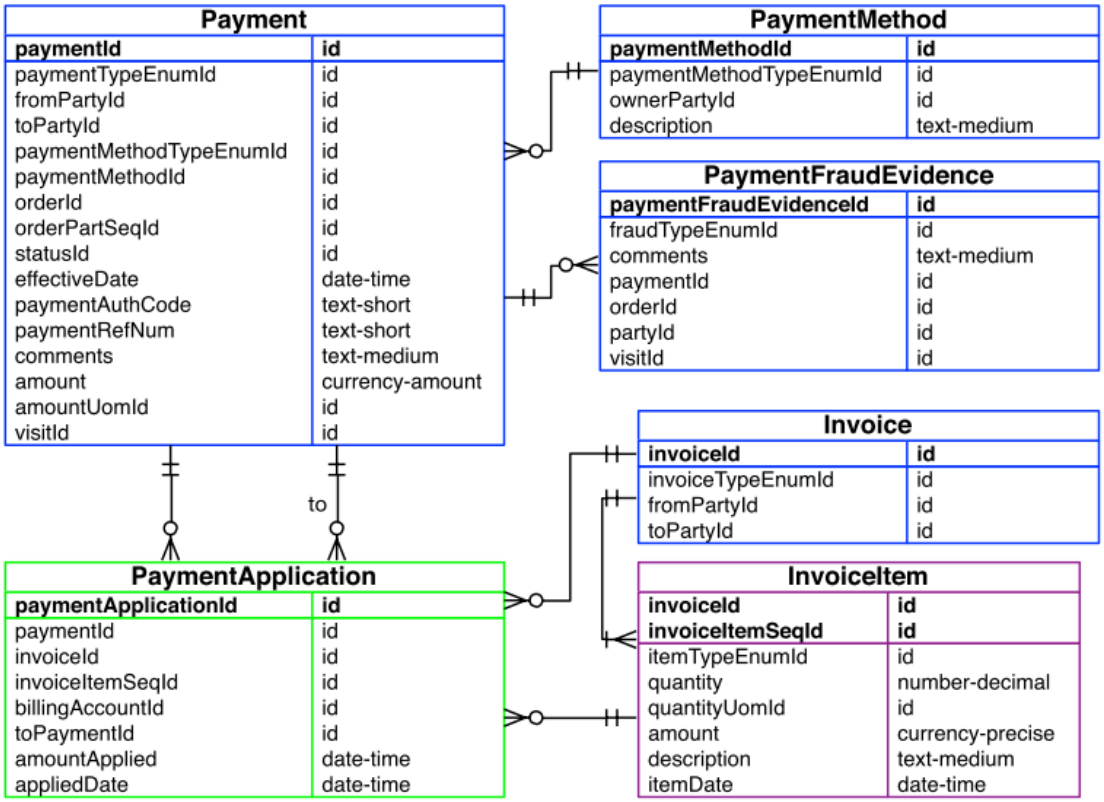
任何时候支付网关的响应明细信息都必须被存储在 PaymentGatewayResponse 实体中。通常这些响应信息有很多的字段去记录支付过程中的响应编码并关联到某个支付（**paymentId**）上。

➢ *会计 - 支付 (mantle.account.payment)*

支付 Payment 通常是响应借方（**fromPartyId**）向贷方（**toPartyId**）提供的发票 Invoice 而进行的操作。支付 Payment 的双方当事人正好与发票的双方相反。支付类型（**paymentTypeEnumId**）包括发票付款（Invoice Payment），支付（Disbursement）和退款（Refund）。支付通常有总额 **amount** 和支付的货币类型（**amountUomId**）。

Payment 通常有一个付款方法类型（**paymentMethodTypeEnumId**）如现金，支票或者信用卡，并且可用的付款方法类型也应该有个付款方法（**paymentMethodId**）关联。

如果支付通过支付网关自动进行支付了，认证的支付网关需被记录在 **paymentGatewayConfigId** 中，这样才可以进行随后的诸如捕获和失效操作。为了方便起见（既然这些支付都是基于支付网关响应的），自动支付中用于记录授权结果的 **paymentAuthCode** 和 **paymentRefNum** 字段以及参照的代码，就可以被用于随后的操作了。当使用信用卡和类似的支付方式通过网关时，还有些明细字段可被使用，包括 **presentFlag**, **swipedFlag**, **processAttempt** 以及 **needsNsfRetry**。



一个支付有很多种状态（**statusId**）包括：建议，约定，批准，交付，确认支付，撤销，失效，拒绝和退款。

支付不像发票一样有很多的明细条目，但有些特殊的情况会有一些扣除额/扣减项目的信息，被记录在 Deduction 实体中。

Payment 记录在订单过程中可以很早就被创建去指定支付的是全部的订单还是订单中的部分产品。对于某个既定的订单表头 OrderHeader 或者订单部分 OrderPart 可能会有很多

条支付记录，这些支付记录使用 **orderId** 关联到订单，并可以用 **orderPartSeqId** 字段去关联订单明细条目。支付（Payment）实体中查看某个订单或者部分的支付明细就使用这些字段了。

支付可以关联到一个金融账户（**finAccountId**）上，并且可以在这个账户上进行授权认证（**finAccountAuthId**）和进行交易（**finAccountTransId**）。

对于欺骗敏感的公司和应用，当处理在线支付交易事务时，使用 Payment 的 **visitId** 字段去关联访问 Visit 是十分重要的。这个字段可以追溯客户端 IP 地址，其他 HTTP 客户端和 session 信息。当一个伪造交易被识别到时，相应的证据应该记录在 PaymentFraudEvidence 实体中，并且这通常用于改变关联的付款方法（PaymentMethod.**trustLevelEnumId**）和联系机制（ContactMech.**trustLevelEnumId**）上的信任级别。

对于企业来说，处理多币种的支付需要转化换算为公司本身使用的币种，或者匹配关联的发票的币种。在这种情况下，原始的总额和币种应当使用 **originalCurrencyAmount** 和 **originalCurrencyUomId** 字段在银行或对账信息中记录。

当状态变为已交货时，财务支付会影响并触发总帐过账。注意，如果借贷双方是企业内部组织，财务设置将会对双方发送支付信息。如果 **overrideOrgPartyId** 字段数据被填充了，则将使用组织进行发送票据，而不是 **fromPartyId** 或 **toPartyId**，这取决于是否是个内部组织（这种情况适用于借贷双方不是通常的内部组织）。

自动生成发送票据的会计事务（AcctgTrans）将通过一个入口发送到总帐会计（GlAccount）中，总帐配置为现金账户的付款方式类型（除非填充 overrideGlAccountId 的内容，否则将默认使用这个类型）。同时，结算事务入口将按照发票总额发送一个传出的应付款发票和一个传入的应收款发票。

PaymentApplication 实体被用于更为复杂的一个发票 Invoice 对应明确支付的场景，它可以支持单次支付对应于多张发票或者一张发票对应多次支付的情况。当借贷双方出现往来支付有一方对另一方取消支付的状况时，一个支付可能会作用于另一个支付。

基于总帐过账的目的，一个没有被关联到发票或者部分被关联总额的 Payment 将会被过账到一个未实施的支付账户中，而不是现金账户。当支付在另一个会计事务中被关联时，将会影响为实施的支付账户和现金账户的核销平衡。

当一个 Payment 是某个预算支出的部分时，它将使用 PaymentBudgetAllocation 关联上一个或者多个预算科目 BudgetItem 记录。

- ***总帐 - 会计(mantle.ledger.account)***

总分类账目（General ledger accounts : GlAccount）构成了内部的组织 Organization 会计科目表。每个科目都有个分类（**glAccountClassEnumId**）去确定科目结算是增加还是减少交易总数并最终形成报表（特别是：一边是资产负债表，另一边是抵减资产，负债以及所有者权益；还有损益表的收入，收入抵销，销售成本，收入和费用科目）。下面是开箱即用的总分类账目的分类结构（这些能够通过不同的 GlAccountClass 类型的 Enumeration 记录去改变）：

1 借方

1.1 资产

1.1.1 流动资产

1.1.1.1 现金和等价物

1.1.1.2 库存资产

1.1.1.3 应收账款

1.1.1.4 预付费用和其他

1.1.2 固定资产

1.1.2.1 土地和建筑

1.1.2.2 设备

1.1.3 其他资产

1.2 费用支出

1.2.1 现金支出

1.2.2 利息费用

1.2.3 销售，日常以及管理费用

1.2.4 非现金费用

1.2.4.1 折旧

1.2.4.2 分期偿还

1.3 销售成本

1.3.1 产品销售成本

1.3.1.1 存货调整

1.3.2 服务销售成本

1.4 收入抵销

1.5 股权配置

1.5.1 资本返还

1.5.2 红利

1.6 未过账

2 贷方

2.1 收益

2.1.1 现金收益

2.1.2 非现金收益

2.2 税收

2.2.1 产品税收

2.2.2 服务税收

2.3 股权/所有者权益

2.3.1 所有者权益

2.3.2 留存收益

2.4 负债

2.4.1 短期负债

2.4.1.1 应付帐款

2.4.1.2 应计费用

2.4.2 长期负债

2.5 抵减资产

2.5.1 累积折旧

2.5.2 累计摊销

3 资源

GlAccount 记录有个类型（**glAccountTypeEnumId**）字段用于自动过账设置。GlAccountType

类型的 Enumeration 记录都是可用的总分类帐目的类型。这里有不少开箱即用的类型定义，例如应收款（AR），应付款（AP），固定资产，短期负债，库存，制成品库存，税金，亏损，销售成本，费用，客户保证金，佣金费用（还有其他很多类型）。虽然有些总分类帐目的类型有重复，但是它们用于不同的字段去标识不同的事情。

总分类帐目使用 **parentGlAccountId** 字段去指定父级账目，用以形成层次结构。每个账目都有一个有别于 **glAccountId** 的编号 **accountCode**，这样编号就可以被改变了，同时账目还有名称（**accountName**）和描述字段（**description**）。账目信息中使用 **postedBalance** 字段来维护每个过账以及与 GlAccount 关联的 AcctgTransEntry 实体记录。

更常见的会计处理会使用典型的总帐 GlAccount 资源类型（**glResourceTypeEnumId**），通常是金钱，也可以是例如原材料，劳动力或者是制成品等类型。这里同样还有个 **glXbrlClassEnumId** 字段来制定报告（XBPL）类型如美国公认会计原则（US GAAP）和 IAP。

为了支持多公司的财务会计，在 GlAccount 记录中有个统一的会计科目表并且每个内部公司 Organization 使用 GlAccountOrganization 实体去维护管理自己的子集账目。过账平衡（**postedBalance**）字段在会计结算时，只对某个公司的信息发生改变。在 GlAccountOrgTimePeriod 中有对每个总帐 GlAccount，Organization 和会计期间（TimePeriod：财政月、一个季度或一年）更为具体的记录描述，同时它有更多的关于 **postedDebits**，**postedCredits**，**beginningBalance** 和 **endingBalance** 的总计信息。他们都是通过总帐服务进行维护的。

其他当事人可以通过 GlAccountParty 实体与总帐进行关联。总帐 GlAccount 可以与预算进行关联，同时预算明细类型使用 GlBudgetXref。

除了固有的总帐层次结构，可能也有两种其他的结构：分类和分组。GlAccountCategory 通过 GlAccountCategoryMember 将总分类帐目彼此之间任意组合并形成一个多对多的关系。这一般用于例如成本中心这种为了追踪和出报告的目的。

GlAccountGroup 是一种更为受限的去组合总帐 GlAccount 记录的方式，一般是为了出报告以及填充税务表单的目的。举个例子，美国国税局 1120 表单（美国企业所得税纳税申报表）就有很多的分组，这些分组有“1a 收入或销售总额”，“1b 退货和折让”以及“4 红利”。每条总帐账目都通过 GlAccountGroupMember 关联至少一种组类型。这也是为了避免只使用总帐账目一次而复制其值。

- ***总帐 - 配置(mantle.ledger.config)***

对一个内部公司 Organization 来说，其财务的偏好信息记录在 PartyAcctgPreference 中。它记录了税收表单类型（**taxFormEnumId**）和产品销售成本方法（COGS method，**cogsMethodEnumId**）的字段，其基于货币会计（**baseCurrencyUomId**）并且字段用于管理发票 ID 流水号(**invoiceSequenceEnumId**, **invoiceIdPrefix**, **invoiceLastNumber**, **invoiceLastRestartDate**, 和 **useInvoiceIdForReturns**)，订单 ID 流水号(**orderSequenceEnumId**, **orderIdPrefix**, **orderLastNumber**)以及默认的退款的支付方法（**refundPaymentMethodId**）。

最为重要的一个字段是 **errorGlJournalId**。它用于自动过账发生问题时，总帐凭证 GlJournal 中放置的交易事务（AcctgTrans）记录。这个凭证的事务应当被定期的复核，并且对于在关闭期间，排除异常问题和交易过账的业务开展前来说，非常重要。最常见的问题就是找不到总帐 GlAccount 配置的某个特定条目（AcctgTransEntry），另一个可能的问题就是借贷不匹配。

包中其他配置总帐 GlAccount 的实体用于处理各种类型的记录去自动过账。通常我们使用 GlAccountTypeDefault 和 GlAccountTypePartyDefault 来配置默认的不同类型的总帐账目，

除非去重新指定。

对于发票 Invoice **过账**来说，各种条目按照其条目类型 ItemType（条目类型和订单条目 OrderItem，退单条目 ReturnItem 和发票条目 InvoiceItem 一样）使用 ItemTypeGlAccount 去自动进行映射。如果发现有一个明确的发票条目 InvoiceItem 即被使用。特定的产品可能会用 ProductGlAccount 或者 ProductCategoryGlAccount，抑或是将 TaxAuthorityGlAccount 用于特定的税务登记 TaxAuthority 的税收条目上。平衡结算通常是发票作为借方对应默认的应收款类型的账目，或是作为贷方对应默认的应付款类型的账目。

对于支付 Payment **过账**来说，PaymentTypeGlAccount 实体用于找寻负载平衡或者对某企业 Organization 进行支付的资产(AR, AP 等)总帐，它一般有 **paymentTypeEnumId**, **isApplied**, 和 **isPayable** 字段（例如应付账款及相对的应收账款）。默认会使用现金账目作为过账的付款方式（PaymentMethodTypeGlAccount），除非指定使用信用卡类型（CreditCardTypeGlAccount）的付款方式或者金融账户类型（FinancialAccountTypeGlAccount）的付款方式。

对于库存过账来说，通常使用确定的 AssetTypeGlAccount 来做总帐，但是对于盘点差异来说，盘点盈亏决定于 VarianceReasonGlAccount 实体配置的差异原因。

- ***总帐 - 对账 (mantle.ledger.reconciliation)***

GlReconciliation 用于记录与外部来源的对账结果，如银行的对账单。每个 GlReconciliation 都和一个总帐 GlAccount 通过 **glAccountId** 字段进行关联，并且通常都是具体某天（**reconciledDate**）和某个具体的企业 Organization（**organizationPartyId**）进行对账。它会追踪期初余额结余（**openingBalance**）以及对账平衡（**reconciledBalance**）。实际的对账记录在 AcctgTransEntry 实体中，同时 GlReconciliationEntry 实体中记录了每条对账的和解金额 **reconciledAmount** 的追溯信息。

- ***总帐 - 事务 (mantle.ledger.transaction)***

会计事务（AcctgTrans）可以被很多事情触发，它一般会关联到触发它的事情或者对触发它的事情添加细节，包括：资产发行（**assetIssuanceId**），资产接收（**assetReceiptId**），盘点库存（**physicalInventoryId**），发票（**invoiceId**），支付（**paymentId**），付款应用（**paymentApplicationId**），金融账户业务（**finAccountTransId**），运输（**shipmentId**）以及工作投入（**workEffortId**）。业务也可以被手动的创建，例如非自动的过账。

会计事务有很多种类型（**acctgTransTypeEnumId**）。常用的类型有销售发票（企业到客户），采购发票（供应商到企业），资产接收，销售库存，到款（收入）以及支付的款项（支出）。还有一些外部的类型包括分期偿还，资本总额，关闭期间和付款通知。

AcctgTrans 发生在一个内部组织 Organization（**organizationPartyId**）中，并且发生在明确的时间点（**transactionDate**）。业务清楚是否交易已经过账（**isPosted**），并且过账的时间（**postedDate**）。事务可能使用 **glJournalId** 字段并是一条分类账，就像企业的错误分类账一样。事务使用的币种可以通过 **amountUomId** 追踪到，并且如果交易的币种不同于业务原始单据（例如订单）的币种，可以通过 **origCurrencyAmountUomId** 找到原始币种。

每个事务的条目（AcctgTransEntry）可能是个借方或者贷方（**debitCreditFlag** 标识为'D'或者'C'），并且有一个总额 **amount** 信息。如果交易的币种不同于业务原始币种将会转化为 **origCurrencyAmount**。每个条目都会关联到一个特殊的总帐类型（**glAccountTypeEnumId**），并且为了过账成功需要关联一个总帐科目（**glAccountId**）。

一条事务条目可能是外部系统会计业务的概要，则 **isSummary** 字段应被设置为 **Y**。对

于发票明细来说，**invoiceId** 记录在 AcctgTrans 中，同时 **invoiceItemSeqId** 保存在 AcctgTransEntry 中。事务条目也可以被产品（productId）和/或资产（assetId）相关联。

日记账（GlJournal）可以用来跟踪特定的会计事务，通常是进行中的错误或者手动类型的事务（**glJournalTypeEnumId**）。它们通常是为了特定的企业（**organizationPartyId**）服务，同时一次性的日记账会一次性被过账并记录是否过账（**isPosted**）和过账日期（**postedDate**）。事务关联日记账通过 AcctgTrans.**glJournalId** 字段。

- ***其他 - 预算 (mantle.other.budget)***

预算 Budget 通常会使用 **timePeriodId** 字段去关联一个会计期间 timePeriodId，并有很多诸如资本和操作的类型（**budgetTypeEnumId**）。每一个预算条目 BudgetItem 都有个总额以及文字描述预算的目的以及缘由。条目类型（**budgetItemTypeEnumId**）通常有必需的或自由支配。当事人通过 BudgetParty 与预算进行关联。

其他与预算条目 BudgetItem 关联的实体对预算进行了更多的细节描述，包括：通过 PaymentBudgetAllocation 进行关联的支付 Payment，EmplPosition，OrderItem，以及通过 RequirementBudgetAllocation 进行关联的需求 Requirement。

当一个预算通过特定的当事人审核之后，结果被记录在 BudgetReview 实体中。BudgetRevision 和 BudgetRevisionImpact 实体用于保持预算的修正历史版本信息。

当进行预算方案时，各种情境都会被讨论和建模。这些场景都可以通过 BudgetScenario 进行记录，并且可以通过 BudgetScenarioApplication 来记录特定的明细条目，用 BudgetScenarioRule 来存储更常用的情景条目类型。

- ***其他 - 税金 (mantle.other.tax)***

税务局（TaxAuthority）是一个征收某个地区（**taxAuthGeoId**）企业税金的政府部门的实体（**taxAuthPartyId**）。对于增值税税务机关来说 **includeTaxInPrice** 字段需要被设置为 Y。如果一条税收 ID 需要免税，则设置 **requireTaxIdForExemption** 字段为 Y。

很多的税务机关对于不同类型的产品征收不同的税率。我们可以为每个产品类型创建一个产品分类（ProductCategory）并使用税收分类（TaxAuthorityCategory）关联它来设置税率。税务机关能够通过 TaxAuthorityAssoc 和其他的税务机关进行关联，用来豁免继承或者作为一个托收人（**assocTypeEnumId**）。例如，某个美国州的税务局征收本州内的城市和县城的税金，当豁免州级别的税务级别时也可能会豁免城市或者县城的税务级别。

当事人通过 TaxAuthorityParty 与税务局（TaxAuthority）相关联。这种表示方式十分有用，在一个内部组织有关系（**isNexus=Y**）或者一个客户税务豁免（**isExempt=Y**）的情况下，税务局会发放一个当事人 ID 并记录在 **partyTaxId** 字段中。

税金可能通过一个外部系统或者通过 TaxGatewayConfig 配置的内部服务去进行计算，在两种情况下都统一指向计算服务（**calculateServiceName**）去计算税金或者调用外部系统。以前使用了一个 TaxAuthorityRate 的实体去专门配置本地的税金计算规则，不过现在已经使用了更灵活的 Drools 的决策表去处理了。税金网关配置（TaxGatewayConfig）使用 ProductStore.**taxGatewayConfigId** 字段与 ProductStore 进行关联。

设施

➢ *设施 (mantle.facility)*

设施是一个建筑物，单元，房间，土地或甚至占地面积。这里还有特殊的设施类型，例如仓库和办公室。设施的主实体是你能想象到的(Facility)并且通过单个主键字段(**facilityId**)进行定义。就像很多的主要(主)实体所带的，设施有类型(**facilityTypeEnumId**)，状态(**statusId**)以及名称（**facilityName**）字段。同样还有字段用于所有者，大小，开放/关闭日期等。

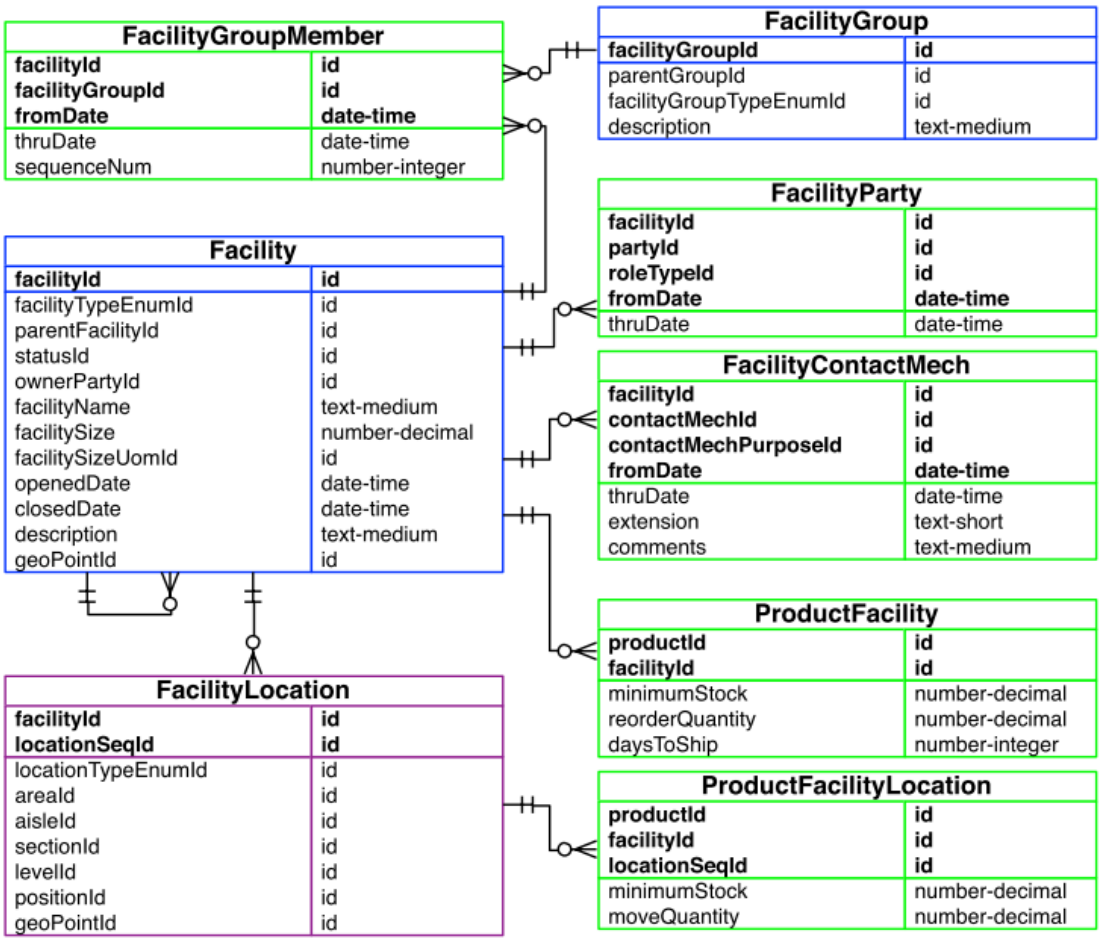
设施是对事物进行层次建模的，就像建筑物中有单元同时单元中有房间。Facility.**parentFacilityId** 字段用于指定每个设施的父亲。理论上说，它可用于仓库库存位置，但为了使结构简化和扁平化，FacilityLocation 实体仅用于一个设施中的库存位置。这些位置有类型(例如大部分的或分拣/主要的)，定位(**areaId**, **aisleId**, **sectionId**, **levelId** 和 **positionId**)字段，甚至还有一个 **geoPointId** 字段用于 GPS 自动导航。

ProductFacilityLocation 实体的记录用于关联一个 Product 和一个 FacilityLocation，表示产品存放的位置，同时设置 **minimumStock** 和 **moveQuantity** 值用于推荐的库存移动（当从大部分零散的位置补货到分拣/主要的位置时）。如果你需要追踪关于特定位置指定产品的更多信息，则可以扩展这个实体。

类似的，ProductFacility 实体用于关联 Product 与 Facility，通过指定 **minimumStock** 和 **reorderQuantity** 字段值，可用于简单的自动（推荐）补货。其他关于特定设施中特定产品信息的关联字段可以按需添加到这个实体上。

设施的实际物理位置可以有两种方式进行记录：通过 Facility.**geoPointId** 字段参照的一条 GeoPoint 记录，或是在 FacilityContactMech 实体中 PostalAddress 类型的 ContactMech 字段中。FacilityContactMech 实体同样可用于设施中更普通的联系信息，包括电话/传真等（通讯）号码，电子邮件及 web 站点，和甚至多个邮政地址（当诸如收到信件，收到货运，装运返回地址等，这些不同的事情都有一个邮政地址时）。

ContactMech 中更多的细节，请参看 ***Contact Mechanism (mantle.party.contact)***部分。



使用 FacilityParty 在指定的角色（**roleTypeId**）和有效时间范围（**fromDate**, **thruDate**）内去关联一个当事人（**partyId**）和一个设施（**facilityId**）。这可用于任何角色，同时可能用于记录在一个特定的设施里谁是所有者，承租者，占有者，管理者，分拣者，包装者等。

使用 FacilityContent 实体去关联资源门面内容（**contentLocation**）和一个设施。它有一个内容类型（**facilityContentTypeEnumId**），例如内部的内容（文档等）和图片，以及任何时候都有用的有效时间范围（**fromDate**, **thruDate**）。

使用 FacilityGroup 实体是为定价和管理的目的去组织设施，或更通常的是去保持更好的追踪大量的设施。设施组有一个 **description**，是层级结构（**parentGroupId**）的，并有一个诸如管理架构或定价组的类型（**facilityGroupTypeEnumId**）。FacilityGroupMember 实体用于关联 Facility 和 FacilityGroup。你同样也可以使用 FacilityGroupParty 去关联一个特定角色的当事人和一个设施组。

人力资源

- ***能力 (mantle.humanres.ability)***

能力最通常的展现形式是当事人的履历 PartyResume，它可能在 **resumeText** 字段中有完整的履历文本或可能指向到一个资源门面的内容 **contentLocation**。

从 PartyQualification 实体的结构中可获得更多的诸如学历，资质证书及工作经验等信息。可用的类型（**qualificationTypeEnumId**）是 Enumeration 实体中为 QualificationType 类型的记

录，同时你可以在这里按需要进行添加。它有一个 **verificationStatusId** 字段用于状态的跟踪验证，还有一个更普通的状态（**statusId**）字段，可以为已完成，未完成，或延期的，专职的，兼职的，或有工作经验的承包人。

PartySkill 实体相对于更普通的资历，主要用于更具体的技能。它应该包括诸如特定的编程语言及类库，设备操作，甚至是创新人才。技能类型（**skillTypeEnumId**）是 SkillType 类型的 Enumeration 实体记录。关于技能的字段例如有 **yearsExperience**，**skillLevel** 和 **startedUsingDate**。

绩效评估 PerformanceReview 是用在经理（**managerPartyId**）和某个特定职位（**emplPositionId**）上的雇员（**employeePartyId**）之间的。其中的条目（PerformanceReviewItem）有各种类型（**reviewItemTypeEnumId**），例如职责（Responsibility），态度（Attitude），和带有评定等级（**reviewRatingEnumId**）的工作满意度（Job Satisfaction），还有对每个条目都有的评论 **comments**。除了评估的环境，这里也许还有绩效说明被记录在 PerformanceNote 实体中。

TrainingClass 实体用于追踪雇主赞助的和其他培训的可用课程，同时 PersonTraining 实体用于批准和/或实际已完成的课程。

- ***雇佣（mantle.humanres.employment）***

雇佣 Employment 实体用于追踪一个雇员（**employeePartyId**）在某个时间范围内（**fromDate**, **thruDate**）受雇于一个雇主（**employerPartyId**）的某个明确职位（**emplPositionId**）的雇佣信息。当雇佣关系终止时，它将会追踪记录一个终止原因（**terminationReasonEnumId**）和终止类型（**terminationTypeEnumId**）。

BenefitType 福利类型的福利通过 EmploymentBenefit 实体进行追踪，其通过 EmploymentPayGrade 关联了 PayGrade，并使用 PayrollPreference 关联了薪酬偏好。

在雇佣之前，这里也许会有一个申请人（**applyingPartyId**）为某个职位（**emplPositionId**）提交了一个就职申请（EmploymentApplication），并可选的关联了一个工作要求 JobRequisition（**jobRequisitionId**）。

雇佣结束之后，任何的失业索赔都可以通过 UnemploymentClaim 进行追踪。

- ***职位（mantle.humanres.position）***

EmplPosition 是在一个组织 organization（**employerOrganizationPartyId**）中，对个人 Person（**filledByPartyId**）的具体职位。使用 EmplPositionParty 实体去将其他的诸如经理或部门当事人关联到职位上。EmplPosition 实体有一个工资等级（**payGradeId**），可能是预算（**budgetId**, **budgetItemSeqId**）的一部分，并被计划在某个时间范围（**estimatedFromDate**, **estimatedThruDate**）内。

职位关联了雇佣职位种类（**emplPositionClassId** 指向到 EmplPositionClass），例如程序员，商业分析员，项目经理等等。通常一个种类中有很多的职位，而且一个种类可以单独的存在并按一个简化的模型直接与当事人（EmplPositionClassParty）进行关联去决定等级，同时这种情况下无需 EmplPosition 记录。

诸如财务管理，库存管理和采购管理的职责可能使用 EmplPositionResponsibility 实体与职位进行关联，或使用 EmplClassResponsibility 实体与职位种类进行关联。这里有一些开箱即用已定义好的职责，同时可以使用 EmploymentResponsibility 类型的 Enumeration 记录来定义额外的职责。

- ***费率 (mantle.humanres.rate)***

在一个企业中，制定工资标准等级通常十分的有用。有效的工资等级使用 PayGrade 实体，同时 PayGradeSalary 实体记录了一个时间范围内（**fromDate**, **thruDate**）实际的支付金额 **amount**。

使用 RateAmount 实体用于更多的细节和组织费率信息。这可用于服务执行去给客户结算利率，以及如果可用的话，实际的执行服务去支付给外部的供应商（**ratePurposeEnumId**）。费率类型（**rateTypeEnumId**）包括了标准的，打折的，延期的以及现场工作。

rateAmount（结合币种 **rateCurrencyUomId** 和时间单位 **timePeriodUomId**）是一个有效时间范围内（**fromDate**, **thruDate**），并限制为某个特定的当事人（**partyId**），工作投入（**workEffortId**）或职位种类（**emplPositionClassId**）。

- ***招聘 (mantle.humanres.recruitment)***

招聘过程通常是始于新建了一个 JobRequisition 以及一个或多个 EmplPosition 记录的人员征用。典型的就职申请 EmploymentApplication 记录随后应用到职位上，然后对于某些申请需要 0 到多次的工作面试 JobInterview 记录，每个面试的完成都有一个面试官（**interviewerPartyId**）面试候选人（**candidatePartyId**）的信息。对于每个职位来说，当一个候选人被雇佣时，新建一条雇佣 Employment 记录。

市场

- ***市场活动(mantle.marketing.campaign)***

市场活动 MarketingCampaign 一般用于追踪市场推广工作，并也许用于系统中追踪的工作力度，或还可能用于组织其他事情，例如通讯录，跟踪代码以及销售机会。

活动中有各种预算/花费字段包括 **budgetedCost**, **actualCost** 以及 **estimatedCost**。它在一个可选的时间范围（**fromDate**, **thruDate**）内有效。对于活动的结果有这些字段如 **convertedLeads**, **expectedResponsePercent** 以及 **expectedRevenue**。

活动中可能有各种当事人，像是营销商，销售代表，管理者，潜在客户以及领导者，并使用 MarketingCampaignParty 实体与活动关联。MarketingCampaignNote 实体用于追踪活动的注意事项，它是对于活动本身的 **campaignName** 和 **campaignSummary** 字段信息的补充。

- ***沟通 (mantle.marketing.contact)***

通讯录 ContactList 用于计划和追踪众多对外的沟通信息，例如营销，新闻通讯以及通告（**contactListTypeEnumId**）。它可以通过电子邮件，电话，邮政信件或其他任何形式的沟通（**contactMechTypeEnumId**）。它可能会关联一个市场活动 MarketingCampaign（**marketingCampaignId**）。

通讯录一般由一个具体的当事人 Party（**ownerPartyId**）所有/管理。其他的当事人可以使用 ContactListParty 去关联它。通讯录主要用于接收对外沟通的当事人以及可选的他们应如何进行沟通（**preferredContactMechId**）。大部分的邮件列表是选择性加入的（opt-in），而

且通常是一个对外的电子邮件去验证地址后才可以完成，同时它有一个代码会通过 **optInVerifyCode** 字段被核实追踪。

ContactListParty 实体有一个状态（**statusId**）字段，可能是等待接受，已接受，拒绝，在用，无效，取消等待，或未经同意的。

使用 ContactListEmail 实体去配置列表中对外的电子邮件，其中包含了诸如订阅通知，退订验证，取消订阅通知及发送的邮件等类型（**emailTypeEnumId**）。它指向一条 Moqui 的 EmailTemplate 记录（**emailTemplateId**）并使用 org.moqui.impl.EmailServices.**send#EmailTemplate** 服务。

使用 CommunicationEvent 记录去追踪实际的沟通，并通过 ContactListCommStatus 去关联通讯录。尽管深入的细节在 CommunicationEvent 记录上，但使用这种方式可以追踪到当事人 Party（**partyId**）和实际使用的沟通机制 ContactMech（**contactMechId**）。查看 ***Communication Event (mantle.party.communication)***部分获得额外的细节。

- ***市场划分 (mantle.marketing.segment)***

市场划分 MarketSegment 及其关联实体用于定义某一组（划分）当事人 Party 记录，使用细分市场分类 MarketSegmentClassification 实体去指定当事人的类别 PartyClassification，细分区域市场 MarketSegmentGeo 用于 Geo（地理边界），同时 MarketSegmentParty 用于 Organization 企业当事人，指的是企业内所有的当事人。

一个划分可用于多个目的，如基于当前系统中匹配市场划分条件的所有当事人记录，去填充 ContactListParty 记录，或者使用 MarketInterest 实体去记录感兴趣的在某个产品分类 ProductCategory 中的产品集合。

- ***跟踪 (mantle.marketing.tracking)***

跟踪代码 TrackingCode 可以用于内部路径追踪，关键的 web 页面，AB 或其他多元的测试。它同样可用于追踪源自子公司特殊订单的链接去支付子公司的佣金。

一旦一个跟踪代码出现在系统中，它可以使用 TrackingCodeVisit 实体去和一个 Moqui 的 web 访问 Visit 进行关联，TrackingCodeOrder 实体用于关联一个订单（为了转换跟踪和子公司的佣金使用），同时 TrackingCodeOrderReturn 实体用于关联退货单。

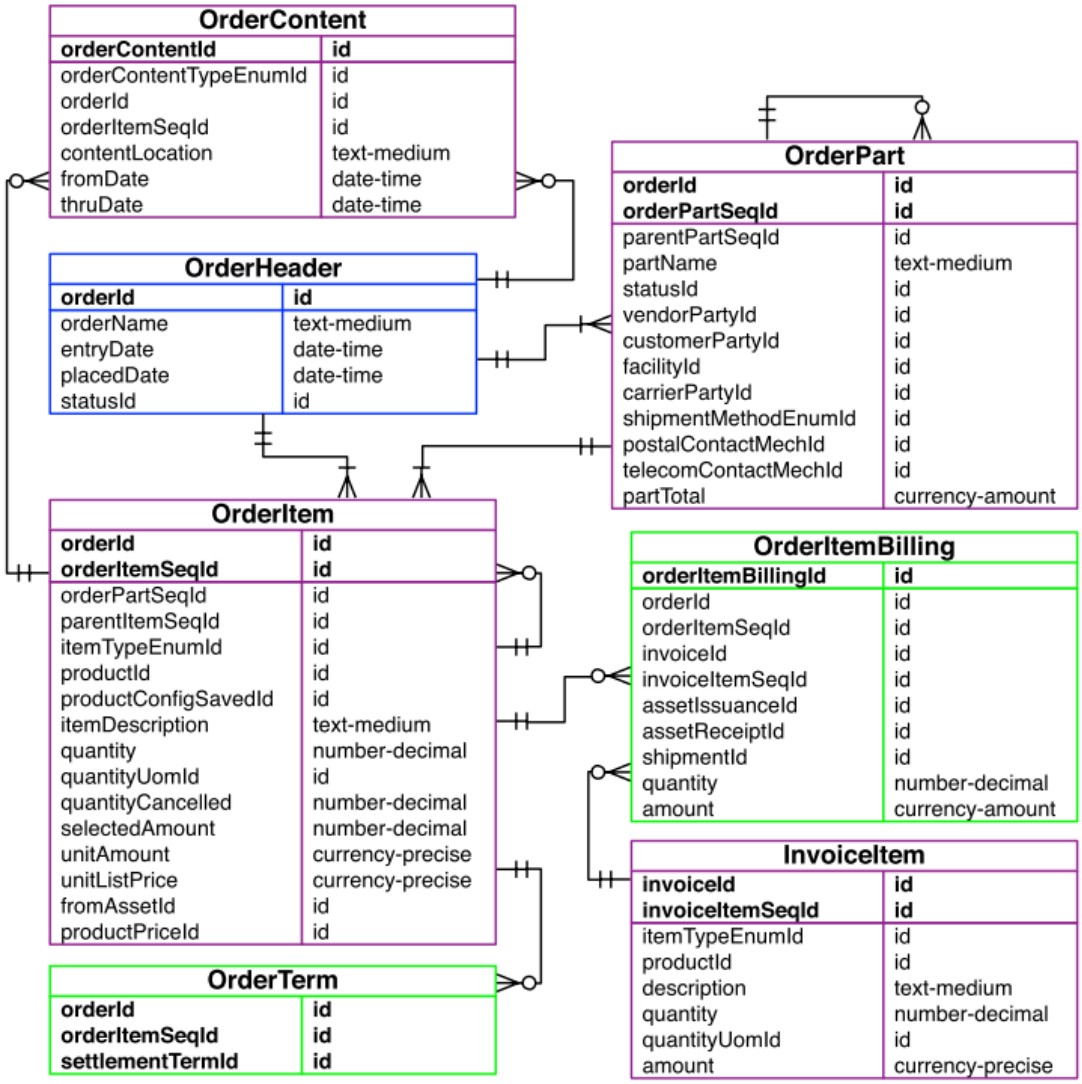
对于子公司的佣金来说，遵循浏览器 cookie 的保护原则，跟踪代码通常放到一个 cookie 中，然后当一个订单被订购（相对于通过更多的手段去记住）时，再从 cookie 中拉取出来。这与跟踪代码去关联一个访问是不同的，通常所有的跟踪代码用在一个访问期间，而订单过后可以通过 OrderHeader 上的 **visitId** 字段去绑定它们。

订单

- ***订单 (mantle.order)***

订单头（OrderHeader）是订单的主实体。一个订单可以是一个采购单或者是销售单，并且事实上由于每个子订单（OrderPart）都有客户（**customerPartyId**）和供应商（**vendorPartyId**），所以子订单从结构上可支持多当事人的单据。子订单可以被拆分成多个，用于例如发货到不

同的地方或者采用不同的付款方式，从不同的地点发货等等。子订单可以通过 OrderPartParty 与其他的当事人进行关联。订单拆单成子订单同样可用于不同的收货地址，不同的运输方式，不同的交割日等。



收货地址（一种联系途径）通过 OrderPart.**postalContactMechId** 字段进行参照，同时一个可选的字段 **telecomContactMechId** 用于指定一个手机号码。订单还可以通过 OrderContactMech 实体关联扩展的联系方式，例如付费电话，收货地址等。

订单有很多种状态，其可以是购物车，报价单（供应商提供），或者是到货单（客户接收到的）。还有很多种其他的状态，比如意愿清单，礼物登记以及自动再订货（一个始终开启的自动循环下单机制，其中每个订单都是独立的）。

订单下完之后就等待履行了，并且它最终可以被客户确认完成/取消或者被供应商拒绝。订单可以被保留或者临时的打上一个正在变化中的状态，去避免产品运费或者税金的重新自动计算。订单头（OrderHeader）和子订单（OrderPart）都有个状态（**statusId**）字段去独立的追踪状态变化。订单条目没有状态 **statusId** 字段，它们的状态决定于条目的数量和是否充足的数量等。

订单的明细被记录为订单条目（OrderItem）信息。为简化起见，每个订单条目 OrderItem 都只关联一个子订单 OrderPart 记录。订单条目 OrderItem 记录是有层级的，所以它们可用于调整和对父级条目进行详细描述。这种方式十分有用，比如营业税和折扣就可以使用一条

条目来记录了。同样，对于高复杂度的，条目彼此组织关联的订单也十分有益，比如特定的建筑材料用于构建其结构的不同部分的阶段。

订单条目和其他条目的类型类似，包括 mantle.account.invoice.InvoiceItem 和 mantle.order.return.ReturnItem。它们共享的条目类型信息被定义在 ItemTypeData.xml 文件中。这里有很多种类型，例如销售，采购，费用，佣金和工资单。对于销售订单来说，常用的类型有产品，时间周期，运费，营业税以及优惠折扣。

当订单条目 OrderItem 被计费（被发票记录）时，使用 OrderItemBilling 实体去关联发票条目（InvoiceItem）。通常当装运 Shipment 被发送（实际上通常是打包完成）或者接收时，实物的计费也将完成，所以在 OrderItemBilling 实体中有个 **shipmentId** 字段。对于外发的订单来说，发布的库存清单作为一条 AssetIssuance 记录被创建，同时我们用 **assetIssuanceId** 字段去指向这条记录。同样的我们用 **assetReceiptId** 字段指向一条传入的运输，记录在 AssetReceipt 中。

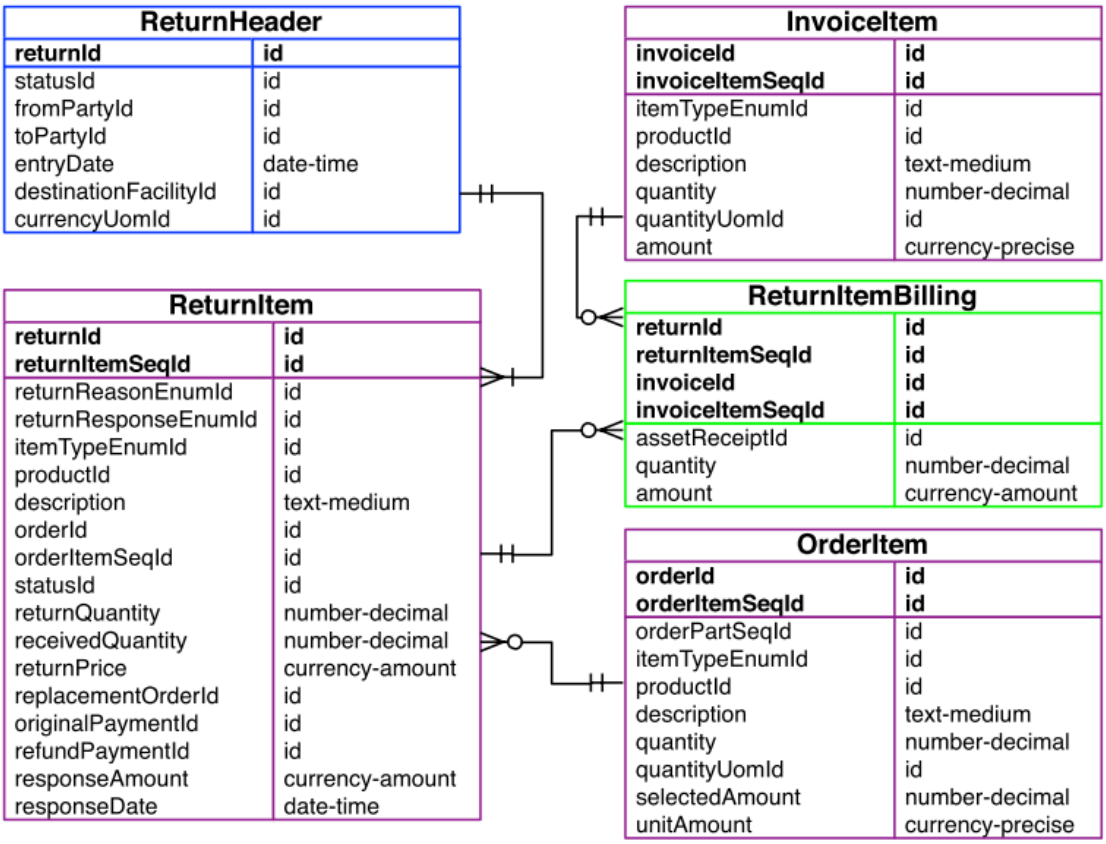
当一个订单条目 OrderItem 关联任务，项目或者其他类型的工作投入 WorkEffort（通常是工作/服务的订单），它使用 OrderItemWorkEffort 实体去关联它们。

订单还有一些其他的记录关联它，包括通讯事件（OrderCommunicationEvent），内容（OrderContent，文档或图片），说明（OrderNote）以及条款（OrderTerm）的支付。

- ***退货单 (mantle.order.return)***

退货单头（ReturnHeader）跟踪了来源于客户（**fromPartyId**）对供应商（**toPartyId**）的请求和执行的退回订单条目的明细信息。注意，组织可能是任意其中的一种当事人 Party，或者换句话说，退单可能是进（从客户那接收一个退单）或者出（向供应商发送退单）。

每个退单条目（ReturnItem）记录都指向了一条订单条目（OrderItem），并指定了退货数量（**returnQuantity**）。这里还有个独立的字段，接收数量（**receivedQuantity**），用于追踪实际返还的条目数量。每个退单条目 ReturnItem 类似订单条目 OrderItem 都有个 **itemTypeEnumId** 字段，这样每个类型的条目都可以被打上“退回”的标记（包括产品，税金，运费，折扣等），以达成退款的目的等。



每个退单条目 ReturnItem 都有个 **returnReasonEnumId** 字段（例如不想要，次品，送货异常等）去追踪退货的原因。退单条目通过 **returnResponseEnumId** 字段去指定企业响应退货的条目（例如退款，退料单，各种替换方式等）。还有其他的一些字段去追溯关联的信息（**replacementOrderId**, **refundPaymentId**, **billingAccountId**, **finAccountTransId**）。对于退款来说，一般会有个发票基于返还的财务追踪（结果将会记录到总帐中等），ReturnItemBilling 用于关联每个退单条目 ReturnItem 和发票条目 InvoiceItem。

退单头（ReturnHeader）和退单条目（ReturnItem）都有个状态字段（**statusId**）去追踪每个条目的执行过程，主记录步骤状态的完成标识退单完成。开箱即用的状态包括：创建，请求，批准，运输，接收，完成，人工响应需求以及撤销。

当事人

➢ *当事人（mantle.party）*

在这里当事人（party）这个术语的含义类似于法律用语中的诉讼当事人，作为一个个人或者组织，而不是娱乐意义的聚会。这里有两种类型的当事人并且每种有自己的可用的明细实体去扩展当事人 Party 实体：Person 实体代表个人；Organization 实体表示一个团体，其中的成员可能是个人也可能是组织。这些实体都有和当事人 Party 实体相同的主键字段 **partyId**，这样它们之间就有个一对一的关系。

当事人 Party 的名称按照其类型来源于不同的字段。对于组织来说，Organization.**organizationName** 字段是其名称。对于个人来说，在 Person 实体中有多个字段组成了名称：称谓（**salutation**），**firstName**, **middleName**, **lastName**, 头衔（**personalTitle**），

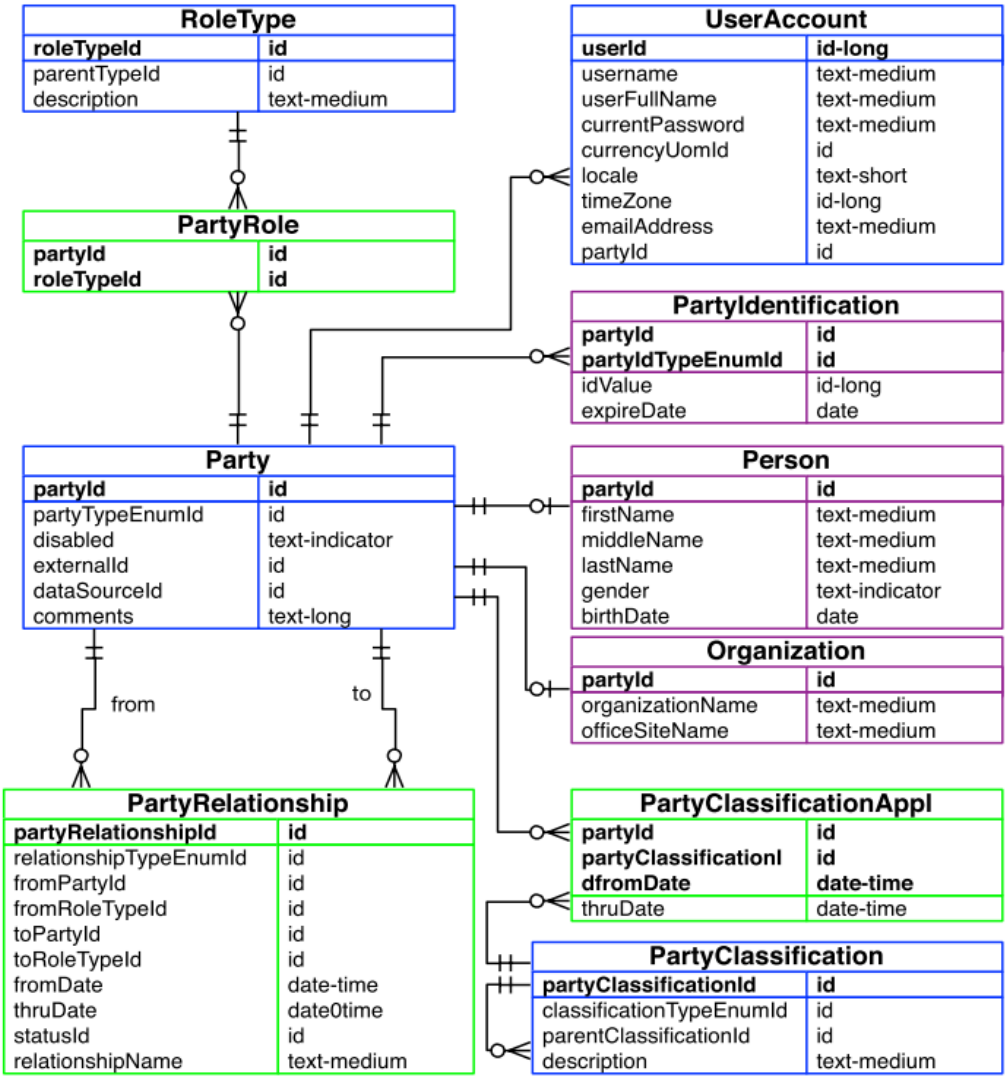
后缀（**suffix**），以及昵称（**nickname**）。通常至少使用姓和名字（first and last names），其他的较少用到。还有很多其他的字段在 Party, Person 和 Organization 实体中，用于指定当事人信息的细节，就像其他任意的实体一样你可以扩展字段去添加你需要的信息。

每个当事人都有零到任意个角色，用于定义当事人如何与系统中其他的数据结构如订单，工作投入（任务等），协议，甚至是其他当事人发生关联。角色类型（RoleType）实体用于定义可用的角色，并且在 Mantle 中定义了相当多全面的、开箱即用的类型。一些例子角色包括：承运人，收票的客户，发货的供应商，雇员，子公司和配偶。

角色和当事人通过当事人角色（PartyRole）实体进行关联。其他的实体特意的设置了 **partyId** 和 **roleTypeId** 两个外键字段去关联 Party 和 RoleType 实体，而不是 PartyRole，这样的话 PartyRole 的记录就变为可选的（非强依赖了）。在有些场景下，当事人具有某个确定的角色十分有用时，你就要使用 PartyRole 实体了。

PartyRelationship 实体记录了当事人之间的关系。它们包括：团体的成员，企业/组织的雇员，企业层级结构，联系人，朋友等。**relationshipTypeEnumId** 字段对应了很多开箱即用的关系类型，并且你可以通过 **enumTypeId**=PartyRelationshipType 的方式在 Enumeration 实体记录中添加更多的枚举类型记录。为了更细致的描述自然界的关系，这里添加了来源和目标当事人以及角色类型字段。如果有需要，这里还有有效时间（**fromDate**, **thruDate**）字段和状态 **statusId** 字段（和大部分状态字段一样可以设置 **enable-audit-log**=true 去记录状态变更的历史）。

当事人有很多的标识，例如有司机牌照号码，员工编号以及相关的外部系统标识，这些都记录在 PartyIdentification 实体中。这个实体有 **idValue**，用于标识 ID 类型的 **partyIdTypeEnumId** 以及一个可选的标识到期时间的 **expireDate** 字段。



当事人分类用于将当事人按照行业/SIC/NAICS，规模，税收，少数派/EEOC 等进行分门别类。每个分类 PartyClassification（例如 NAICS 行业分类 541511-自定义计算机编程服务）都有个 **classificationTypeEnumId**（例如 PctNaicsCode）和一个可选的 **parentClassificationId** 字段去组织。PartyClassificationAppl 实体用于在一个时间区间（**fromDate**, **thruDate**）内去关联当事人和分类。

当事人可以通过 PartyContent 关联资源门面内容，PartyGeoPoint 关联地理位置，PartyNote 关联说明信息。当事人也可以通过 UserAccount.**partyId** 字段关联 Moqui 框架的用户帐户 UserAccount 实体，在 Mantle 中是使用扩展实体（extend-entity）方式添加的。

- ***协议 (mantle.party.agreement)***

协议 Agreement 适用于追踪销售，人力资源，委托以及其他类型(**agreementTypeEnumId**)的协议。协议特指两个当事人之间（**fromPartyId**, **toPartyId**）并且两者具有特定的角色（**fromRoleTypeId**, **toRoleTypeId**）。AgreementParty 实体用于关联额外的当事人。AgreementItemParty 实体用于当事人关联协议中特定的条目。

协议创建于某个具体的日期（**agreementDate**）并在一个时间段（**fromDate**, **thruDate**）内有效。你可以在 **textData** 字段中记录协议文本的内容描述 **description**。

AgreementItem 实体用于记录协议的细节描述并有更多的结构。一个条目有自己的详细文本（**itemText**）和自己的有效时间段（**fromDate**, **thruDate**）。条目有特定的类型（**agreementItemTypeEnumId**），例如子协议，定价，协议片段或者佣金比率。**currencyUomId** 字段用于关联追踪条目的币种信息。

如果协议变化了，则需使用 AgreementAddendum 实体去追踪变更的附件/附录，可能是某个特定的条目或者是整个协议中应用附件。

条目或者整个协议中条款记录在 AgreementTerm 实体中，例如支付，酬金，罚款，动机，解约，赔偿，佣金以及采购条款。这里有很多的类型定义，并且你可以定义添加各种其他的 TermType 条款类型的 Enumeration 枚举记录。这些类型通过结算条款（SettlementTerm）实体也同样适用于计费账户（BillingAccount）以及发票 Invoice。

AgreementItemGeo 实体用于关联一个协议条目和一个特定的地理边界（Geo），AgreementItemWorkEffort 实体关联协议和一个例如项目的工作投入 WorkEffort。当协议（条目）用于人力资源时，AgreementItemEmployment 实体关联雇用 Employment 信息和协议。

对于产品定价条目来说，产品价格（ProductPrice）有一个 **agreementId** 和一个指向协议条目 AgreementItem 的 **agreementItemSeqId** 字段。这里提供了结构化的定价细节能够用于特定订单的自动计算价格。

- ***沟通事件 (mantle.party.communication)***

CommunicationEvent 实体用于持续跟踪任意的角色（**fromRoleTypeId**, **toRoleTypeId**），当事人之间的沟通情况（**fromPartyId**, **toPartyId**），并可以通过任意的沟通机制（**fromContactMechId**, **toContactMechId**；参见下个部分的 ContactMech 明细）。甚至于没有联系机制与沟通事件关联，例如手机/电话号码，电子邮件等的联系机制类型（**contactMechTypeEnumId**）字段也可以达到类似的目的。

其他沟通双方之外的独立角色和联系机制的当事人可以通过 CommunicationEventParty 实体与沟通事件 CommunicationEvent 产生关联。这种对于会议和电话会议事件来说特别的有用。

沟通事件类型在 CommunicationEvent 实体中的 **communicationEventTypeId** 字段进行指定，它指向了一条沟通事件类型（CommunicationEventType）的记录。这些类型与联系机制类型相关联。举个例子，手机沟通事件类型使用 CommunicationEventType 实体上的 **contactMechTypeEnumId** 属性字段与电话号码联系机制相关联。

沟通事件 CommunicationEvent 对于进和出的事件都有个状态（**statusId**）字段，状态包括进行中，准备就绪，发送，接收，已阅，已解决，参考，已回访，未知当事人以及撤销。这些状态应该能处理大部分的场景包括呼入需要待查看和处理（解决）的邮件队列。对于状态历史这个字段使用了实体门面（Entity Facade）的审计日志。特定事件的时间通过 **entryDate**, **datetimeStarted**, 和 **datetimeEnded** 三个字段进行追踪。

沟通事件是层级结构的，使用 **parentCommEventId** 字段去追踪之前的或者父级沟通事件用来处理线性的讨论，同时 **rootCommEventId** 字段将这个线性讨论的源头和其他所有的讨论串联起来。

沟通事件的内容可以用标题（**subject**），内容类型（**contentType**：MIME type）和内容体（**body**）字段使用任何说明（**note**）字段中的说明文字来记录。对于邮件消息记录来说，**Message-ID** 信头标记了电子邮件可以用 **emailMessageId** 字段来记录。其他的内容可以使用资源门面本地存储（Resource Facade location）来保存并关联沟通事件记录于 CommunicationEventContent 实体中。

其他诸如个性化服务和销售询价的目的，沟通事件能通过 CommunicationEventPurpose 实体进行追踪（用于区分不同目的的实体与沟通事件进行关联）。沟通目的通过 **purposeEnumId** 字段进行指定，它指向了一条 CommunicationPurpose 类型的 Enumeration 枚举记录，这样可以使用枚举值并添加更多可用的目的。

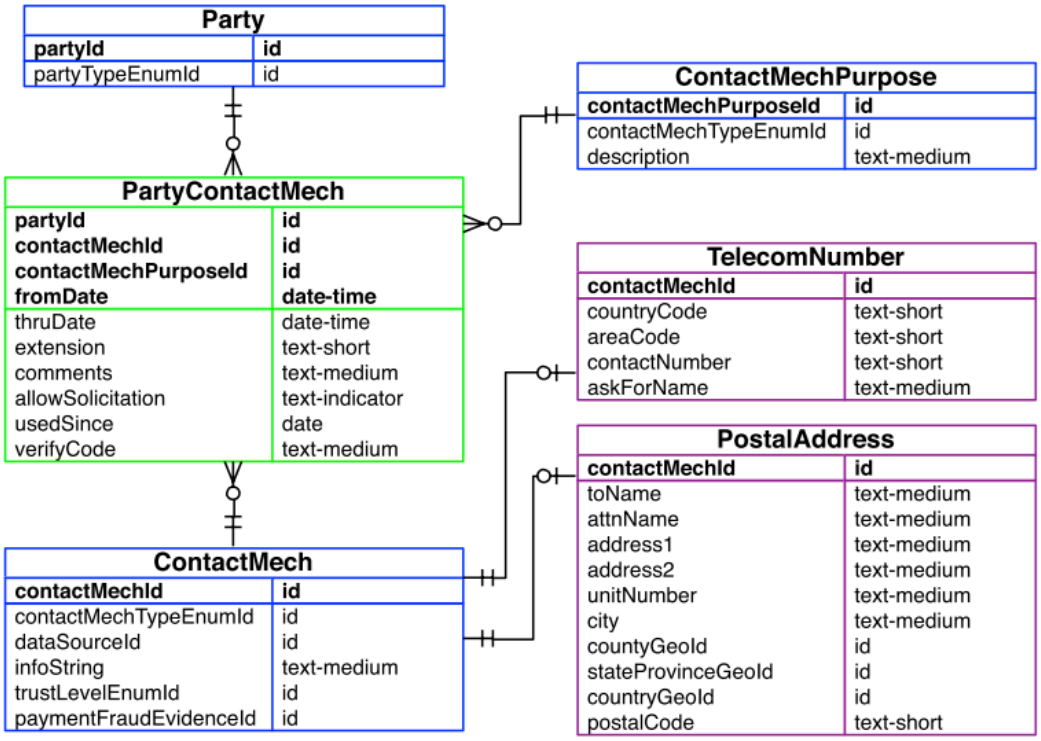
当关联产品时可以使用 CommunicationEventProduct 实体与沟通事件进行关联。

- ***联系机制 (mantle.party.contact)***

联系机制是联系当事人的一种手段。主实体是 ContactMech 并且同时有很多的类型字段，但只有其中的两个有其他的实体扩展的信息字段：通讯地址（PostalAddress）和电话号码（TelecomNumber）。剩下来的类型（例如邮件地址）使用 ContactMech.**infoString** 字段。

ContactMech 实体的主键字段为 **contactMechId**。类似于 Party，Person 和 Organization 的数据库模式，ContactMech，PostalAddress，和 TelecomNumber 实体使用相同的主键这样它们之间就有个一对一的关联关系。

PartyContactMech 实体用于关联联系机制 ContactMech 和当事人 Party 信息。**contactMechPurposeId** 字段用于描述当事人 Party 的联系机制 ContactMech 中，类似发货目的地或者计费电话号码等用途。ContactMechPurpose 实体中定义了目的的记录，这里有比较全的开箱即用的类型并且你可以添加定义更多。



PartyContactMech 实体有一个有效时间（**fromDate**, **thruDate**）字段去定义当事人的联系机制的时间段。Party. ContactMech 记录中的联系机制 ContactMech 是不变的（他们应该永不被改变），所以它们可以被用到其他地方而无需改变以致影响到其他地方（同时保持一个联系信息的历史）。当一个联系需要变化时，创建一个新的关联当事人 Party 的联系机制记录，同时对老的当事人联系机制 PartyContactMech 记录设置截止时间 **thruDate** 的数值去终止使用它。参见 mantle.party.ContactServices.**update#PartyContactOther** 服务中更多的细节点关于如何实现的(对于通讯地址和电话号码中附加字段和独立的实体也是类似的服务处理方

式)。

联系机制 ContactMech 应该有一个 **trustLevelEnumId** 信任级别集合去防止伪造欺骗。开箱即用的操作包括：新数据，有效/清除（通过第三方服务），验证（使用外部关联或者认证），灰名单以及黑名单。如果信任级别是灰名单或者黑名单时，则必须有一个 **paymentFraudEvidenceId** 字段去指定 PaymentFraudEvidence 记录缘由的明细。

另一个类似当事人 Party 的联系机制 ContactMech 的使用方式的是 mantle.facility.Facility。设施类似于当事人一样，有一个长久的，多种联系机制的联系信息记录并永不改变。就像当事人 Party 一样有服务（参见 mantle.facility.ContactServices 服务）去更新设施的联系机制，终止老的创建新的。

有很多实体关联联系机制，并且其中的一些为了不同的目的使用关联实体去关联这些联系机制。它们包括 InvoiceContactMech，OrderContactMech，ReturnContactMech，ShipmentContactMech，以及 WorkEffortContactMech。这些实体没有有效时间（**fromDate**, **thruDate**）字段，因为它们是短生命周期的，并且如果联系信息变化，只需改变 **contactMechId** 指向到另外一条记录。

- ***时间周期 (mantle.party.time)***

时间周期 TimePeriod 实体是用于自定义的时间周期，相对于类似财务年度/季度/月和销售季度（**timePeriodTypeId**，参见 TimePeriodType 实体）这种日历时间周期而言。它们可能会和日历时间周期相匹配，例如开始时间（**fromDate**）是一个日历周期的开始并且终止时间（**thruDate**）是实际的日历周期截止点，但是对于无论任何功能来说，时间周期都比日期时间具有更多其他的用途。

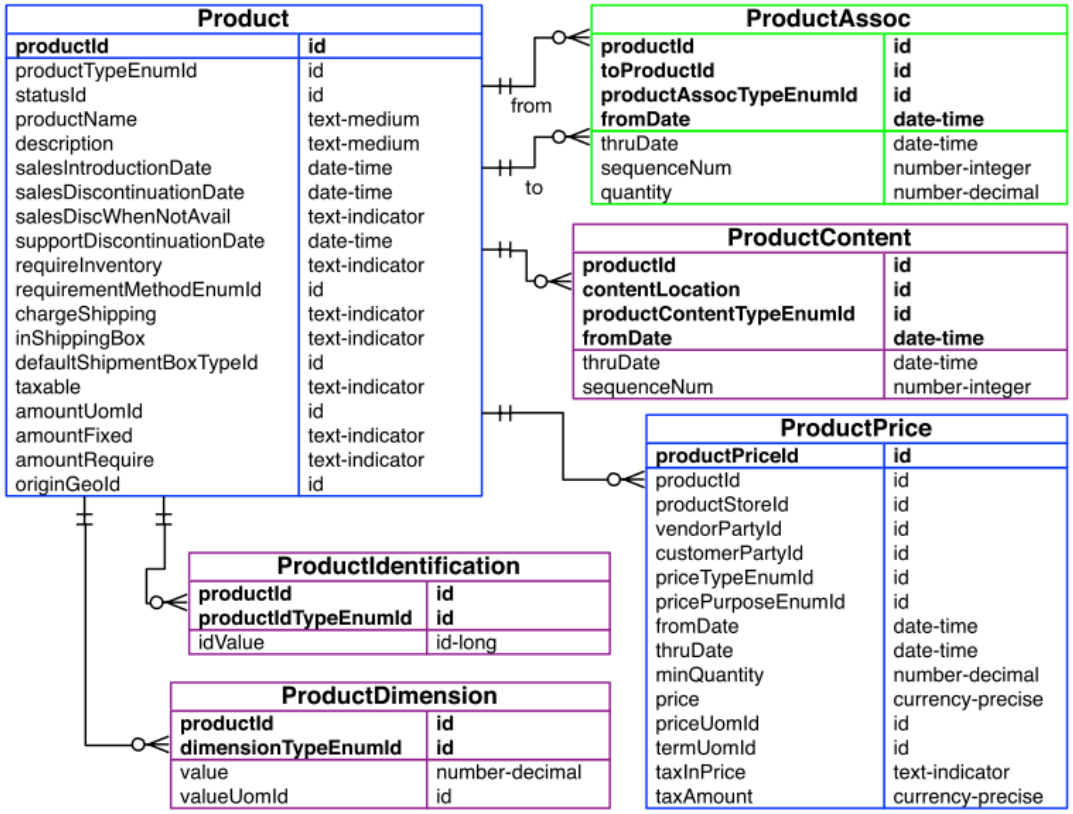
时间周期可以关联其父周期（**parentPeriodId**）和前置周期（**previousPeriodId**）。它同样可以与当事人 Party（**partyId**）进行关联，类似财务周期一样为了企业的财务会计的目的去指定。当时间周期 TimePeriod 用于总账时，**isClosed** 字段指定了周期的关闭时间并且这个时间后业务不允许再进行过账了。

产品

- ***定义 - 产品 (mantle.product)***

产品可以是人，地点或者事物。实际上，这就是一个名词，产品（products）也是类似的。产品 Product 是一个服务描述，设施使用，资产使用，或是一个数码或实际的用于销售的商品。对于制造业来说，产品可以是原材料，组件，成品等。产品的类型通过 **productTypeEnumId** 字段指定并且可用的操作使用 ProductType 类型定义在枚举（Enumeration）数据中。

一个产品 Product 的实例被追踪于不同的地点取决于产品的类型。实体产品使用资产（Asset）和关联的实体作为库存被追踪。资产使用产品作为资产记录被追踪，同时它们的排程对应使用工作投入（WorkEffort）记录关联。设施使用产品作为设施 Facility 记录被追踪，并同样的使用工作投入 WorkEffort 去运行排程。服务产品通过关联请求（Request）和需求（Requirement）记录的项目，任务等各种工作投入 WorkEffort 被追踪。服务通常有一个或者多个当事人 Party 关联记录，因为个人或者组织会执行或被要求执行服务。



产品 Product 实体有一个状态字段（**statusId**），但这用于比较特殊的情况，通常并不是为了某个确定的事情而是看起来像状态值但实际上被建模为日期类型，包括：销售宣传时间（**salesIntroductionDate**），销售中止时间（**salesDiscontinuationDate**）以及供货中止时间（**supportDiscontinuationDate**）。如果你想了解一个产品是否可销售，你只需检查当前时间和销售时间字段的对比，而无需查看指示或者状态字段。

对于产品内容来说，有产品名称 **productName** 和描述 **description** 字段，并且其他的一切例如本地化名称/描述，细节描述，图像，说明，警告，按钮/链接文本等信息都记录在 ProductContent 实体中。**contentLocation** 字段指向一个资源门面的本地路径，这样内容就可以存在数据库中（通过 DbResource/File 实体），JCR 中（java 内容仓库，如 Apache JackRabbit），本地文件系统中或者其他任何配置的开箱即用的本地资源或者你自己指定的方式。参见 ***Resource Locations*** 部分获得更多的细节。

产品有库存（**requireInventory**, **requirementMethodEnumId**），货运（**chargeShipping**, **inShippingBox**, **defaultShipmentBoxTypeId**, **returnable**）以及税金（**taxable**, **taxCode**）设置。有些产品关联了一个总额，例如一箱产品中有多个罐头或者允许用户采购时输入总数量。这种方式使用了 **amountUomId**, **amountFixed**, 和 **amountRequire** 字段。

ProductDimension 实体记录了各种可能的产品规格信息。它包含了重量，长度，货运，数量，价格以及其他你想要定义的规格。你可以通过 ProductDimensionType 类型添加其他规格的 Enumeration 枚举记录。例如 UPC，ISBN，EAN 等的识别码（ProductIdentification）也有类似的结构。

产品有一个 **originGeoId** 字段去指定进出限制或者纯粹想要知道的产品来源地。关于更多的货运或者库存限制的地理 Geo 细节信息存储在 ProductGeo 实体中。

产品使用 ProductAssoc 实体关联其他的产品。这十分的有用，对于交叉/起卖，大小/颜色差别，配件以及制造业中的 BOM 细节。ProductParty 实体用于关联产品 Product 和当事人

Party。ProductReview 实体用于记录用户/客户复核及评级。

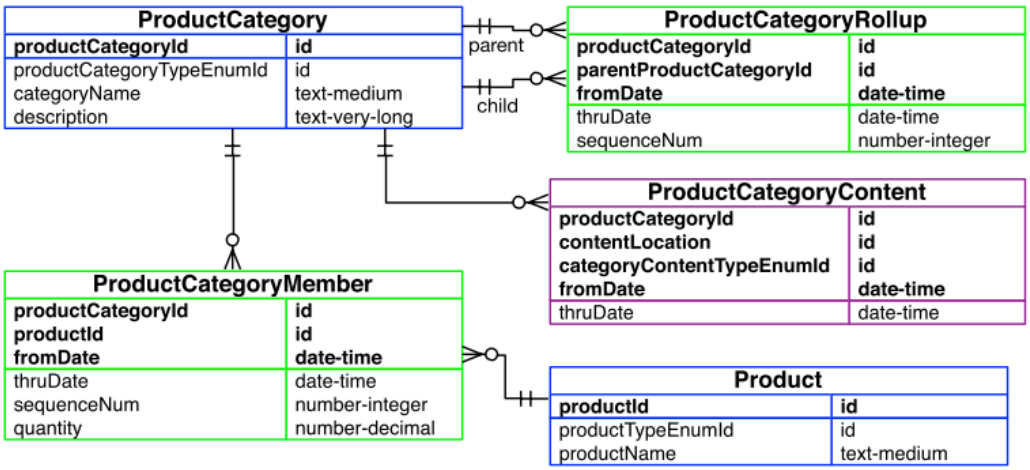
ProductPrice 实体用于各种价格，包括源于供应商的价格和对客户的报价（经由 **vendorPartyId** 和 **customerPartyId** 字段）；清单，流通，最大/最小，促销，竞价等价格（**priceTypeEnumId**）；采购，重复的以及使用价格（**pricePurposeEnumId**）。对于数量中断这里有一个最小数量（**minQuantity**）字段（对于任意大于或等于，小于下一个最高的数量匹配最小数量 **minQuantity**）。

价格在一个时间段（**fromDate**, **thruDate**）内有效，并且特别的受商店 ProductStore（**productStoreId**）的限制。对于带有增值税的司法辖区来说，价格通常含有税金（**taxInPrice**=true），并且使用其他的税金字段去指定明细（**taxAmount**，**taxPercentage**，**taxAuthPartyId**，**taxAuthGeoId**）。参见 ***mantle.other.tax*** 部分查询更多关于税金的计算细节。

实际的价格存在 **price** 字段中并且币种使用 **priceUomId** 字段。参见数据模型资源手册书中 ***Units of Measure*** 章节查询更多关于 UOMs 的细节。重复出现的价格存在 **termUomId** 字段中，同时这个价格是每个单位的价格（例如时间，数据量等）。

➢ *定义 - 分类 (mantle.product.category)*

产品分类（ProductCategory）是一个明显的将产品组成一个类别的用法，但只能是其中的一个类型（**productCategoryTypeEnumId**）。更通常的做法是一个分类是一个产品的集合。其他通用的类型包括税金，交叉销售，库存，检索和热销。



产品使用 ProductCategoryMember 实体和分类进行关联，并且它们关联方式是多对多的关联关系（产品可以有很多的分类，一个分类可以有多种产品）。分类使用 ProductCategoryRollup 实体去关联父级/子级分类，它也是种多对多的关系，这样一个分类可以有多个父级和子级的分类。它们都有有效时间（**fromDate**，**thruDate**）以及流水编号（**sequenceNum**）字段去按照类别以及分类的子类去排序产品。

ProductCategoryMember 同样有个数量字段（**quantity**）用于某个分类表现为一组产品是以某些专门的包装组成一起的形式进行组合的。

资源门面的内容使用 ProductCategoryContent 实体与分类进行关联。类似的，当事人使用 ProductCategoryParty 实体与分类进行关联。

➢ *定义 - 配置 (mantle.product.config)*

产品配置实体用于特定搭配好的产品（Product.**productTypeEnumId**=PtConfigurableGood）

的配置操作。ProductConfigItem 实体用于指定配置项，ProductConfigItemAppl 实体用于配置应用于可配置商品的产品信息记录。配置项的操作记录在 ProductConfigOption 实体中，并且操作使用 ProductConfigOptionProduct 实体与其他产品 Product（有自己的库存，价格，供应商等细节）进行关联。

为了有助于阐明区分**可配置**的产品和**组件**产品，这里有一个路径： Product ==> ProductConfigItemAppl ==> ProductConfigItem ==> ProductConfigOption ==> ProductConfigOptionProduct ==> Product。

ProductConfigOption 实体有一个描述（**description**）字段，本地描述信息和内容通过 ProductConfigItemContent 实体形成一个与资源门面内容的本地路径相关联的操作使用。

当一个可配置产品被配置过了，并要添加到订单时，我们使用 ProductConfigSaved 实体来保存其配置信息，并且使用 OrderItem.**productConfigSavedId** 字段来为订单条目参照相应的配置。对于每个条目中已保存的配置操作被记录在 ProductConfigSavedOption 实体中。

- ***定义 - 成本 (mantle.product.cost)***

成本构成（CostComponent）记录用于分解产品 Product 的成本，特别是制造业的产品。供应商采购来的产品都有个相近的成本价格，价钱支付给供应商。成本构成包含估算的和实际原材料，供给，设备使用以及其他成本。各种的成本构成在一个时间范围内汇总到一起就形成了产品的成本。

CostComponentCalc 实体和 ProductCostComponentCalc 实体作用于产品上，用于指定一个成本构成 CostComponent 如何被计算的或者是产品的成本基于什么组成的。

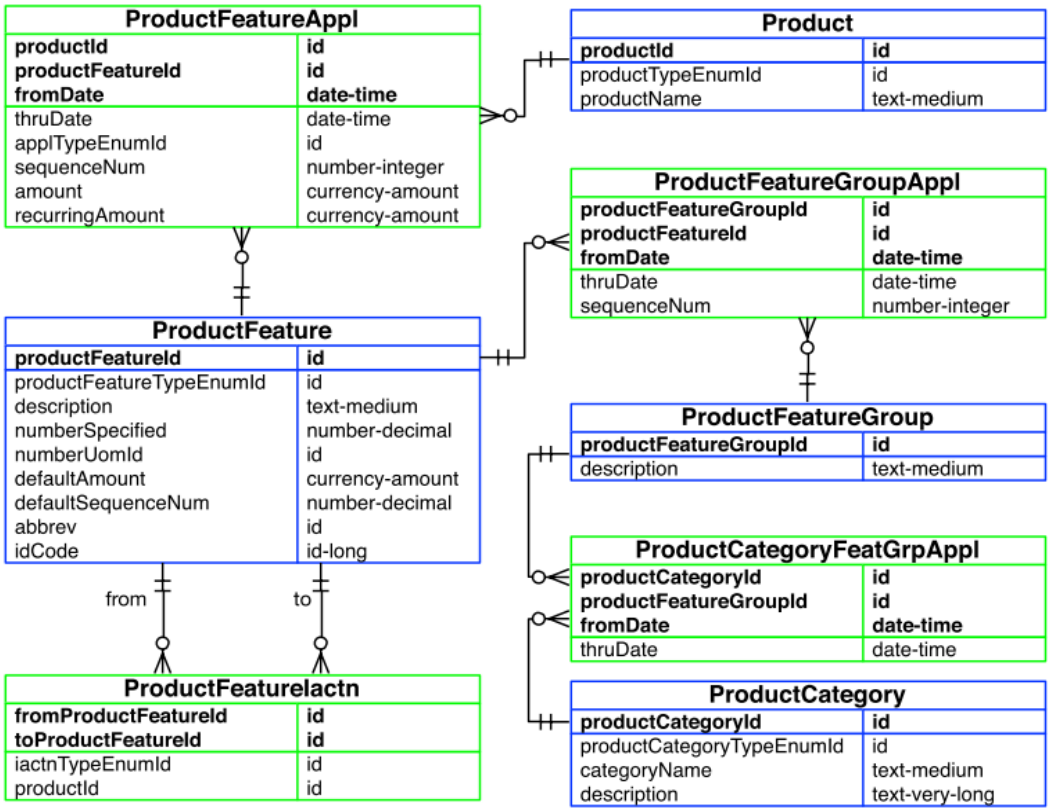
ProductAverageCost 实体用于特定的设施 Facility 和特定的企业 Organization 持续的追踪产品 Product 一段时间(通过起始/截止有效时间段)的平均成本。这对于计算销货成本(COGS)的目的来说十分有用，通过计算的是一个历史的平均成本而不是基于销售的某个实际产品的成本。

- ***定义 - 特性 (mantle.product.feature)***

产品特性（ProductFeature）通过结构化的方式描述产品 Product。这里有很多开箱即用的特性类型（**productFeatureTypeEnumId**）定义，像品牌，颜色，面料，许可证以及大小。通常可以通过 ProductFeatureType 类型去添加 Enumeration 枚举记录。

特性通过 ProductFeatureAppl 实体去应用到产品上，并使用 **applTypeEnumId** 字段去指定产品（可选操作的特性，产品内在方面的标准，或者是区分差异描述，或者是虚拟产品）的特性类型，同时还有时间区间字段（(**fromDate**, **thruDate**）。

有时有些特性是互斥或者依赖的需求，这时就要使用 ProductFeatureIactn 实体了。



特性是各种类型自然的组织构成，但通常为特定的目的去定义特性集合十分的有用，例如为检索出明确的产品或者用于描述产品的明确类型（通常是管理的需要）。ProductFeatureGroup 实体就是为这个使用的。ProductFeatureGroupAppl 实体用于指定特性属于哪个分组。ProductCategoryFeatGrpAppl 实体用于关联产品分类 ProductCategory 和特性分组。

➢ *定义 - 订阅 (mantle.product.subscription)*

订阅（Subscription）用于记录当事人（**subscriberPartyId**）在一个时间区间（**fromDate**, **thruDate**）去访问订阅资源（SubscriptionResource）的信息。它特定用于关联采购订阅的订单条目 OrderItem 和创建订阅去便于产品 Product 采购。除此之外或可选的是，订阅的时间区间可以按照实际使用时间（**useTime**）而不是日历时间或者使用量（**useCountLimit**）来限制。

当产品 Product 使用 ProductSubscriptionResource 实体被采购时，可以配置订阅资源 SubscriptionResource 可访问的信息，有 **availableTime**, **useTime** 和 **useCountLimit** 字段。

SubscriptionDelivery 实体用于持续追踪关联订阅的沟通事件（CommunicationEvent），特别是数字资源的订阅交货。

➢ *资产 - 资产 (mantle.product.asset)*

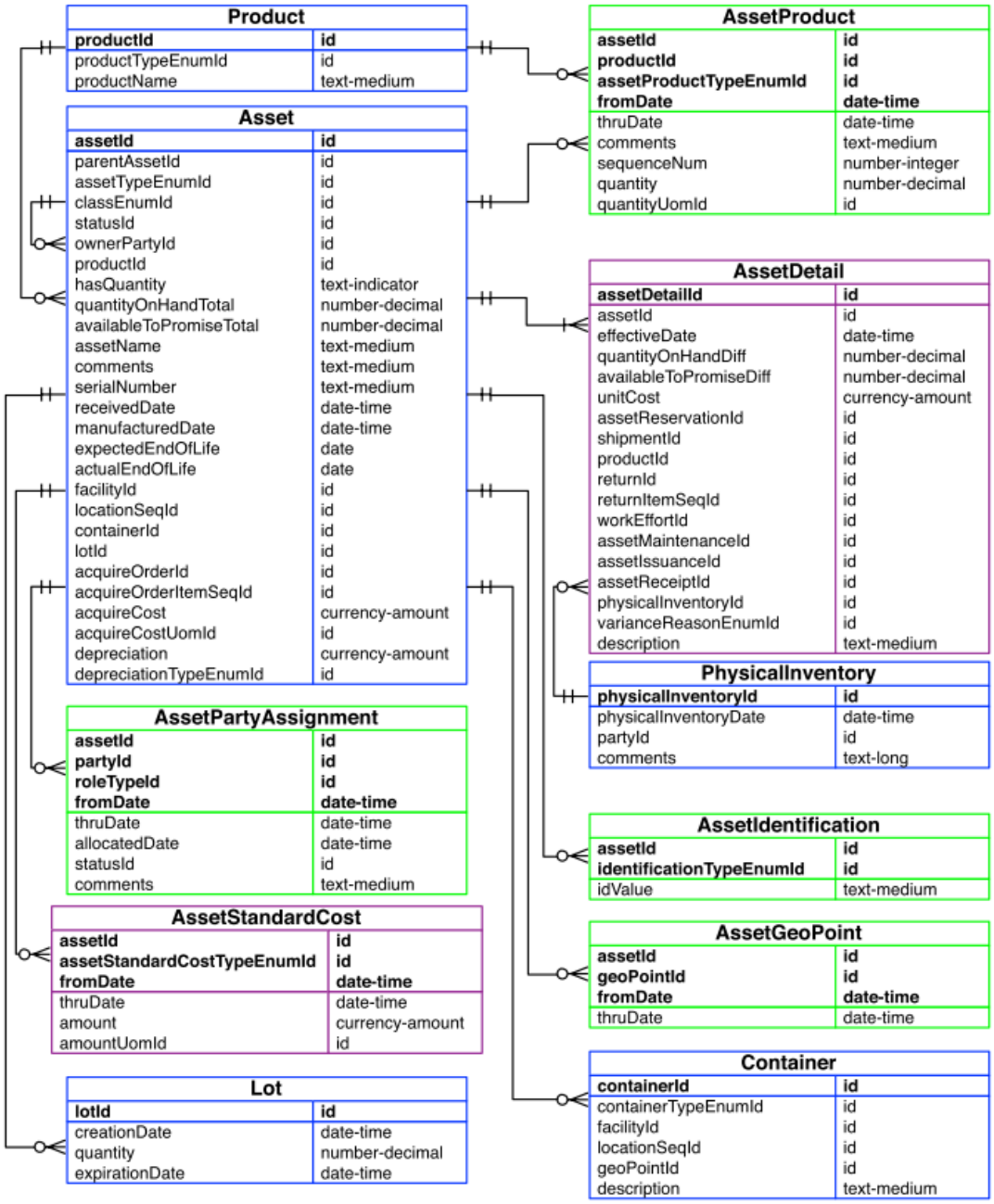
资产（Asset）用于库存，设备以及作为财务追踪的固定资产（**assetTypeEnumId**）的其他任何事物。资产通过 **assetId** 字段去鉴别。资产有一个分类（**classEnumId**）字段，如叉车，拖拉机，笔记本电脑，甚至软件去用于区分资产的类型，特别是制造业中为找到设备需求路线（生产任务）的特定目标。你可以通过 AssetClass 类型去添加资产分类的 Enumeration 枚

举记录。

资产 Asset 通常表现为一个产品 Product 的实例，或者换句话说产品 Product 记录描述的一个实物。**productId** 字段指定产品是哪个。资产有资产名称（**assetName**）字段和用于记录注释/说明的注释（**comments**）字段。

资产有很多开箱即用的状态类型（**statusId**）字段用于序列化的库存和设备。一个序列化的库存资产得名于其表现为单个实物并通常有一个序列号（**serialNumber**）。

非序列化的库存资产（**hasQuantity**=Y）体现为处理简单的库存需求多于一个数量并且都相同，无需独立追踪的产品。当前在手实物数量维护在 **quantityOnHandTotal** 字段中，同时这个数量可以在 **availableToPromiseTotal** 字段中被保留或者确定。产品的库存通常由多种资产 Asset 的记录组成，他们有相同的 **receivedDate**, **lotId**, **facilityId**，**locationSeqId**（FacilityLocation 中用于记录资产存储的位置）和 **ownerPartyId**（通常是内部仓库拥有者的当事人 **Party**）字段，以及可用的状态字段（**statusId**）。通常接收到一批货就创建一条新的资产 Asset 记录。



资产实体中的数量字段有总数 Total 这样的后缀，这因为该字段的值是通过 AssetDetail 实体的 **quantityOnHandDiff** 和 **availableToPromiseDiff** 字段计算出来的。每个资产明细（AssetDetail）记录都代表了资产的变化,例如确认的订单（**assetReservationId**）的资产数量保留，产品运输的发货（**assetIssuanceId**），接收到货（**assetReceiptId**），实际库存盘点（**physicalInventoryId**, **varianceReasonEnumId**），以及作为生产路线的工作投入（**workEffortId**）的生产和消耗。如果涉及到货运，用 **shipmentId** 字段进行记录，并且所有的细节都需要有有效日期 **effectiveDate** 的字段。

如果实际库存总数盘点完成，需记录在 PhysicalInventory 实体中，并且每个库存的差异细节都需记录在 AssetDetail 实体中。

资产可能需要很多的日期字段来进行进行使用：收件时间（**receivedDate**），接收时间（**acquiredDate**），生产时间（**manufacturedDate**），预计失效时间（**expectedEndOfLife**）以及实际失效时间（**actualEndOfLife**）。为了追踪采购的资产和实际成本数据，资产 Asset 使用了 **acquireOrderId**, **acquireOrderItemSeqId**, **acquireCost** 和 **acquireCostUomId** 字段。对于固定资产折旧追踪并添加到相应的会计事务 AcctgTrans 记录中，资产包含了折旧（**depreciation**），折旧类型（**depreciationTypeEnumId**）和 残值（**salvageValue**）字段。

AssetGeoPoint 实体用于记录资产的位置以及其曾经的位置历史信息（通过起止时间）。AssetIdentification 实体作为追踪文本编号，生产序列号，VIN 等，用于资产的 ID 编号信息。资产 Asset 使用 AssetPartyAssignment 实体为了使用，管理，生产等目的，与当事人 Party 按照某个特定的角色和有效时间段（**fromDate**, **thruDate**）进行关联。

既然资产 Asset 是产品 Product 的一个实例，那么产品额外的信息会关联到资产上去展现，例如租金或者出售的资产等信息。AssetProduct 实体用来持续追踪这些关联的产品。

库存资产 Asset 以及某些时候其他类型的资产，通常使用 FacilityLocation 实体的 **facilityId** 和 **locationSeqId** 字段去记录位置，同时，他也可能位于一个集装箱中，使用 Container 实体去跟踪运输轨迹。这种情况下，**facilityId** 和 **locationSeqId** 字段值为空并且 Asset.**containerId** 字段会被填充。之前一种场景下，实际的资产存放位置会在 **facilityId** 和 **locationSeqId** 字段中找到，并且 **geoPointId** 字段也可以使用。这些字段都是审计记录用于持续追踪类似集装箱运输的资产位置的变化。

- ***资产 - 发货 (mantle.product.issuance)***

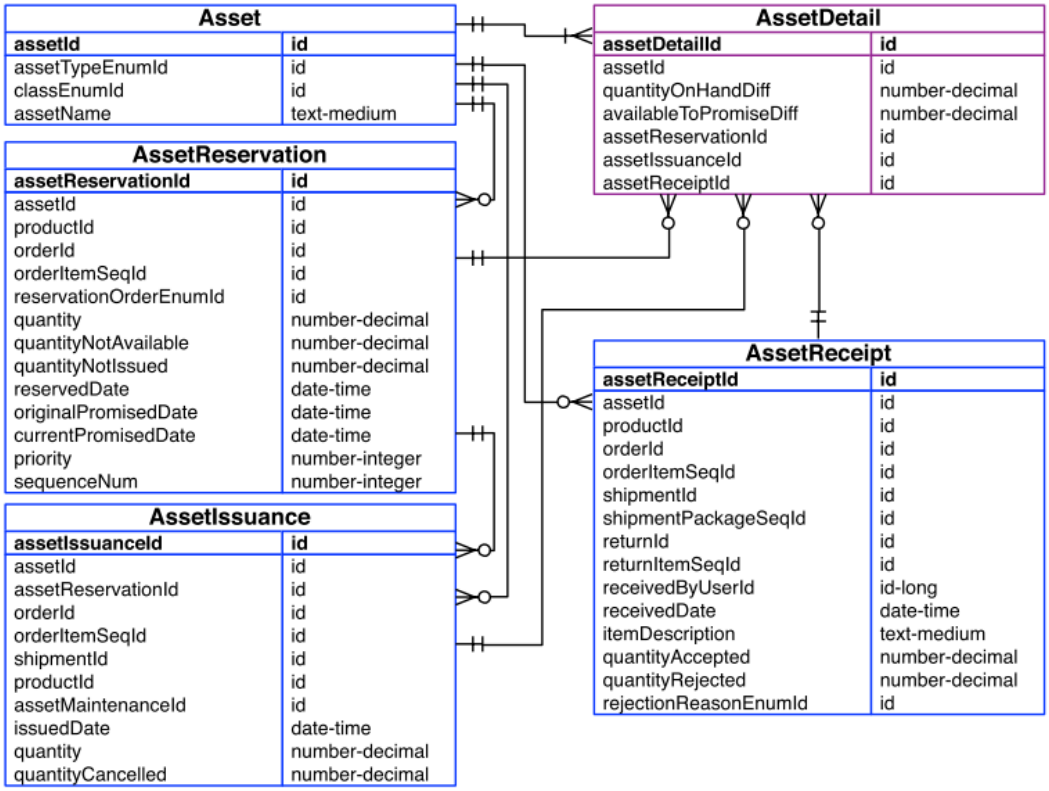
因为竞争特定的库存物品是很常见的，好比产品的销售订单下单时库存有限，就十分有必要去追查 AssetReservation 实体中的订单条目下单时就确定的资产保留记录。稍后当实物产品数量被填充足额时，一条 AssetIssuance 实体记录被创建，同时 AssetReservation 中失效的记录被删除。

资产保留用于方便的关联资产 Asset（**assetId**），产品 Product（**productId**）和订单条目 OrderItem（**orderId**, **orderItemSeqId**）。资产保留中的保留数量（**quantity**）字段用于保留在手库存之外的一个数量，这个数量是不允许被答应抢占的，通过 **quantityNotAvailable** 字段追踪。

发货用于关联资产 Asset（**assetId**），资产保留 AssetReservation（**assetReservationId**，如果可用的话），产品 Product（**productId**，方便起见），订单条目 OrderItem（(**orderId**, **orderItemSeqId**），货运 Shipment（**shipmentId**）或者资产发货的资产维护 AssetMaintenance（**assetMaintenanceId**）信息。发货有发货时间（**issuedDate**）以及发货的数量（**quantity**），撤销数量（**quantityCancelled**）也可以被使用。

发货与特定角色的当事人通过 AssetIssuanceParty 实体进行关联。

当资产发货（AssetIssuance）记录创建时将会触发财务过账事务去扣除在手的库存值。这是标准的会计业务中库存销售的部分。



- ***资产 - 接收 (mantle.product.receipt)***

当资产 Asset 特别是库存资产被接收到，AssetReceipt 实体用于追踪接收信息。方便起见，资产的 **productId** 字段和条目描述（**itemDescription**）字段也记录在其中。接收用于关联订单条目 OrderItem（**orderId**, **orderItemSeqId**），运输打包 ShipmentPackage（**shipmentId**, **shipmentPackageSeqId**），运输条目 ShipmentItem（**shipmentId**, **productId**）以及退货条目 ReturnItem（**returnId**, **returnItemSeqId**）。

这里还有很多的字段用于追踪资产的接收者（**receivedByUserId**），接收日期（**receivedDate**），接受数量（**quantityAccepted**）和拒收数量（**quantityRejected**），并且如果存在拒收数量，还有一个原因（**rejectionReasonEnumId**）字段。

当资产接收（AssetReceipt）记录创建时会触发总账会计业务去增加在手库存数量。这是标准的会计业务中库存采购的部分。

- ***资产 - 维护 (mantle.product.maintenance)***

基于资产 Asset 是产品 Product 的一个实例的模式，产品描述资产就包含了关联产品的维护计划，通过 ProductMaintenance 实体进行构建单据。这里通过 **maintenanceTypeEnumId** 字段去指定很多种维护类型，例如换油和清洁。你可以通过 MaintenanceType 类型去增加 Enumeration 枚举记录。

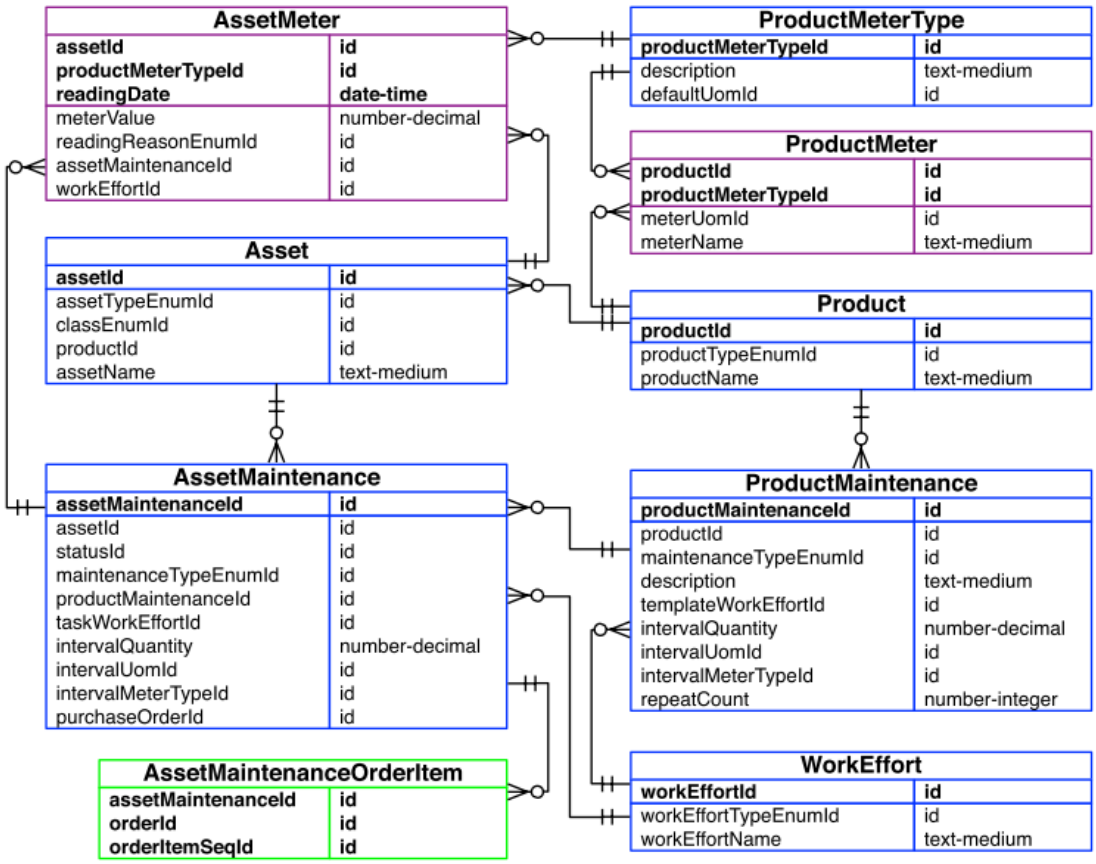
维护按照单位为 **intervalUomId**，数量为 **intervalQuantity** 字段指定的间隔去进行完成。

间隔可能按照米去度量，资产的产品测量类型（ProductMeterType）通过 **intervalMeterTypeId** 去定义。测量同样可以通过 ProductMeter 直接与产品进行关联。如果重复次数（**repeatCount**）字段在 ProductMaintenance 记录中被指定，则维护就需要按照次数进行完成。

维护可以通过工作投入（WorkEffort）去追踪，并且简化起见，这种预定义好的工作投入可以作为一个模板和副本，从维护计划（ProductMaintenance.**templateWorkEffortId**）到实际的维护（AssetMaintenance.**taskWorkEffortId**）记录，去分配任务，状态变化等等。

对于特定资产实际的维护投入来说，当维护实际执行时，AssetMaintenance 实体有类似的字段，像是 **intervalQuantity** 字段以及关联的字段。这里还有个状态字段（**statusId**）用于追踪计划和维护的完成度。它还有个 **purchaseOrderId** 字段用于当维护工作被外包出去的时候追踪相应的订单。维护工作可以采购或者销售，并且关联的订单条目通过 AssetMaintenanceOrderItem 实体进行追踪。

当维护完成时通常会测量，每个测量的值都会通过 AssetMeter 实体与产品（ProductMeter）关联记录下来。其他的维护环境之外的测量也会通过 AssetMeter 进行记录。这些通常都是在加油，路线的航点，生产任务的前/后等（**readingReasonEnumId**）并且这些记录通常对于财务管理和纳税义务来说都很重要。



AssetRegistration 实体记录了在政府权威机构（**govAgencyPartyId**）注册的资产 Asset 信息，其中包括了授权号（**licenseNumber**）和注册登记号（**registrationNumber**）。注册发生在一个确定的时间（**registrationDate**）并在某个时间段（**fromDate**, **thruDate**）内有效。

- ***商店 (mantle.product.store)***

对于电子商务网站或者销售终端 POS（point of sale）系统的销售订单过程来说，我们需

要一种方式去追踪所有的关联设置。ProductStore 以及关联的实体就用于这种场景。

商店有一个名字（**storeName**）以及被某个内部组织（**organizationPartyId**）所拥有/运转。虽然商店可以提供多语言和币种的支持，但是关注于某个国家/地区的商店还是最好使用单一的语言（**defaultLocale**）以及币种（**defaultCurrencyUomId**）。

商店包含：

- **可用的产品：** ProductStoreCategory 实体关联商店和产品分类（ProductCategory）记录去浏览根，默认为搜索，允许采购等，并且这些分类或者子分类的产品构成了商店中可用的产品。
- **电子邮件通知：** ProductStoreEmail 实体关联 Moqui 框架的 EmailTemplate 实体和商店用于电子邮件通知诸如登记，订单确认，订单变更，退货完成，密码变更等。
- **库存预订：** 最常用的场景是使用单个的库存设施（Facility）作为商店，通过 ProductStore.**inventoryFacilityId** 字段去进行指定。当多于一个的设施被需要时，使用 ProductStoreFacility 实体。ProductStore.**reservationOrderEnumId** 字段指定了订单中的库存保留方式，如按照接收时间或者截止时间先进先出（FIFO）或者后进先出（LIFO）。ProductStore.**requirementMethodEnumId** 字段可以设置自动补给的需求。
- **支付过程：** ProductStorePaymentGateway 实体用于配置每种付款方法（**paymentMethodTypeEnumId**）的支付网关（PaymentGatewayConfig）。
- **运输操作和运价计算：** ProductStoreShippingGateway 实体用于配置每个运输（**carrierPartyId**）的货运配置（ShippingGatewayConfig）。
- **税金计算：** ProductStore.**taxGatewayConfigId** 字段指定了商店销售/增值税计算的配置（TaxGatewayConfig）记录。

ProductStoreParty 实体通常用于关联特定角色的当事人和商店的需求。其中一个十分有用的用法是如果 ProductStore.**requireCustomerRole** 字段被设置为 Y，那么只有客户角色的当事人才允许访问商店。

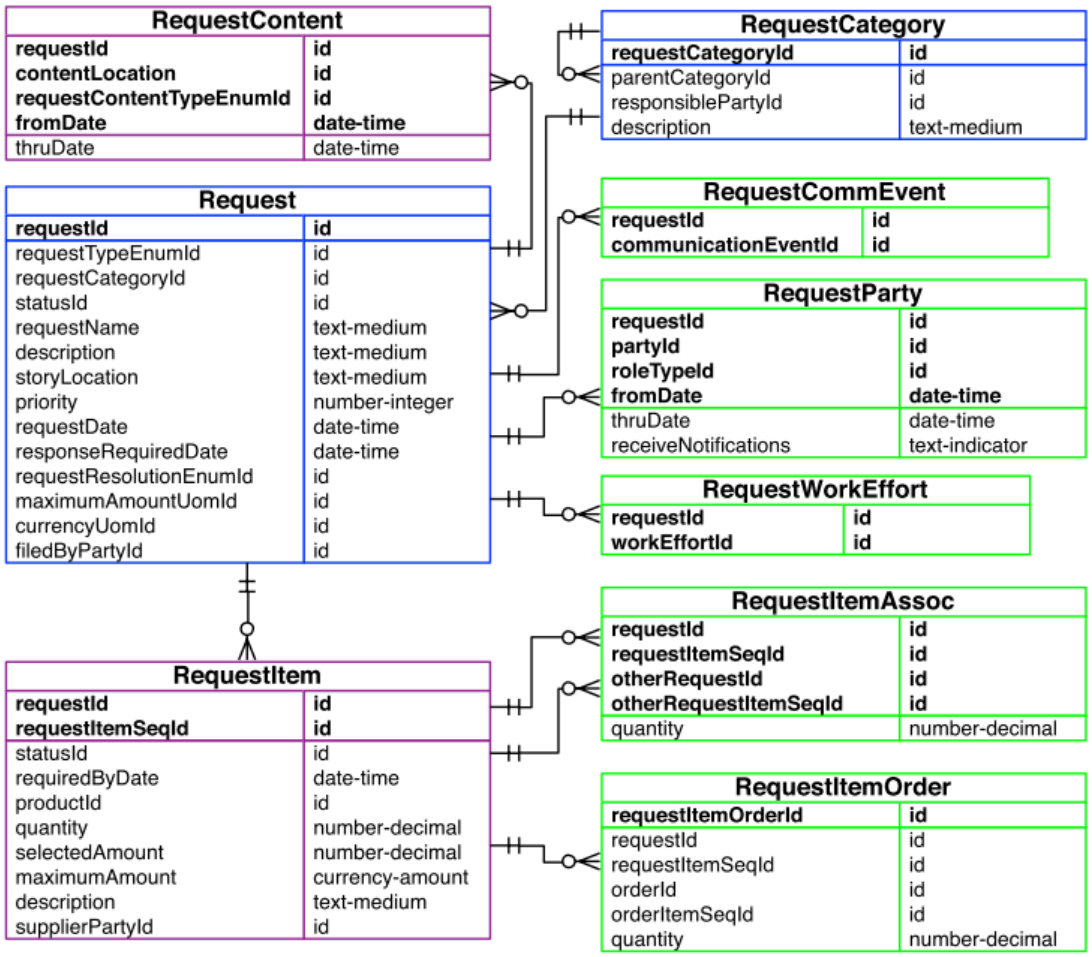
当管理大量的店铺或者使特定的店铺集合自动化管理时，使用 ProductStoreGroup 实体去展现一个店铺组，并且 ProductStoreGroupMember 实体用于关联商店和组。一个商店可以关联多个分组。ProductStoreGroupParty 实体用于关联当事人和分组。

请求

➢ *请求（mantle.request）*

请求 Request 可以来源于企业内的一个当事人（**filedByPartyId**），例如一个雇员的库存或一般采购请求，或源于企业外如对一个终端或顾客的报价、建议，抑或是在软件世界里的 bug 修正或新的特性。这些都在 **requestTypeEnumId** 字段中进行指定，同时虽然这里有一些开箱即用已定义好的选项，但是你也可以添加 RequestType 类型的 Enumeration 记录来定义其他的。

默认的 Request 状态（**statusId**）包含了草稿，已提交，已评审，处理中，已完成以及撤销。它同样有个解决状态类型（**requestResolutionEnumId**）的字段用于默认未解决的请求，默认的选项包括已修正，不可复现，无法确定，重复，已拒绝以及信息不足。额外的解决类型可按照 RequestResolution 类型的 Enumeration 记录的方式进行添加。**fulfillContactMechId** 字段用于指定发送解决的结果到何处。



Request 有名称（**requestName**），描述（**description**）字段，以及如果在资源门面内容中有额外的业务故事细节，使用 **storyLocation** 字段进行引用。为了有助于决定请求上的工作次序和一般的信息，请求有 **priority**，**requestDate** 和 **responseRequiredDate** 字段。请求可能会关联一个 Facility（**facilityId**）和 ProductStore（**productStoreId**）。

请求的细节存放在 RequestItem 记录中。一个条目可以有自己的状态 **statusId**（使用与请求相同的状态值），请求时间 **requiredByDate** 并典型的有个描述字段 **description**。如果请求是服务于 Product，则使用 **productId**，**quantity** 以及 **selectedAmount** 字段去指定细节。

对于报价和其他类似类型的请求，对于条目有个支付的最大金额/价格，通过条目上的 **maximumAmount** 字段进行指定。金额的单位在 Request 记录上的 **maximumAmountUomId** 字段中。这些类型的请求同样具有代表性，最终会形成一个订单，同时 RequestItem 使用 RequestItemOrder 实体去关联 OrderItem。

手工组织请求使用 RequestCategory 去指定层级结构（通过 **parentCategoryId** 字段）的请求分类，并使用 Request.**requestCategoryId** 字段与请求进行关联。

请求可能会关联 CommunicationEvent 用来关联请求和沟通信息（RequestCommEvent），还有额外的内容或文档使用资源门面内容（RequestContent），工作在请求上的当事人 Party 或其他与请求相关的（RequestParty），和用于处理请求相关的任务及其他工作的工作投入 WorkEffort（RequestWorkEffort）。请求可能还有说明（RequestNote）。

举个为某个软件 bug 进行修正的新建请求的例子。这个请求通过一条 RequestParty 记录去指派到某个人。这个人创建了一个关联到这个请求的任务（WorkEffort），信息通过 RequestWorkEffort 进行记录。这个任务可能会指派到同一个人或其他人，或甚至一个组。一

旦任务完成，它的状态就会更新，请求的状态也一样会更新。

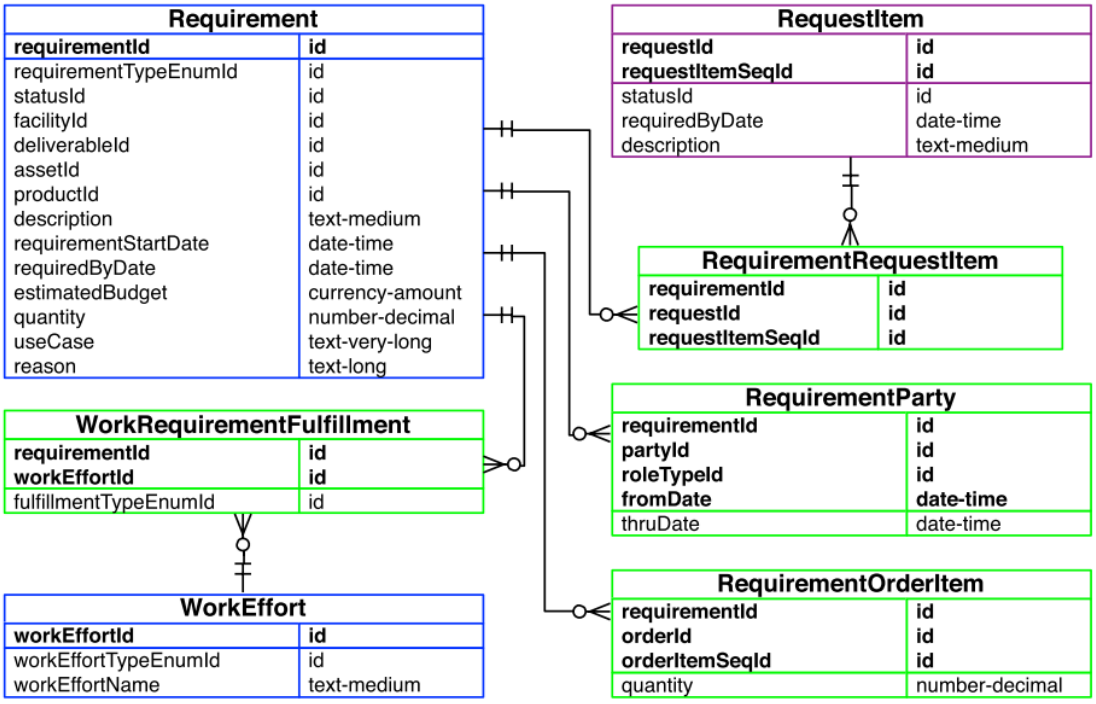
➢ *需求（mantle.request.requirement）*

需求 Requirement 可能用于工作，库存，普通客户或内部的需求等（**requirementTypeEnumId**）。你可以添加自己的 RequirementType 类型的 Enumeration 记录。需求的状态（**statusId**）包括推荐的，已创建，已批准，已订购，以及已拒绝。库存需求及其它可应用的类型可能用于特定的 Facility（**facilityId**）和 Product（**quantity**）。

需求将有典型的 **requirementStartDate** 和 **requiredByDate** 字段。为了描述需求的细节，特别是软件需求，**useCase** 和 **reason** 就是你需要的。当事人可使用 RequirementParty 实体关联到需求上。

对于自动库存补给的库存需求来说，可以基于 ProductStore **requirementMethodEnumId** 字段的设置进行创建。普通的选项包括：基于每个订单创建一个需求，当 ATP（可承诺量）或 QOH（在手数量）降到关联的 ProductFacility 记录配置的水平之下时，或第三方直接发货的订单目的。需求创建之后，它们可以被 Product 和 Facility 进行概括，然后选择了一个供应商之后，一个总数量的订单会被创建并关联到 RequirementOrderItem 实体上。

工作需求遵循一个不同的途径。它们可能为工作会有一个订单关联，但更通常的是会导致产生一个具体的 RequestItem（使用 RequirementRequestItem 进行关联），或直接生成一个 WorkEffort（使用 WorkRequirementFulfillment 进行关联）。工作投入类型可以为工具，修理，部署，测试或交付（**fulfillmentTypeEnumId**）。



Requirement 实体有一个简单的预估预算 **estimatedBudget** 字段，同时对于更复杂的预算需求或包含在一个更大的预算计划中时，需求可以使用 RequirementBudgetAllocation 实体去关联 BudgetItem。

销售

➢ *商机 (mantle.sales.opportunity)*

作为销售自动化（SFA）的一部分，SalesOpportunity 实体用于持续追踪商机。一个商机是典型的关联于一个明确的销售阶段（SalesOpportunityStage），同时你可以定义任意系列期望的阶段。

可能有很多的当事人会关联到一个商机上，包括客户/预期，销售代表，经理等。使用 SalesOpportunityParty 实体去记录他们。你同样可将其应用于竞争对手，但是通常对于竞争对手有额外的信息，所以使用 SalesOpportunityCompetitor 实体去记录它们。

商机经常会关联到一个报价，报价可能会转化为一个订单。使用 SalesOpportunityQuote 实体去持续追踪它们。一个商机可能会关联会议，其他日历事件，或者任务，并使用 SalesOpportunityWorkEffort 实体去关联它们。

这里有一对市场记录的触点。一个是使用 SalesOpportunity.**marketingCampaignId** 字段去指向市场活动 MarketingCampaign 实体。另一个是市场追踪代码 TrackingCode 记录，它们通过 SalesOpportunityTracking 实体被关联。查看 ***Marketing*** 部分获得这些内容更多的细节。

➢ *预测 (mantle.sales.forecast)*

销售预测 SalesForecast 可能是服务于一个完整的内部组织（**organizationPartyId**），或是这个组织 Organization（**internalPartyId**）内特定的当事人 Party。它关联一个时间区间并有总额字段用于最终的结果，包括 **quotaAmount**，**forecastAmount**，**bestCaseAmount** 及 **closedAmount**。

实际的产品 Product 销售细节被记录在 SalesForecastDetail 实体中，每条记录都有每个产品 Product 和/或产品分类 ProductCategory 的售价 **amount** 和销售数量 **quantity**。

➢ *需要 (mantle.sales.need)*

当客户或其他当事人 Party 需要产品时（可以是内部的或外部的），使用 PartyNeed 实体进行记录。它可以是一个产品 Product 和/或一个各种产品的产品分类 ProductCategory 以满足需要，或当还不知道明确的产品需要时。它通常源于一个沟通事件 CommunicationEvent，或者通过一个 web 应用的访问 Visit，因此这里对两者都有字段进行记录。

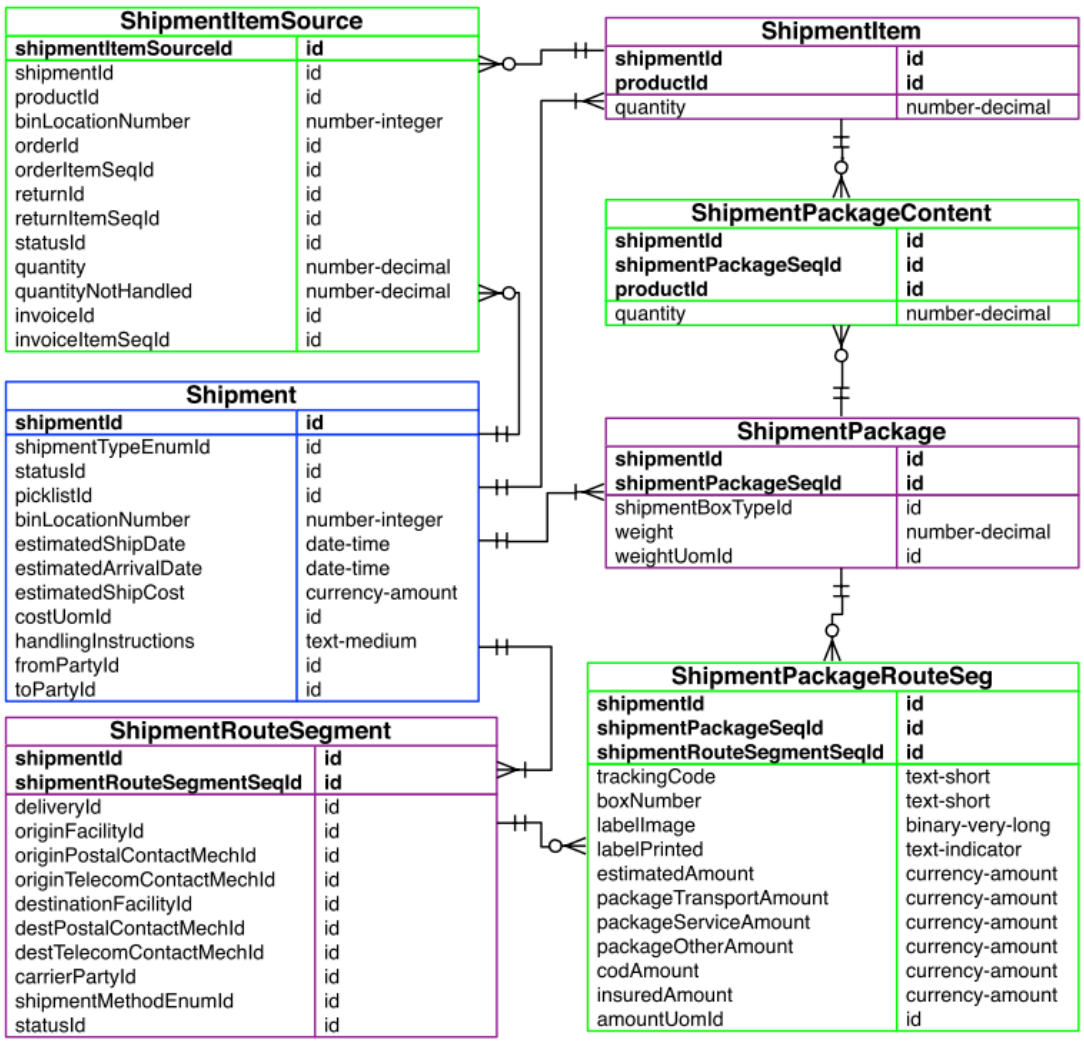
装运

➢ *装运 (mantle.shipment)*

装运（Shipment）和关联的实体可用于传入和传出的货运类型（**shipmentTypeEnumId**），并更多的指定用于销售退货，销售货运，采购货运，采购退货，供应商直接发货以及转移发货。装运 Shipment 通常是从一个当事人 Party（**fromPartyId**）发往另一个当事人（**toPartyId**）的。如果需要可以在 **handlingInstructions** 字段中存放特殊的介绍。

为了计划的目的，装运有预计准备日期（**estimatedReadyDate**），预计发货日期（**estimatedShipDate**），预计到达日期（**estimatedArrivalDate**）以及最新取消日期（**latestCancelDate**）值。为了进一步的了解细节，将装运作为一个事件放入日历中，可以使用 **shipWorkEffortId** 和 **arrivalWorkEffortId** 字段去指向工作投入（WorkEffort）记录。这里还有些装运的预计成本信息，使用 **estimatedShipCost** 以及币种的 **costUomId** 字段进行追踪。**addtlShippingCharge** 字段用于调整货运成本，并且 **addtlShippingChargeDesc** 字段用于描述附加的费用信息。

对于全部的装运 Shipment 来说，这里有个状态（**statusId**）字段，状态包括投入，排程，分拣，装箱，运输，送达以及撤销（Input, Scheduled, Picked, Packed, Shipped, Delivered 以及 Cancelled）。这个字段是状态历史的审计记录。**Packed** 打包状态是其中一个重要的关键，为了开票的目的装满运输。打包状态的变化用于触发运输订单的发票创建，并且可能的话可以使用支付过程。



每个装运都有个装运条目 ShipmentItem 的记录，并且装运的产品 Product（**productId**）都有个数量（**quantity**）字段。

装运 Shipment 通常有一个或多个包裹（ShipmentPackage），并且 ShipmentPackageContent 实体中记录了每个包裹中产品（**productId**）的数量 **quantity**。每个包裹都有可能会使用到箱子（**shipmentBoxTypeId**，指向一条装运箱类型 ShipmentBoxType 记录），同时装运的包裹总重量 **weight** 是和其重量单位（**weightUomId**）有关系的。

一个装运 Shipment 总有一条或多条航段（ShipmentRouteSegment）。最简单的完成客户发货是一个承运人（**carrierPartyId**）使用某种运输方式（**shipmentMethodEnumId**）走单一的路线从明确的起点（**originPostalContactMechId**, **originTelecomContactMechId**）到目的地（**destPostalContactMechId**, **destTelecomContactMechId**）。装运可能有其他的联系信息关联记录保存在 ShipmentContactMech 实体中。

对于完成客户发货来说，起点通常是一个仓库设施并通过 **originFacilityId** 来指定。对于客户退货或者库存采购运输来说，通常用 **destinationFacilityId** 字段来记录到达的目的设施。这里有很多的日期信息来记录路线相关信息，包括预计开始日期（**estimatedStartDate**），预计到达日期（**estimatedArrivalDate**），实际开始日期（**actualStartDate**）和实际到达日期（**actualArrivalDate**）。

每个包裹对于每条航段来说都有明确的细节信息（ShipmentPackageRouteSeg），包括 **trackingCode**, **boxNumber**（装运中可选字段），以及含有 **labelImage**, **labelIntlSignImage**, **labelHtml**, **labelPrinted** 以及 **internationalInvoice** 等信息的标签/文档。

为了计费的需要，每个航段（ShipmentPackageRouteSeg）的包裹都有个 **estimatedAmount** 字段用于记录承运商的预计运输报价，加上 **packageTransportAmount**, **packageServiceAmount**, 和 **packageOtherAmount** 记录承运商实际的数字，连同 **codAmount** 和 **insuredAmount** 字段信息一起应用于这种场景。所有的这些都使用 **amountUomId** 来指定币种。

对于航段（例如 ShipmentRouteSegment）上的所有包裹来说，这里有很多的字段用于总计，如 **actualTransportCost**, **actualServiceCost**, **actualOtherCost** 和 **actualCost**，并且使用 **costUomId** 来指定币种。航段同样有一个总计的使用 **billingWeightUomId** 单位的 **billingWeight** 字段，用于记录航段中所有包裹的计费重量信息。航段同样有一个状态字段（**statusId**）用来持续追踪和承运商的联系（通常是集成的方式），包括：未开始，已确认，接收和废弃。

装运 Shipment 通常基于一个或多个订单或者退单，并通常会导致产生一张或者多张发票。装运条目源（ShipmentItemSource）实体用于追踪这些信息，并且 ShipmentItemSource 还有更多关于运输条目（ShipmentItem）的信息记录在其中。运输条目可以关联多个订单条目（**orderId**, **orderItemSeqId**）或者退单条目（**returnId**, **returnItemSeqId**），并且通常关联一个或者多个发票条目（**invoiceId**, **invoiceItemSeqId**）。

ShipmentItemSource.**quantity** 字段用于指定来源于订单或者退单条目的 ShipmentItem.**quantity** 的数量有多少。这里还有个未在手数量（**quantityNotHandled**）字段在源中用于指定应当被装运但实际并未装运的数量。

装运的 **picklistId** 和 **binLocationNumber** 字段，以及装运条目源 ShipmentItemSource 的 **binLocationNumber** 和 **statusId** 字段用于仓库的分拣和打包。参见下面 ***Picklist (mantle.shipment.picklist)***的部分，查询更多的细节。

- ***承运商 (mantle.shipment.carrier)***

承运商特指类似于 UPS 或者 FedEx 这样的公司。CarrierShipmentMethod 实体用于配置承运商（**carrierPartyId**）提供的承运方式（**shipmentMethodEnumId**），承运商服务编号（**carrierServiceCode**）以及承运方式的承运商标准代码（**scaCode**）。

类似的，CarrierShipmentBoxType 实体用于配置承运商的运输装箱类型 ShipmentBoxType（通过 **shipmentBoxTypeId**）记录以及它们关联的 **packagingTypeCode** 及可用的 **oversizeCode**。运输方式和装箱代码都特用于承运商集成去指定服务级别以及承运商提供的装箱型号。

PartyCarrierAccount 用于追踪承运商当事人 Party 的账户信息。

ShippingGatewayConfig 用于指定与承运商集成的细节，一般是为了装运估价，报价，得

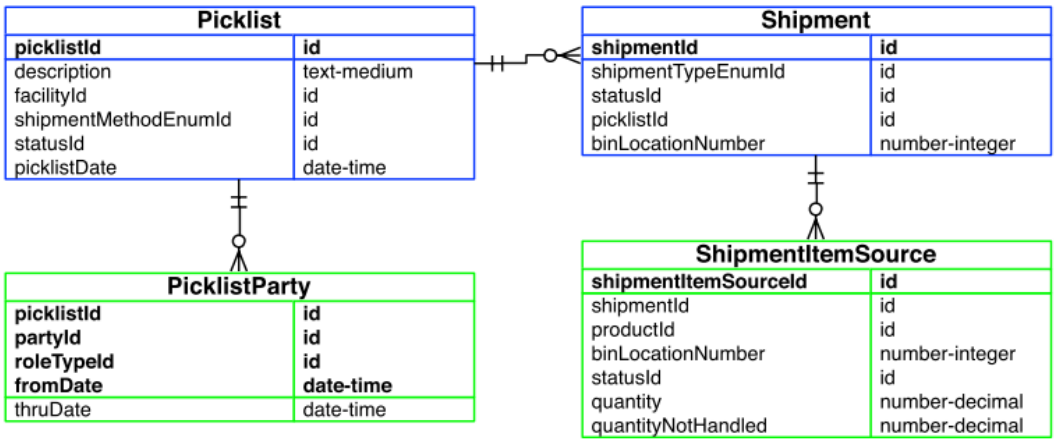
到标签，废弃标签，追踪包裹甚至校验地址等目的。为了实现货运网关（承运商集成），为上述目的去实现服务并创建记录去指向它们，然后通过 ProductStoreShippingGateway 实体去关联商店 ProductStore 与网关。

➢ *分拣清单（mantle.shipment.picklist）*

分拣清单（Picklist）用于组织分拣/打包过程期间的装运（Shipment）记录。分拣清单不像装运（Shipment）一样，它没有拆分结构。同样的，这里没有分拣清单条目，装运条目源（ShipmentItemSource）用于在分拣"容器"中追踪装运条目，并记录装运条目的特定数量来源于关联的订单，退单以及发票的细节。

对于输入或已排程状态的装运 Shipment 来说，Shipment.**picklistId** 字段指向了其所在的分拣清单 Picklist。分拣清单通常关联设施 Facility（**facilityId**）并可能关联特定的装运方式（**shipmentMethodEnumId**）来计划和执行客户发货。为了管理和历史追溯的目的，分拣清单 Picklist 有一个时间字段（**picklistDate**）来记录单据设计时间。PicklistParty 实体用于关联分拣清单和特定角色的当事人如分拣员，包装工，管理者等。

在一个特定的分拣业务过程中，多个装运可同时进行分拣，并将内容装到一个箱子中。可以使用 Shipment.**binLocationNumber** 字段去追踪这种场景，除非装运使用多个箱子进行分割（例如装运中每个定单一个箱子），然后并且使用 ShipmentItemSource.**binLocationNumber** 字段去覆盖装运 Shipment 的这条记录。



装运条目源（ShipmentItemSource）实体通过订单条目 OrderItem（**orderId**, **orderItemSeqId**）的明细记录去找到关联的资产保留（AssetReservation）数据中分拣的数量 **quantity**（或使用的未发行数量 **quantityNotIssued**），同时，通过关联的资产（Asset）信息可以找到待分拣的资产存放的设施地点 FacilityLocation。

装运条目源（ShipmentItemSource）为了分拣和打包的目的有个状态（**statusId**）字段，状态值包括等待，分拣，打包，接收和撤销。注意接收状态在典型的分拣/打包过程之外，用于某些需求场景中追溯运输条目的接收情况。

一个典型的分拣单有一个订单所罗列资产的所有存放设施地点的清单，这样就可以很方便的去分拣单据上需装运的货品。对于每个存放地的要分拣的产品和数量都在单据上罗列了出来，这样就可以选择合适的箱子数量以及为正确的货运的箱子装正确数量的产品。上面一系列的实体可获得所有的细节信息。

工作投入

➢ 工作投入 (mantle.work.effort)

最基本的工作投入 WorkEffort 类型是任务和日历事件。更普遍的 WorkEffort 用于项目，里程碑，任务，制造路线，会议，电话，出差，及甚至是休假和有效工作。

这些通过类型（**workEffortTypeEnumId**）和目的（**purposeEnumId**）进行指定。类型有更多的自动化在它们的周围并有更多的限制，当前包括了项目，里程碑，任务，事件，有效工作和休假。目的则更为灵活，这里有一个更大的集合，并且你可以添加 WorkEffortPurpose 类型的 Enumeration 记录。

工作投入是层级结构的，**rootWorkEffortId** 字段用于定义根（例如一个项目），同时 **parentWorkEffortId** 字段用于指向层级中的直接父亲。举个例子，一个项目类型的工作投入作为根，顶层（第一层）的任务是任务类型的工作投入记录，并使用 **rootWorkEffortId** 字段指向到项目，同时没有 **parentWorkEffortId**。子任务有相同的 **rootWorkEffortId** 数值，但是它们的 **parentWorkEffortId** 字段指向到顶层的任务。

WorkEffort 有一个任务或事件所需要的、所有基本的字段，包括：名称（**workEffortName**），**description**，**location**，**infoUrl**，**estimatedStartDate**，**estimatedCompletionDate**，**percentComplete**，和 **priority**。对于 iCal（iCalendar：互联网日历）文件以及类似文件所使用的 **workEffortId**，这个字段并不是一个全球唯一标识符，所以这里有个 **universalId** 字段用于这种情况。为了历史追踪，工作投入同样还有 **actualStartDate** 和 **actualCompletionDate** 字段。

工作投入可能会在一个办公室，仓库或其他类型的 Facility 里进行，并使用 **facilityId** 字段进行跟踪。对于额外的位置和联系信息来说，使用 WorkEffortContactMech 实体去关联联系机制，例如邮政地址，电话号码（用于电话会议等），电子邮件地址等等。WorkEffortCommEvent 实体用于持续跟踪工作投入关联的实际沟通，并关联 CommunicationEvent 记录。

WorkEffort 可能会是内部的，敏感的或完全开放的，这是通过 **visibilityEnumId** 字段进行指定的。开箱即用的选项包括 General（公开访问），Work Group（仅组内可访问），Restricted（私有访问），以及 Top Secret（机密访问）。

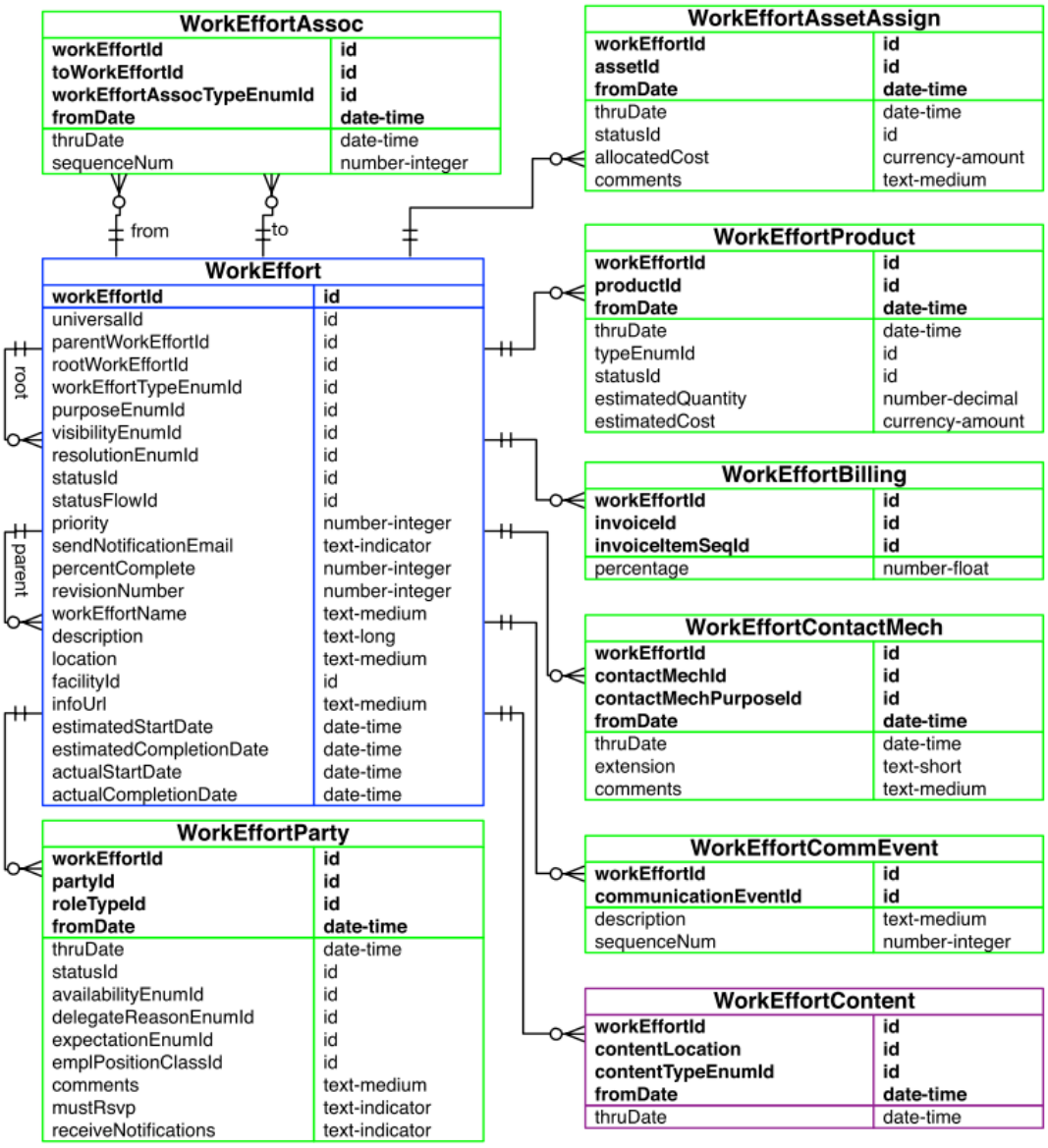
对于某些类型的投入例如制造任务，更细致的时间津贴和跟踪是有必要的。这里有一些用于此处的十进制数字段：**estimatedWorkTime**, **estimatedSetupTime**, **remainingWorkTime**, **actualWorkTime**, **actualSetupTime** 和 **totalTimeAllowed**。这些字段的时间单位在 **timeUomId** 字段中进行指定。

WorkEffort 状态（**statusId**）选项包括：计划中，已批准/已安排，执行中，完成，已关闭，暂停以及撤销。这些状态值是默认 Default 的 StatusFlow。如要使用不同的 StatusFlow，则需将 **statusFlowId** 字段不但用于一个指定的 WorkEffort（根据实现）上，而且应用在 **rootWorkEffortId** 字段指定的其根 WorkEffort 之上。

除了状态之外，WorkEffort 还有个解决状态（**resolutionEnumId**）字段。开箱即用的选项包括未解决的（默认的），已完成，未完成，不会完成，重复，不可复现以及信息不足。额外的解决状态可以通过 WorkEffortResolution 类型的 Enumeration 记录进行添加。

除了层级的结构之外，工作投入可能会使用 WorkEffortAssoc 实体去相互关联，包括的关联类型有依赖（Depends On），复制（Duplicates），归因于（Caused By），不依赖于 Independent Of（并行），路线组件（Routing Component）以及里程碑（Milestone）。注意里程碑并不是作为一个父的 WorkEffort 与任务进行关联的。这是因为一个任务可能随着时间的推移会与多个里程碑发生关联，所以我们有一个历史和长期规划选项。额外的关联类型可以通过

WorkEffortAssocType 类型的 Enumeration 记录进行添加。



WorkEffortAssetAssign 实体用于记录工作投入中使用的（但不是消耗的）设备或其他类型的资产 Asset。Asset 记录分配这种方式通常被认为在工作投入 WorkEffort 期间是繁忙的（否则不可用）。使用 WorkEffortAssetNeeded 实体去为某种类型资产所需要的产品 Product（**assetProductId**）做计划，同时这个产品表现为某个类型的资产。Product 记录可能会因为其他原因关联 WorkEffort，这要使用 WorkEffortProduct 实体。WorkEffortAssetUsed 用于追踪诸如原材料和供应物资的资产在一个工作投入中被使用（消耗），同时工作投入中生产出来的资产使用 WorkEffortAssetProduced 记录。

有时，通过一个更一般的成果 Deliverable 去组织工作投入是十分有用的。使用 WorkEffortDeliverableProd 去关联它和工作投入。

使用 WorkEffortSkillStandard 去记录一个 WorkEffort 中所需要的技能（源自 HR/人力资源实体中 SkillType 类型的 Enumeration 记录），通常为指派到工作投入的当事人选择的一部分。

这里有各种原因去关联一个当事人 Party 和一个工作投入 WorkEffort，而且参与到工作投入中的当事人（就像关联到其他实体的当事人）取决于角色（**roleTypeId**）。这可能为经理，工人，操作员或其他任何角色（包括不适用的 Not Applicable）。由于计费的原因，

WorkEffortParty 可能通过 **emplPositionClassId** 字段被关联到 EmplPositionClass 实体上。

每个当事人 Party 关联 WorkEffort 都有一个状态（**statusId**；Offered，Assigned，Declined，Unassigned），可用性（**availabilityEnumId**；Available，Busy，Away），预期（**expectationEnumId**；For Your Information， Involvement Required， Involvement Requested， Immediate Response Requested）以及这种情况下委托的原因（(**delegateReasonEnumId**；Need Support or Help，My Part Finished， Completely Finished）。

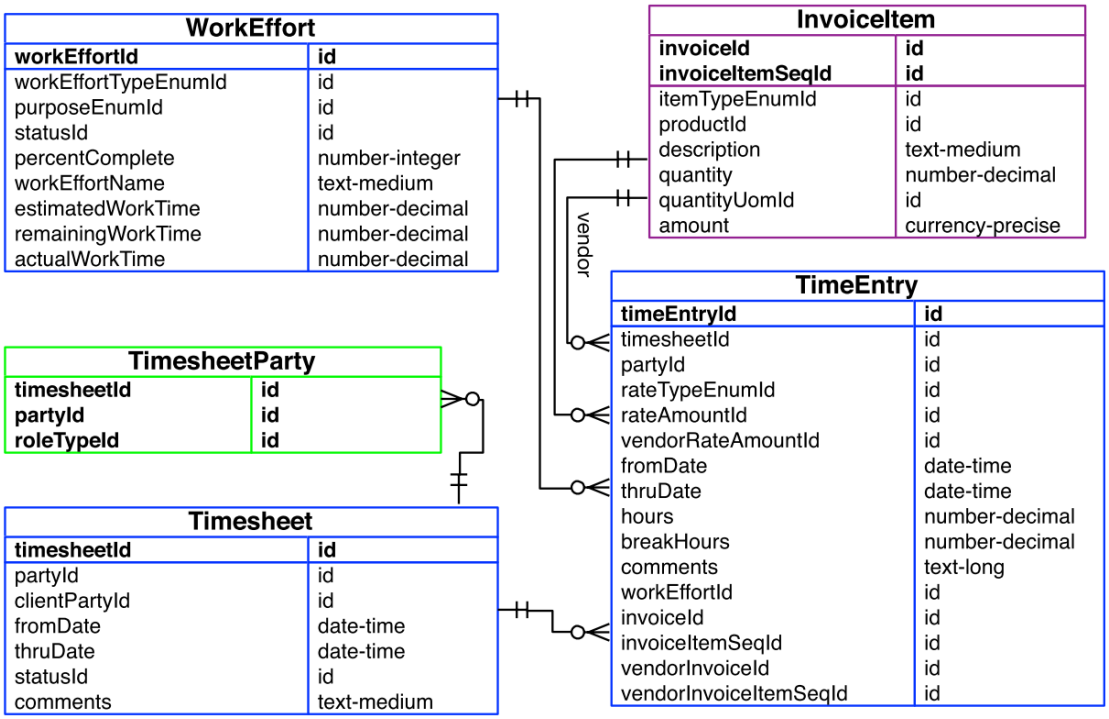
使用 WorkEffortInvoice 实体去关联一个高层次 WorkEffort（如项目）的和一个发票。对于具体任务的更多计费细节或其他低层次的工作投入，抑或是甚至任务的百分比来说，使用 WorkEffortBilling 实体。

通用的资源门面内容以及文档可能使用 WorkEffortContent 实体和 WorkEffort 进行关联。工作的说明被记录在 WorkEffortNote 中。

- ***时间条目 (mantle.work.time)***

TimeEntry 实体用于记录一个任务上花费的的工作时间（**hours**）或一个特定当事人 Party（**partyId**）的其他类型的 WorkEffort（**workEffortId**）。工作的时间落在 **fromDate** 和 **thruDate** 之间，并且如果任何在这个区间内的时间不用于工作，则将其记录在 **breakHours**。一般 **hours** + **breakHours**，如果都指定了，应当匹配在 **fromDate** 和 **thruDate** 之间的时间。

为了计费的目的，RateType 一般使用 **rateTypeId** 字段进行指定。常用的类型包括：标准的，折扣的，超时的，以及现场工作。它可用于查找一个 RateAmount 记录连同其他可应用的数据（可能包含 **partyId**，**workEffortId**，**emplPositionClassId**，和作为客户或供应商的 **ratePurposeEnumId**）一起。这可能需要做两次，一次是为了客户的价格（客户支付给供应商），而另一次是为了供应商的价格（供应商支付给工人），同时它们记录在 **rateAmountId** 和 **vendorRateAmountId** 字段中。



一旦一个 TimeEntry 被计费了，关联的 InvoiceItem 被 **invoiceId** 和 **invoiceItemSeqId** 字段引用用于源自供应商发给客户的发票，同时 **vendorInvoiceId** 和 **vendorInvoiceItemSeqId** 字段

用于源自工人给供应商的发票。当这些字段被填充时，意味着时间条目已经被计费了。

Timesheet 可能用于组织 TimeEntry 记录，或使得时间条目更简单。一般一个工时单有两个当事人关联它，工人当事人 Party（**partyId**）以及客户当事人 Party（**clientPartyId**）。其他当事人可通过 TimesheetParty 进行关联。

一个工时单 Timesheet 通常用于某个具体的时间范围（**fromDate**, **thruDate**）内。在工时单本身的生命周期中有个状态（**statusId**）字段，典型的有 In-Process（工作正在完成，时间正在被记录），Completed（所有关联的工作都已完成并记录了时间），或 Approved（计费已批准）。

USL 业务过程

这部分内容主要是 Mantle 中高等级的商业过程的概述。它是商业过程概念以及每个过程中特定的服务以及实体的一个介绍。这里还有很多的服务以及实体并未涉及到，换句话说这里并未包含完整的，所有可用的服务和操作。这些只是给你一个已存在构建的通用功能的思路，从这里你可以轻松的回顾代码和参考业务构件的关系。

地幔业务构件（Mantle Business Artifacts）有非常多的功能，包括**采购到付款**，**订单到收款**以及**工作计划到收款**的过程，细节如下：

- ✓ 采购和销售订单（包含产品，服务，物料等；面向仓库和设备/供应商的 POS 机等）
- ✓ 项目，任务以及包含时间和费用的需求管理，计费/支付价格（面向项目/任务/客户/工人等）
- ✓ 各种类型的进出发票以及打印的或者邮件的 XSL:FO 模板
- ✓ 自动产生的发票：采购订单（AP），销售订单（AR），项目客户工期和费用（AR），项目供应商/工人工期和费用（AP）
- ✓ 支付，通过支付执行接口的人工和自动执行的记录；支持支付到发票
- ✓ 实现销售订单（包括基本的分拣和装箱）和接收采购订单
- ✓ 仓库管理包括出库和入库，以及销售订单的仓库保留
- ✓ 自动总帐过账和进出发票，进出支付，支付应用以及仓库进出库
- ✓ 通用总帐功能：帐期，验证的交易，期间关闭
- ✓ 资产负债表和损益表报告（基本的期间的过账总数和财务平衡）
- ✓ 产品价格，运费计算以及税金计算的 Drools 规则

采购到付款

这个过程的 Spock 测试套件在 OrderProcureToPayBasicFlow.groovy 文件中。样例的部分关联的安装数据在下面进行描述，你可以在 ZzaGlAccountsDemoData.xml, ZzbOrganizationDemoData.xml, 和 ZzcProductDemoData.xml 文件中找到其余部分的数据。

- ***供应商的产品定价***

这里有一些测试数据去调用外部提供商（供应商）当事人 Party MiddlemanInc（**vendorPartyId**）对内部企业 ORG_BIZI_RETAIL（**customerPartyId**）的产品 DEMO_1_1 的价格，数量测试为 1 和 100：

```
String vendorPartyId = 'MiddlemanInc', customerPartyId = 'ORG_BIZI_RETAIL'
String priceUomId = 'USD', currencyUomId = 'USD'
String facilityId = 'ORG_BIZI_RETAIL_WH'

Map priceMap = ec.service.sync()
    .name("mantle.product.PriceServices.get#ProductPrice")
    .parameters([productId:'DEMO_1_1', priceUomId:priceUomId, quantity:1,
```

```
        vendorPartyId:vendorPartyId,
        customerPartyId:customerPartyId]).call()

Map priceMap2 = ec.service.sync()
    .name("mantle.product.PriceServices.get#ProductPrice")
    .parameters([productId:'DEMO_1_1', priceUomId:priceUomId, quantity:100,
        vendorPartyId:vendorPartyId,
        customerPartyId:customerPartyId]).call()
```

下面是样例的产品（Product）记录以及产品价格（ProductPrice）记录用于配置供应商价格：

```
<mantle.product.Product productId="DEMO_1_1"
    productTypeEnumId="PtFinishedGood" chargeShipping="Y"
    returnable="Y" productName="Demo Product One-One" description=""/>
<mantle.product.ProductPrice productPriceId="DEMO_1_1_CS1"
    productId="DEMO_1_1" vendorPartyId="MiddlemanInc"
    pricePurposeEnumId="PppPurchase" priceTypeEnumId="PptCurrent"
    fromDate="2010-02-03 00:00:00" minQuantity="1" price="9.00"
    priceUomId="USD"/>
<mantle.product.ProductPrice productPriceId="DEMO_1_1_CS100"
    productId="DEMO_1_1" vendorPartyId="MiddlemanInc"
    pricePurposeEnumId="PppPurchase" priceTypeEnumId="PptCurrent"
    fromDate="2010-02-03 00:00:00" minQuantity="100" price="8.00"
    priceUomId="USD"/>
```

确认结果如下，注意数量为 1 时价格为 9.00，数量为 100 时价格为 8.00：

```
priceMap.price == 9.00
priceMap2.price == 8.00
priceMap.priceUomId == 'USD'
```

- ***收货地址和批准采购订单***

对于采购订单来说没有产品仓库，所以我们无需从配置中获取支付，运输，当事人以及其他的设置信息。在调用创建订单服务（create#Order）时，我们可以明确的设置客户和供应商。下面是一个创建采购订单的服务代码片段，它添加了产品条目，运费，计费和运输信息，收货地址以及批准订单。

```
Map orderOut = ec.service.sync()
    .name("mantle.order.OrderServices.create#Order")
    .parameters([customerPartyId:customerPartyId,
        vendorPartyId:vendorPartyId, currencyUomId:currencyUomId]).call()

purchaseOrderId = orderOut.orderId
orderPartSeqId = orderOut.orderPartSeqId

ec.service.sync()
    .name("mantle.order.OrderServices.add#OrderProductQuantity")
    .parameters([orderId:purchaseOrderId, orderPartSeqId:orderPartSeqId,
        productId:'DEMO_1_1', quantity:150,
        itemTypeEnumId:'ItemProduct']).call()
ec.service.sync()
    .name("mantle.order.OrderServices.add#OrderProductQuantity")
    .parameters([orderId:purchaseOrderId, orderPartSeqId:orderPartSeqId,
        productId:'DEMO_3_1', quantity:100,
        itemTypeEnumId:'ItemProduct']).call()
ec.service.sync()
    .name("mantle.order.OrderServices.add#OrderProductQuantity")
    .parameters([orderId:purchaseOrderId, orderPartSeqId:orderPartSeqId,
        productId:'EQUIP_1', quantity:1,
        itemTypeEnumId:'ItemAsset', unitAmount:10000]).call()
// add shipping charge
ec.service.sync()
    .name("mantle.order.OrderServices.create#OrderItem")
    .parameters([orderId:purchaseOrderId, orderPartSeqId:orderPartSeqId,
        unitAmount:145.00, itemTypeEnumId:'ItemShipping',
        itemDescription:'Incoming Freight']).call()
// set billing and shipping info
setInfoOut = ec.service.sync()
    .name("mantle.order.OrderServices.set#OrderBillingShippingInfo")
    .parameters([orderId:purchaseOrderId, orderPartSeqId:orderPartSeqId,
        paymentMethodTypeEnumId:'PmtCompanyCheck',
        shippingPostalContactMechId:'ORG_BIZI_RTL_SA',
        shippingTelecomContactMechId:'ORG_BIZI_RTL_PT',
        shipmentMethodEnumId:'ShMthNoShipping']).call()

// one person will place the PO
ec.service.sync()
    .name("mantle.order.OrderServices.place#Order")
    .parameters([orderId:purchaseOrderId]).call()
// typically another person will approve the PO
ec.service.sync()
    .name("mantle.order.OrderServices.approve#Order")
    .parameters([orderId:purchaseOrderId]).call()
```

一旦这个过程完成，采购订单（PO）会以某种方式发给供应商（vendor）。下面是订单创建的实体 XML 文件。注意子订单（OrderPart）的大量细节信息包括供应商和客户当事人，支付和运输信息等等。同样注意，系统用户（ec.user）设置的有效时间（**effectiveTime**）作为一个有效的线性时间（代码运行前执行）：

```
long effectiveTime = System.currentTimeMillis()
ec.user.setEffectiveTime(new Timestamp(effectiveTime))
```

下面是订单的 XML 文件：

```
<mantle.order.OrderHeader orderId="${purchaseOrderId}"
    entryDate="${effectiveTime}" placedDate="${effectiveTime}"
    statusId="OrderApproved" currencyUomId="USD" grandTotal="11795.00"/>
<mantle.order.OrderPart orderId="${purchaseOrderId}" orderPartSeqId="01"
    vendorPartyId="MiddlemanInc" customerPartyId="ORG_BIZI_RETAIL"
    shipmentMethodEnumId="ShMthNoShipping"
    postalContactMechId="ORG_BIZI_RTL_SA"
    telecomContactMechId="ORG_BIZI_RTL_PT" partTotal="11795.00"/>
<mantle.account.payment.Payment paymentId="${setInfoOut.paymentId}"
    paymentMethodTypeEnumId="PmtCompanyCheck" orderId="${purchaseOrderId}"
    orderPartSeqId="01" statusId="PmntPromised" amount="11795.00"
    amountUomId="USD"/>

<mantle.order.OrderItem orderId="${purchaseOrderId}" orderItemSeqId="01"
    orderPartSeqId="01" itemTypeEnumId="ItemProduct" productId="DEMO_1_1"
    itemDescription="Demo Product One-One" quantity="150" unitAmount="8.00"
    isModifiedPrice="N"/>
<mantle.order.OrderItem orderId="${purchaseOrderId}" orderItemSeqId="02"
    orderPartSeqId="01" itemTypeEnumId="ItemProduct" productId="DEMO_3_1"
    itemDescription="Demo Product Three-One" quantity="100"
    unitAmount="4.50" isModifiedPrice="N"/>
<mantle.order.OrderItem orderId="${purchaseOrderId}" orderItemSeqId="03"
    orderPartSeqId="01" itemTypeEnumId="ItemAsset" productId="EQUIP_1"
    itemDescription="Picker Bot 2000" quantity="1" unitAmount="10000"
    isModifiedPrice="Y"/>
<mantle.order.OrderItem orderId="${purchaseOrderId}" orderItemSeqId="04"
    orderPartSeqId="01" itemTypeEnumId="ItemShipping"
    itemDescription="Incoming Freight" quantity="1" unitAmount="145.00"/>
```

- ***创建进入配送和购买发票***

下面的代码为子订单 OrderPart 创建发货运输 Shipment（这里只有一个子订单，所以我们只需创建一个），然后标记这个装运 Shipment 为已发货，接着为这个完整的子订单 OrderPart 创建一个发票 Invoice。

在真实世界的场景中，接收的发票可能并不和期望的相匹配，或者可能是多个采购单抑或是采购单的一部分。在这个例子中我们简单的自动创建一个源于订单的发票，并某种形式上接近真实世界的场景。在实际的业务过程中，我们更多的创建一个 InvoiceIncoming 状态的发票 Invoice，然后允许供应商手动的改变发票文本的状态为 InvoiceReceived。

```
shipResult = ec.service.sync()
    .name("mantle.shipment.ShipmentServices.create#OrderPartShipment")
    .parameters([orderId:purchaseOrderId, orderPartSeqId:orderPartSeqId,
        destinationFacilityId:facilityId]).call()
ec.service.sync()
    .name("mantle.shipment.ShipmentServices.ship#Shipment")
    .parameters([shipmentId:shipResult.shipmentId]).call()
invResult = ec.service.sync()
    .name("mantle.account.InvoiceServices.create#EntireOrderPartInvoice")
    .parameters([orderId:purchaseOrderId, orderPartSeqId:orderPartSeqId,
        statusId:'InvoiceReceived']).call()
```

创建的装运 Shipment 记录如下：

```
<mantle.shipment.Shipment shipmentId="${shipResult.shipmentId}"
    shipmentTypeEnumId="ShpTpPurchase" statusId="ShipInput"
    fromPartyId="MiddlemanInc" toPartyId="ORG_BIZI_RETAIL"/>
<mantle.shipment.ShipmentPackage shipmentId="${shipResult.shipmentId}"
    shipmentPackageSeqId="01"/>
<mantle.shipment.ShipmentRouteSegment shipmentId="${shipResult.shipmentId}"
    shipmentRouteSegmentSeqId="01"
    destPostalContactMechId="ORG_BIZI_RTL_SA"
    destTelecomContactMechId="ORG_BIZI_RTL_PT"/>
<mantle.shipment.ShipmentPackageRouteSeg
    shipmentId="${shipResult.shipmentId}" shipmentPackageSeqId="01"
    shipmentRouteSegmentSeqId="01"/>

<mantle.shipment.ShipmentItem shipmentId="${shipResult.shipmentId}"
    productId="DEMO_1_1" quantity="150"/>
<mantle.shipment.ShipmentItemSource shipmentItemSourceId="55400"
    shipmentId="${shipResult.shipmentId}" productId="DEMO_1_1"
    orderId="${purchaseOrderId}" orderItemSeqId="01" statusId="SisPending"
    quantity="150" quantityNotHandled="150" invoiceId=""
    invoiceItemSeqId=""/>

<mantle.shipment.ShipmentItem shipmentId="${shipResult.shipmentId}"
    productId="DEMO_3_1" quantity="100"/>

<mantle.shipment.ShipmentItemSource shipmentItemSourceId="55401"
    shipmentId="${shipResult.shipmentId}" productId="DEMO_3_1"
    orderId="${purchaseOrderId}" orderItemSeqId="02" statusId="SisPending"
    quantity="100" quantityNotHandled="100" invoiceId=""
    invoiceItemSeqId=""/>

<mantle.shipment.ShipmentItem shipmentId="${shipResult.shipmentId}"
    productId="EQUIP_1" quantity="1"/>
<mantle.shipment.ShipmentItemSource shipmentItemSourceId="55402"
    shipmentId="${shipResult.shipmentId}" productId="EQUIP_1"
    orderId="${purchaseOrderId}" orderItemSeqId="03" statusId="SisPending"
    quantity="1" quantityNotHandled="1" invoiceId="" invoiceItemSeqId=""/>
```

ship#Shipment 服务之后调用的发货记录如下：

```
<mantle.shipment.Shipment shipmentId="${shipResult.shipmentId}"
    shipmentTypeEnumId="ShpTpPurchase" statusId="ShipShipped"
    fromPartyId="MiddlemanInc" toPartyId="ORG_BIZI_RETAIL"/>
```

下面是发票 Invoice 的 XML 格式。注意每条发票条目 InvoiceItem 都有一个相对应的 OrderItemBilling 记录去关联所基于的订单条目 OrderItem。

```
<!-- Invoice created and received, not yet approved/etc -->
<mantle.account.invoice.Invoice invoiceId="${invResult.invoiceId}"
    invoiceTypeEnumId="InvoiceSales" fromPartyId="MiddlemanInc"
    toPartyId="ORG_BIZI_RETAIL" statusId="InvoiceReceived"
    invoiceDate="${effectiveTime}"
    description="Invoice for Order ${purchaseOrderId} part 01"
    currencyUomId="USD"/>

<mantle.account.invoice.InvoiceItem invoiceId="${invResult.invoiceId}"
    invoiceItemSeqId="01" itemTypeEnumId="ItemProduct" productId="DEMO_1_1"
    quantity="150" amount="8.00" description="Demo Product One-One"
    itemDate="${effectiveTime}"/>
<mantle.order.OrderItemBilling orderItemBillingId="55400"
    orderId="${purchaseOrderId}" orderItemSeqId="01"
    invoiceId="${invResult.invoiceId}" invoiceItemSeqId="01" quantity="150"
    amount="8.00" shipmentId="${shipResult.shipmentId}"/>

<mantle.account.invoice.InvoiceItem invoiceId="${invResult.invoiceId}"
    invoiceItemSeqId="02" itemTypeEnumId="ItemProduct" productId="DEMO_3_1"
    quantity="100" amount="4.50" description="Demo Product Three-One"
    itemDate="${effectiveTime}"/>
<mantle.order.OrderItemBilling orderItemBillingId="55401"
    orderId="${purchaseOrderId}" orderItemSeqId="02"
    invoiceId="${invResult.invoiceId}" invoiceItemSeqId="02" quantity="100"
    amount="4.50" shipmentId="${shipResult.shipmentId}"/>

<mantle.account.invoice.InvoiceItem invoiceId="${invResult.invoiceId}"
    invoiceItemSeqId="03" itemTypeEnumId="ItemAsset" productId="EQUIP_1"
    quantity="1" amount="10,000" description="Picker Bot 2000"

    itemDate="${effectiveTime}"/>
<mantle.order.OrderItemBilling orderItemBillingId="55402"
    orderId="${purchaseOrderId}" orderItemSeqId="03"
    invoiceId="${invResult.invoiceId}" invoiceItemSeqId="03" quantity="1"
    amount="10,000" shipmentId="${shipResult.shipmentId}"/>

<mantle.account.invoice.InvoiceItem invoiceId="${invResult.invoiceId}"
    invoiceItemSeqId="04" itemTypeEnumId="ItemShipping" quantity="1"
    amount="145" description="Incoming Freight"
    itemDate="${effectiveTime}"/>
<mantle.order.OrderItemBilling orderItemBillingId="55403"
    orderId="${purchaseOrderId}" orderItemSeqId="04"
    invoiceId="${invResult.invoiceId}" invoiceItemSeqId="04" quantity="1"
    amount="145"/>

<!-- ShipmentItemSource now has invoiceId and invoiceItemSeqId -->
<mantle.shipment.ShipmentItemSource shipmentItemSourceId="55400"
    invoiceId="${invResult.invoiceId}" invoiceItemSeqId="01"/>
<mantle.shipment.ShipmentItemSource shipmentItemSourceId="55401"
    invoiceId="${invResult.invoiceId}" invoiceItemSeqId="02"/>
<mantle.shipment.ShipmentItemSource shipmentItemSourceId="55402"
    invoiceId="${invResult.invoiceId}" invoiceItemSeqId="03"/>
```

- ***接货***

这里有一个 **receive#EntireShipment** 接收所有货物的服务，但这个案例中我们想要接收其中的一个条目去展示如何指定更多的细节，并且处理设备类型的产品时，通知系统是设备产品而不是库存产品(**assetTypeEnumId**=AstTpEquipment)，并且记录设备编号（**serialNumber**）。

```
ec.service.sync()
    .name("mantle.shipment.ShipmentServices.receive#ShipmentProduct")
    .parameters([shipmentId:shipResult.shipmentId, productId:'DEMO_1_1',
        quantityAccepted:150, facilityId:facilityId]).call()
ec.service.sync()
    .name("mantle.shipment.ShipmentServices.receive#ShipmentProduct")
    .parameters([shipmentId:shipResult.shipmentId, productId:'DEMO_3_1',
        quantityAccepted:100, facilityId:facilityId]).call()
ec.service.sync()
    .name("mantle.shipment.ShipmentServices.receive#ShipmentProduct")
    .parameters([shipmentId:shipResult.shipmentId, productId:'EQUIP_1',
        quantityAccepted:1, facilityId:facilityId,
        serialNumber:'PB2000AZQRTFP',
        assetTypeEnumId:'AstTpEquipment']).call()
```

产品有相当多的数据，包括：资产 Asset 记录，资产接收 AssetReceipt 记录（展示库存和设备接收）以及资产明细 AssetDetail 记录去展示资产 Asset 记录的数量变化以及数量变化的原因：

```
<mantle.product.asset.Asset assetId="55400"
    assetTypeEnumId="AstTpInventory" statusId="AstAvailable"
    ownerPartyId="ORG_BIZI_RETAIL" productId="DEMO_1_1" hasQuantity="Y"
    quantityOnHandTotal="150" availableToPromiseTotal="150"
    assetName="Demo Product One-One" receivedDate="${effectiveTime}"
    acquiredDate="${effectiveTime}" facilityId="ORG_BIZI_RETAIL_WH"
    acquireOrderId="${purchaseOrderId}" acquireOrderItemSeqId="01"
    acquireCost="8" acquireCostUomId="USD"/>
<mantle.product.receipt.AssetReceipt assetReceiptId="55400" assetId="55400"
    productId="DEMO_1_1" orderId="${purchaseOrderId}" orderItemSeqId="01"
    shipmentId="${shipResult.shipmentId}" receivedByUserId="EX_JOHN_DOE"
    receivedDate="${effectiveTime}" quantityAccepted="150"/>
<mantle.product.asset.AssetDetail assetDetailId="55400" assetId="55400"
    effectiveDate="${effectiveTime}" quantityOnHandDiff="150"
    availableToPromiseDiff="150" unitCost="8"
    shipmentId="${shipResult.shipmentId}" productId="DEMO_1_1"
    assetReceiptId="55400"/>

<mantle.product.asset.Asset assetId="55401"
    assetTypeEnumId="AstTpInventory" statusId="AstAvailable"
    ownerPartyId="ORG_BIZI_RETAIL" productId="DEMO_3_1" hasQuantity="Y"
    quantityOnHandTotal="100" availableToPromiseTotal="100"
    assetName="Demo Product Three-One" receivedDate="${effectiveTime}"
    acquiredDate="${effectiveTime}" facilityId="ORG_BIZI_RETAIL_WH"
    acquireOrderId="${purchaseOrderId}" acquireOrderItemSeqId="02"
    acquireCost="4.5" acquireCostUomId="USD"/>
<mantle.product.receipt.AssetReceipt assetReceiptId="55401" assetId="55401"
    productId="DEMO_3_1" orderId="${purchaseOrderId}" orderItemSeqId="02"
    shipmentId="${shipResult.shipmentId}" receivedByUserId="EX_JOHN_DOE"
    receivedDate="${effectiveTime}" quantityAccepted="100"/>
<mantle.product.asset.AssetDetail assetDetailId="55401" assetId="55401"
    effectiveDate="${effectiveTime}" quantityOnHandDiff="100"
    availableToPromiseDiff="100" unitCost="4.5"
    shipmentId="${shipResult.shipmentId}" productId="DEMO_3_1"
    assetReceiptId="55401"/>
```

```
<mantle.product.asset.Asset assetId="55402"
    assetTypeEnumId="AstTpEquipment" statusId="AstInStorage"
    ownerPartyId="ORG_BIZI_RETAIL" productId="EQUIP_1" hasQuantity="N"
    quantityOnHandTotal="1" availableToPromiseTotal="0"
    assetName="Picker Bot 2000" serialNumber="PB2000AZQRTFP"
    receivedDate="${effectiveTime}" acquiredDate="${effectiveTime}"
    facilityId="ORG_BIZI_RETAIL_WH" acquireOrderId="${purchaseOrderId}"
    acquireOrderItemSeqId="03" acquireCost="10,000"
    acquireCostUomId="USD"/>
<mantle.product.receipt.AssetReceipt assetReceiptId="55402" assetId="55402"
    productId="EQUIP_1" orderId="${purchaseOrderId}" orderItemSeqId="03"
    shipmentId="${shipResult.shipmentId}" receivedByUserId="EX_JOHN_DOE"
    receivedDate="${effectiveTime}" quantityAccepted="1"/>
<mantle.product.asset.AssetDetail assetDetailId="55402" assetId="55402"


    effectiveDate="${effectiveTime}" quantityOnHandDiff="1"
    availableToPromiseDiff="0" unitCost="10,000"
    shipmentId="${shipResult.shipmentId}" productId="EQUIP_1"
    assetReceiptId="55402"/>
```

当记录存在的时候，其他另外两个实体自动更新：OrderItemBilling 实体有 **assetReceiptId** 字段信息需变化， ShipmentItemSource 实体需设置 **quantityNotHandled**="0"以及状态字段（**statusId**）设置为 SisReceived。

```
<mantle.order.OrderItemBilling orderItemBillingId="55400"
    assetReceiptId="55400"/>
<mantle.order.OrderItemBilling orderItemBillingId="55401"
    assetReceiptId="55401"/>
<mantle.order.OrderItemBilling orderItemBillingId="55402"
    assetReceiptId="55402"/>
<mantle.shipment.ShipmentItemSource shipmentItemSourceId="55400"
    statusId="SisReceived" quantity="150" quantityNotHandled="0"/>
<mantle.shipment.ShipmentItemSource shipmentItemSourceId="55401"
    statusId="SisReceived" quantity="100" quantityNotHandled="0"/>
<mantle.shipment.ShipmentItemSource shipmentItemSourceId="55402"
    statusId="SisReceived" quantity="1" quantityNotHandled="0"/>
```

库存入库同样会触发会计业务去平衡销售成本和仓库财务的科目；

```
<mantle.ledger.transaction.AcctgTrans acctgTransId="55400"
    acctgTransTypeEnumId="AttInventoryReceipt"
    organizationPartyId="ORG_BIZI_RETAIL"
    transactionDate="${effectiveTime}" isPosted="Y"
    postedDate="${effectiveTime}" glFiscalTypeEnumId="GLFT_ACTUAL"
    amountUomId="USD" assetId="55400" assetReceiptId="55400"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55400"
    acctgTransEntrySeqId="01" debitCreditFlag="C" amount="1,200"
    glAccountTypeEnumId="COGS_ACCOUNT" glAccountId="501000"
    reconcileStatusId="AES_NOT_RECONCILED" isSummary="N"
    productId="DEMO_1_1"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55400"
    acctgTransEntrySeqId="02" debitCreditFlag="D" amount="1,200"
    glAccountTypeEnumId="INVENTORY_ACCOUNT" glAccountId="140000"
    reconcileStatusId="AES_NOT_RECONCILED" isSummary="N"
    productId="DEMO_1_1"/>

<mantle.ledger.transaction.AcctgTrans acctgTransId="55401"
    acctgTransTypeEnumId="AttInventoryReceipt"
    organizationPartyId="ORG_BIZI_RETAIL"
    transactionDate="${effectiveTime}" isPosted="Y"
    postedDate="${effectiveTime}" glFiscalTypeEnumId="GLFT_ACTUAL"
    amountUomId="USD" assetId="55401" assetReceiptId="55401"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55401"
    acctgTransEntrySeqId="01" debitCreditFlag="C" amount="450"
    glAccountTypeEnumId="COGS_ACCOUNT" glAccountId="501000"
```

```
    reconcileStatusId="AES_NOT_RECONCILED" isSummary="N"
    productId="DEMO_3_1"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55401"
    acctgTransEntrySeqId="02" debitCreditFlag="D" amount="450"
    glAccountTypeEnumId="INVENTORY_ACCOUNT" glAccountId="140000"
    reconcileStatusId="AES_NOT_RECONCILED" isSummary="N"
    productId="DEMO_3_1"/>
```

紧接着，我们整理一下订单和发货，记录发货已完成并且子订单已完成：

```
ec.service.sync().name("update#mantle.shipment.Shipment")
    .parameters([shipmentId:shipResult.shipmentId,
        statusId:'ShipDelivered']).call()
ec.service.sync().name("mantle.order.OrderServices.complete#OrderPart")
    .parameters([orderId:purchaseOrderId, orderPartSeqId:orderPartSeqId])
    .call()
```

因为这里订单只有一个子订单，所以头的状态字段也同样更新。数据显示了更新后的收货状态：

```
<mantle.shipment.Shipment shipmentId="${shipResult.shipmentId}"
    statusId="ShipDelivered"/>
<mantle.order.OrderHeader orderId="${purchaseOrderId}"
    statusId="OrderCompleted"/>
```

- ***采购发票核准以及发送付款***

现在装运 Shipment 已经被接收了，是时候发票 Invoice 核准并进行支付了。这里有个服务调用去执行，注意这种隐式的自动实体服务去改变状态的模式（这是所有状态改变如何通过 SECA 规则去完成并保持一致性）：

```
ec.service.sync().name("update#mantle.account.invoice.Invoice")
    .parameters([invoiceId:invResult.invoiceId, statusId:'InvoiceApproved'])
    .call()
```

下面是更新的发票 Invoice 记录：

```
<mantle.account.invoice.Invoice invoiceId="${invResult.invoiceId}"
    statusId="InvoiceApproved"/>
```

当一个发票 Invoice 进入到 InvoiceApproved 状态时，它将触发这张发票的会计业务过账。下面的 XML 有 AcctgTrans 记录以及相应的 AcctgTransEntry 记录，一个是为了每条发票条目 InvoiceItem，并且最后一个（流水编号 05）是为了平衡总帐 GlAccount 应付帐款记录编号为 210000 的科目。

```
<mantle.ledger.transaction.AcctgTrans acctgTransId="55402"
    acctgTransTypeEnumId="AttPurchaseInvoice"
    organizationPartyId="ORG_BIZI_RETAIL"
    transactionDate="${effectiveTime}" isPosted="Y"

     postedDate="${effectiveTime}" glFiscalTypeEnumId="GLFT_ACTUAL"
     amountUomId="USD" otherPartyId="MiddlemanInc"
     invoiceId="${invResult.invoiceId}"/>
 <mantle.ledger.transaction.AcctgTransEntry acctgTransId="55402"
     acctgTransEntrySeqId="01" debitCreditFlag="D" amount="1200"
     glAccountId="501000" reconcileStatusId="AES_NOT_RECONCILED"
     isSummary="N" productId="DEMO_1_1" invoiceItemSeqId="01"/>
 <mantle.ledger.transaction.AcctgTransEntry acctgTransId="55402"
     acctgTransEntrySeqId="02" debitCreditFlag="D" amount="450"
     glAccountId="501000" reconcileStatusId="AES_NOT_RECONCILED"
     isSummary="N" productId="DEMO_3_1" invoiceItemSeqId="02"/>
 <mantle.ledger.transaction.AcctgTransEntry acctgTransId="55402"
     acctgTransEntrySeqId="03" debitCreditFlag="D" amount="10,000"
     glAccountTypeEnumId="FIXED_ASSET" glAccountId="171000"
     reconcileStatusId="AES_NOT_RECONCILED" isSummary="N"
     productId="EQUIP_1" invoiceItemSeqId="03"/>
 <mantle.ledger.transaction.AcctgTransEntry acctgTransId="55402"
     acctgTransEntrySeqId="04" debitCreditFlag="D" amount="145"
     glAccountTypeEnumId="" glAccountId="509000"
     reconcileStatusId="AES_NOT_RECONCILED" isSummary="N"
     invoiceItemSeqId="04"/>
 <mantle.ledger.transaction.AcctgTransEntry acctgTransId="55402"
     acctgTransEntrySeqId="05" debitCreditFlag="C" amount="11795"
     glAccountTypeEnumId="ACCOUNTS_PAYABLE" glAccountId="210000"
     reconcileStatusId="AES_NOT_RECONCILED" isSummary="N"/>
```

上面订单创建的支付 Payment 作为承诺支付（参见 ***收货地址和批准采购订单*** 部分）。现在我们调用服务去标记承诺支付已被发送。这个服务同时会申请支付 Payment 的发票 Invoice，创建条支付应用（PaymentApplication）的记录。一旦一个采购发票 Invoice 的支付 Payment 被应用，它的状态就变为支付被发送（InvoicePmtSent）。

```
sendPmtResult = ec.service.sync()
    .name("mantle.account.PaymentServices.send#PromisedPayment")
    .parameters([invoiceId:invResult.invoiceId,
        paymentId:setInfoOut.paymentId]).call()
```

这里有个支付应用 PaymentApplication 刚被创建，支付 Payment 记录的状态为 PmntDelivered 并且 **effectiveDate** 字段被设置了，同时发票 Invoice 更新成为了 InvoicePmtSent 状态：

```
 <mantle.account.payment.PaymentApplication
     paymentApplicationId="${sendPmtResult.paymentApplicationId}"
     paymentId="${setInfoOut.paymentId}" invoiceId="${invResult.invoiceId}"
     amountApplied="11795.00" appliedDate="${effectiveTime}"/>
 <mantle.account.payment.Payment paymentId="${setInfoOut.paymentId}"
     statusId="PmntDelivered" effectiveDate="${effectiveTime}"/>
 <mantle.account.invoice.Invoice invoiceId="${invResult.invoiceId}"
     statusId="InvoicePmtSent"/>
```

支付 Payment 状态变为已交货（Delivered）时触发总帐过账。由于检查接收到了并且配置的自动过账设置，支付 Payment 来源于（归功于）普通支票总帐科目（111100）：

```
<mantle.ledger.transaction.AcctgTrans acctgTransId="55403"
    acctgTransTypeEnumId="AttOutgoingPayment"
    organizationPartyId="ORG_BIZI_RETAIL"
    transactionDate="${effectiveTime}" isPosted="Y"
    postedDate="${effectiveTime}" glFiscalTypeEnumId="GLFT_ACTUAL"
    amountUomId="USD" otherPartyId="MiddlemanInc"
    paymentId="${setInfoOut.paymentId}"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55403"
    acctgTransEntrySeqId="01" debitCreditFlag="D" amount="11795"
    glAccountId="216000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55403"
    acctgTransEntrySeqId="02" debitCreditFlag="C" amount="11795"
    glAccountId="111100" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N"/>
```

由于支付 Payment 被发送（从更新状态）在其被应用到发票 Invoice 之前，支付被发送到未实施的应付帐款科目中（216000）。当支付应用 PaymentApplication 被创建，将触发另一个过账为支付应用 PaymentApplication 贷出到未实施的支付科目中并且从主的应付账款科目（210000）中借出。下面是交易事务：

```
<mantle.ledger.transaction.AcctgTrans acctgTransId="55404"
    acctgTransTypeEnumId="AttOutgoingPaymentAp"
    organizationPartyId="ORG_BIZI_RETAIL"
    transactionDate="${effectiveTime}" isPosted="Y"
    postedDate="${effectiveTime}" glFiscalTypeEnumId="GLFT_ACTUAL"
    amountUomId="USD" otherPartyId="MiddlemanInc"
    paymentId="${setInfoOut.paymentId}"
    paymentApplicationId="${sendPmtResult.paymentApplicationId}"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55404"
    acctgTransEntrySeqId="01" debitCreditFlag="D" amount="11795"
    glAccountId="210000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55404"
    acctgTransEntrySeqId="02" debitCreditFlag="C" amount="11795"
    glAccountId="216000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N"/>
```

当资产接收到了，发票支付了并且所有的事情都被总帐过账了，采购到付款的业务过程就完成了。总账过账的网络效应就是出去的支付是对普通支票总帐科目（111100）记入贷方，存货采购的价格是对存货资产科目（140000）记入借方，设备价格是对设备资产科目（171000）记入借方，运费记入在销售运费科目（509000）上。条目间彼此互相过账平衡核销直到零，包括应付账款科目（210000）。

订单到收款

这个业务过程的 Spock 测试套件位于 OrderToCashBasicFlow.groovy 文件中，并且这个文件涵盖了这部分的业务。在 OrderTenantAccess.groovy 文件中包含了关联的测试套件用于租户的销售订单描述和预制金额，同时 OrderToCashTime.groovy 文件中设置了测试销售订单的时间。

下面的文件包含了案例中其他关联的设置数据，你可以在其中找到其余的部分：ZzaGlAccountsDemoData.xml, ZzbOrganizationDemoData.xml 以及 ZzcProductDemoData.xml 文件。

- ***客户的销售订单***

这是以个基本的电子商务过程。这个订单是作为客户（joe@public.com）所有的，所以下面代码的第一步是登录这个用户并且最后一步是登出这个用户，然后一个内部的用户处理出货，触发自动支付过程等等。

下面的代码使用 POPC_DEFAULT 标志的产品商店 ProductStore 例子，使用 ORG_BIZI_RETAIL_WH 设施 Facility 作为库存，一个测试的支付以及本地的税金和运费计算的服务。下面是定义的记录（代码来源于 ZzcProductDemoData.xml 文件）：

```
<mantle.facility.Facility facilityId="ORG_BIZI_RETAIL_WH"
    facilityTypeEnumId="FcTpWarehouse" ownerPartyId="ORG_BIZI_RETAIL"
    facilityName="Biziwork Retail Warehouse"/>
<mantle.product.store.ProductStore productStoreId="POPC_DEFAULT"
    storeName="Biziwork Retail Store" organizationPartyId="ORG_BIZI_RETAIL"
    inventoryFacilityId="ORG_BIZI_RETAIL_WH"
    reservationOrderEnumId="AsResOrdFifoRec" requirementMethodEnumId=""
    defaultLocale="en_US" defaultCurrencyUomId="USD"
    taxGatewayConfigId="LOCAL"/>
<mantle.product.store.ProductStorePaymentGateway
    productStoreId="POPC_DEFAULT" paymentMethodTypeEnumId="PmtCreditCard"
    paymentGatewayConfigId="TEST_APPROVE"/>
<mantle.product.store.ProductStoreShippingGateway
    productStoreId="POPC_DEFAULT" carrierPartyId="_NA_"
    shippingGatewayConfigId="NA_LOCAL"/>
```

下面代码中的 **get#ProductPrice** 服务由于获得（计算）产品的价格并自己调用来进行演示。当添加订单项时，它自己将调用服务去获得价格。

注意，第一次调用 **add#OrderProductQuantity** 服务会导致创建一条新的订单记录，所以我们从服务的结果中获得“购物车”的订单编号（orderId）。随后的服务调用都会传入一个订单编号（orderId），这样就可以按照相同的顺序添加产品的数量了。

然后，它将调用 **set#OrderBillingShippingInfo** 服务，通过一个支付方法和客户资料中的联系机制，去设置订单的计费和运输信息。最终，它将调用客户最终反复确认的订单后发生的 **place#Order** 服务。

```
ec.user.loginUser("joe@public.com", "moqui", null)
long effectiveTime = System.currentTimeMillis()
ec.user.setEffectiveTime(new Timestamp(effectiveTime))

String productStoreId = "POPC_DEFAULT"
EntityValue productStore =
    ec.entity.makeFind("mantle.product.store.ProductStore")
        .condition("productStoreId", productStoreId).one()
String currencyUomId = productStore.defaultCurrencyUomId
String priceUomId = productStore.defaultCurrencyUomId
String vendorPartyId = productStore.organizationPartyId
String customerPartyId = ec.user.userAccount.partyId

Map priceMap = ec.service.sync()
    .name("mantle.product.PriceServices.get#ProductPrice")
    .parameters([productId:'DEMO_1_1', priceUomId:priceUomId,
        productStoreId:productStoreId, vendorPartyId:vendorPartyId,
        customerPartyId:customerPartyId]).call()

Map addOut1 = ec.service.sync()
    .name("mantle.order.OrderServices.add#OrderProductQuantity")
    .parameters([productId:'DEMO_1_1', quantity:1,
        customerPartyId:customerPartyId, currencyUomId:currencyUomId,
        productStoreId:productStoreId]).call()

cartOrderId = addOut1.orderId
orderPartSeqId = addOut1.orderPartSeqId

ec.service.sync()
    .name("mantle.order.OrderServices.add#OrderProductQuantity")
    .parameters([orderId:cartOrderId, productId:'DEMO_3_1', quantity:5,
        customerPartyId:customerPartyId, currencyUomId:currencyUomId,
        productStoreId:productStoreId]).call()
ec.service.sync()
    .name("mantle.order.OrderServices.add#OrderProductQuantity")
    .parameters([orderId:cartOrderId, orderPartSeqId:orderPartSeqId,
        productId:'DEMO_2_1', quantity:7, customerPartyId:customerPartyId,
        currencyUomId:currencyUomId, productStoreId:productStoreId])
    .call()

setInfoOut = ec.service.sync()
    .name("mantle.order.OrderServices.set#OrderBillingShippingInfo")
    .parameters([orderId:cartOrderId, paymentMethodId:'CustJqpCc',
        shippingPostalContactMechId:'CustJqpAddr',
        shippingTelecomContactMechId:'CustJqpTeln', carrierPartyId:'_NA_',
        shipmentMethodEnumId:'ShMthGround']).call()
ec.service.sync().name("mantle.order.OrderServices.place#Order")
    .parameters([orderId:cartOrderId]).call()

ec.user.logoutUser()
```

place#Order 服务会触发支付认证，认证将会更新订单状态为核准（Approved）即为此刻的状态。我们同样有个计费设置 **set#OrderBillingShippingInfo** 的支付 Payment 记录，其余的是子订单 OrderPart 的记录。信用卡认证还有个 PaymentGatewayResponse 支付网关响应记录。为了完成它我们有三条订单条目记录 OrderItem，每条都调用了 **add#OrderProductQuantity**，添加了不同的产品（**productId**）。

```
<mantle.order.OrderHeader orderId="${cartOrderId}"
    entryDate="${effectiveTime}" placedDate="${effectiveTime}"
    statusId="OrderApproved" currencyUomId="USD"
    productStoreId="POPC_DEFAULT" grandTotal="145.68"/>
<mantle.account.payment.Payment paymentId="${setInfoOut.paymentId}"
    paymentTypeEnumId="PtInvoicePayment" paymentMethodId="CustJqpCc"
    paymentMethodTypeEnumId="PmtCreditCard" orderId="${cartOrderId}"
    orderPartSeqId="01" statusId="PmntAuthorized" amount="145.68"
    amountUomId="USD" fromPartyId="CustJqp" toPartyId="ORG_BIZI_RETAIL"/>
<mantle.account.method.PaymentGatewayResponse
    paymentGatewayResponseId="55500" paymentOperationEnumId="PgoAuthorize"
    paymentId="${setInfoOut.paymentId}" paymentMethodId="CustJqpCc"
    amount="145.68" amountUomId="USD" transactionDate="${effectiveTime}"
    resultSuccess="Y" resultDeclined="N" resultNsf="N" resultBadExpire="N"
    resultBadCardNumber="N"/>

<mantle.order.OrderPart orderId="${cartOrderId}" orderPartSeqId="01"
    vendorPartyId="ORG_BIZI_RETAIL" customerPartyId="CustJqp"
    shipmentMethodEnumId="ShMthGround" postalContactMechId="CustJqpAddr"
    telecomContactMechId="CustJqpTeln" partTotal="145.68"/>
<mantle.order.OrderItem orderId="${cartOrderId}" orderItemSeqId="01"
    orderPartSeqId="01" itemTypeEnumId="ItemProduct" productId="DEMO_1_1"
    itemDescription="Demo Product One-One" quantity="1" unitAmount="16.99"
    unitListPrice="19.99" isModifiedPrice="N"/>
<mantle.order.OrderItem orderId="${cartOrderId}" orderItemSeqId="02"
    orderPartSeqId="01" itemTypeEnumId="ItemProduct" productId="DEMO_3_1"
    itemDescription="Demo Product Three-One" quantity="5" unitAmount="7.77"
    unitListPrice="" isModifiedPrice="N"/>
<mantle.order.OrderItem orderId="${cartOrderId}" orderItemSeqId="03"
    orderPartSeqId="01" itemTypeEnumId="ItemProduct" productId="DEMO_2_1"
    itemDescription="Demo Product Two-One" quantity="7" unitAmount="12.12"
    unitListPrice="" isModifiedPrice="N"/>
```

当订单被安置的时候另一个很重要的事情就是订单条目的部分被仓库(在 Asset 实体中)保留。库存保留使用 AssetReservation 实体去追溯，所以我们有 3 条记录相对应（订单中的每个产品 Product 对应一条)。

前两条资产的记录来源于 ZzcProductDemoData.xml 文件中的例子数据。它们在 AssetDetail 记录中已经有可用的库存数量，因此它们可以使用负的 AssetDetail.**availableToPromiseDiff** 值去调整 Asset.**availableToPromiseTotal** 资产可用总数记录。

最后一条资产 Asset 记录有一个流水 ID 是因为这个产品没有库存量，并且这条资产 Asset 记录按照 ATP 和在手数量为 0 进行创建。在 AssetDetail 记录创建后，**availableToPromiseTotal** 被设置为“-7”，表示有一个数量为 7 的产品延期交货。它同样通过 AssetReservation.**quantityNotAvailable** 字段去追踪，作为希望“保留”的数量不可用。

```
<mantle.product.asset.Asset assetId="DEMO_1_1A"
    assetTypeEnumId="AstTpInventory" statusId="AstAvailable"
    ownerPartyId="ORG_BIZI_RETAIL" productId="DEMO_1_1" hasQuantity="Y"
    quantityOnHandTotal="100" availableToPromiseTotal="99"
    receivedDate="1265184000000" facilityId="ORG_BIZI_RETAIL_WH"/>
<mantle.product.issuance.AssetReservation assetReservationId="55500"
    assetId="DEMO_1_1A" productId="DEMO_1_1" orderId="${cartOrderId}"
    orderItemSeqId="01" reservationOrderEnumId="AsResOrdFifoRec"
    quantity="1" reservedDate="${effectiveTime}" sequenceNum="0"/>
<mantle.product.asset.AssetDetail assetDetailId="55500" assetId="DEMO_1_1A"
    effectiveDate="${effectiveTime}" availableToPromiseDiff="-1"
    assetReservationId="55500" productId="DEMO_1_1"/>

<mantle.product.asset.Asset assetId="DEMO_3_1A"
    assetTypeEnumId="AstTpInventory" statusId="AstAvailable"
    ownerPartyId="ORG_BIZI_RETAIL" productId="DEMO_3_1" hasQuantity="Y"
    quantityOnHandTotal="5" availableToPromiseTotal="0"
    receivedDate="1265184000000" facilityId="ORG_BIZI_RETAIL_WH"/>
<mantle.product.issuance.AssetReservation assetReservationId="55501"
    assetId="DEMO_3_1A" productId="DEMO_3_1" orderId="${cartOrderId}"
    orderItemSeqId="02" reservationOrderEnumId="AsResOrdFifoRec"
    quantity="5" reservedDate="${effectiveTime}" sequenceNum="0"/>
<mantle.product.asset.AssetDetail assetDetailId="55501" assetId="DEMO_3_1A"
    effectiveDate="${effectiveTime}" availableToPromiseDiff="-5"
    assetReservationId="55501" productId="DEMO_3_1"/>

<mantle.product.asset.Asset assetId="55500"
    assetTypeEnumId="AstTpInventory" statusId="AstAvailable"
    ownerPartyId="ORG_BIZI_RETAIL" productId="DEMO_2_1" hasQuantity="Y"
    quantityOnHandTotal="0" availableToPromiseTotal="-7"
    receivedDate="${effectiveTime}" facilityId="ORG_BIZI_RETAIL_WH"/>
<mantle.product.issuance.AssetReservation assetReservationId="55502"
    assetId="55500" productId="DEMO_2_1" orderId="${cartOrderId}"

    orderItemSeqId="03" reservationOrderEnumId="AsResOrdFifoRec"
    quantity="7" quantityNotAvailable="7" reservedDate="${effectiveTime}"/>
<mantle.product.asset.AssetDetail assetDetailId="55502" assetId="55500"
    effectiveDate="${effectiveTime}" availableToPromiseDiff="-7"
    assetReservationId="55502" productId="DEMO_2_1"/>
```

- ***销售订单发货***

这里有一个单独的服务调用能够发送整个子订单 OrderPart 的货物：**ship#OrderPart**。

```
shipResult = ec.service.sync()
    .name("mantle.shipment.ShipmentServices.ship#OrderPart")
    .parameters([orderId:cartOrderId, orderPartSeqId:orderPartSeqId])
    .call()
```

这个服务仅仅做了一部分的事情，但在真实世界的系统中这个服务调用，甚至于多个服务调用会有更多的细节的细致操作和更有用：

- ✓ mantle.shipment.ShipmentServices.create#OrderPartShipment（创建一个发货，为订单中所有的产品添加发货条目，打个包和选择航线，并将它们捆绑在一起）
- ✓ mantle.shipment.ShipmentServices.pack#ShipmentProduct（包含产品编号和每个订单条目中的数量）
- ✓ mantle.shipment.ShipmentServices.pack#Shipment（发货变为打包状态，并触发通过 mantle.account.InvoiceServices.create#SalesShipmentInvoices 服务来开票和获取信用卡支付）
- ✓ mantle.order.OrderServices.checkComplete#OrderPart（如果子订单的所有条目的状态全部都改变了，则状态变为完成）

✓ mantle.shipment.ShipmentServices.ship#Shipment

下面的是装运 Shipment 和关联实体的 XML。对每个产品（**productId**）都有一条发货条目（ShipmentItem）记录，并且发货条目源（ShipmentItemSource）记录关联订单条目 OrderItem 以及发票条目 InvoiceItem，还有保持分拣/打包的状态（这种情况下，已打包 Packed 状态就是调用 **pack#ShipmentProduct** 服务）。这里同样还有个 ShipmentPackage 记录加上 ShipmentPackageContent 记录去关联装运条目和打包的信息。最终，这里还有一个 ShipmentRouteSegment 记录以及 ShipmentPackageRouteSeg 记录去关联打包。

```
<mantle.shipment.Shipment shipmentId="${shipResult.shipmentId}"
    shipmentTypeEnumId="ShpTpSales" statusId="ShipShipped"
    fromPartyId="ORG_BIZI_RETAIL" toPartyId="CustJqp"/>
<mantle.shipment.ShipmentPackage shipmentId="${shipResult.shipmentId}"
    shipmentPackageSeqId="01"/>

<mantle.shipment.ShipmentItem shipmentId="${shipResult.shipmentId}"
    productId="DEMO_1_1" quantity="1"/>
<mantle.shipment.ShipmentItemSource shipmentItemSourceId="55500"

    shipmentId="${shipResult.shipmentId}" productId="DEMO_1_1"
    orderId="${cartOrderId}" orderItemSeqId="01" statusId="SisPacked"
    quantity="1" invoiceId="55500" invoiceItemSeqId="01"/>
<mantle.shipment.ShipmentPackageContent
    shipmentId="${shipResult.shipmentId}" shipmentPackageSeqId="01"
    productId="DEMO_1_1" quantity="1"/>

<mantle.shipment.ShipmentItem shipmentId="${shipResult.shipmentId}"
    productId="DEMO_3_1" quantity="5"/>
<mantle.shipment.ShipmentItemSource shipmentItemSourceId="55501"
    shipmentId="${shipResult.shipmentId}" productId="DEMO_3_1"
    orderId="${cartOrderId}" orderItemSeqId="02" statusId="SisPacked"
    quantity="5" invoiceId="55500" invoiceItemSeqId="02"/>
<mantle.shipment.ShipmentPackageContent
    shipmentId="${shipResult.shipmentId}" shipmentPackageSeqId="01"
    productId="DEMO_3_1" quantity="5"/>

<mantle.shipment.ShipmentItem shipmentId="${shipResult.shipmentId}"
    productId="DEMO_2_1" quantity="7"/>
<mantle.shipment.ShipmentItemSource shipmentItemSourceId="55502"
    shipmentId="${shipResult.shipmentId}" productId="DEMO_2_1"
    orderId="${cartOrderId}" orderItemSeqId="03" statusId="SisPacked"
    quantity="7" invoiceId="55500" invoiceItemSeqId="03"/>
<mantle.shipment.ShipmentPackageContent
    shipmentId="${shipResult.shipmentId}" shipmentPackageSeqId="01"
    productId="DEMO_2_1" quantity="7"/>

<mantle.shipment.ShipmentRouteSegment shipmentId="${shipResult.shipmentId}"
    shipmentRouteSegmentSeqId="01" destPostalContactMechId="CustJqpAddr"
    destTelecomContactMechId="CustJqpTeln"/>
<mantle.shipment.ShipmentPackageRouteSeg
    shipmentId="${shipResult.shipmentId}" shipmentPackageSeqId="01"
    shipmentRouteSegmentSeqId="01"/>
```

这里有个订单头（OrderHeader）的状态更新基于子订单 OrderPart 的完成：

```
<mantle.order.OrderHeader orderId="${cartOrderId}"
    statusId="OrderCompleted"/>
```

当一个发货条目 ShipmentItem（或更明确的说是发货条目源 ShipmentItemSource）被仓库打包，通常会被标记为保留，这样就有了一个 AssetReservation 记录。通过这种方式，发货的发行的资产（AssetIssuance）记录加上带有在手差异数量（**quantityOnHandDiff**）的资产明细（AssetDetail）记录就可以调整中的资产在手总数（Asset.**quantityOnHandTotal**）了。下

面是这次发货的记录：

```
<mantle.product.asset.Asset assetId="DEMO_1_1A" quantityOnHandTotal="99"
    availableToPromiseTotal="99"/>
<mantle.product.issuance.AssetIssuance assetIssuanceId="55500"
    assetId="DEMO_1_1A" assetReservationId="55500"
    orderId="${cartOrderId}" orderItemSeqId="01"
    shipmentId="${shipResult.shipmentId}" productId="DEMO_1_1"

    quantity="1"/>
<mantle.product.asset.AssetDetail assetDetailId="55503" assetId="DEMO_1_1A"
    effectiveDate="${effectiveTime}" quantityOnHandDiff="-1"
    assetReservationId="55500" shipmentId="${shipResult.shipmentId}"
    productId="DEMO_1_1" assetIssuanceId="55500"/>

<mantle.product.asset.Asset assetId="DEMO_3_1A" quantityOnHandTotal="0"
    availableToPromiseTotal="0"/>
<mantle.product.issuance.AssetIssuance assetIssuanceId="55501"
    assetId="DEMO_3_1A" assetReservationId="55501"
    orderId="${cartOrderId}" orderItemSeqId="02"
    shipmentId="${shipResult.shipmentId}" productId="DEMO_3_1"
    quantity="5"/>
<mantle.product.asset.AssetDetail assetDetailId="55504" assetId="DEMO_3_1A"
    effectiveDate="${effectiveTime}" quantityOnHandDiff="-5"
    assetReservationId="55501" shipmentId="${shipResult.shipmentId}"
    productId="DEMO_3_1" assetIssuanceId="55501"/>

<mantle.product.asset.Asset assetId="55500" quantityOnHandTotal="-7"
    availableToPromiseTotal="-7"/>
<mantle.product.issuance.AssetIssuance assetIssuanceId="55502"
    assetId="55500" assetReservationId="55502" orderId="${cartOrderId}"
    orderItemSeqId="03" shipmentId="${shipResult.shipmentId}"
    productId="DEMO_2_1" quantity="7"/>
<mantle.product.asset.AssetDetail assetDetailId="55505" assetId="55500"
    effectiveDate="${effectiveTime}" quantityOnHandDiff="-7"
    assetReservationId="55502" shipmentId="${shipResult.shipmentId}"
    productId="DEMO_2_1" assetIssuanceId="55502"/>
```

资产发行是一个影响财务会计的业务活动，所以这里会有一个总账过账的会计事务。这里有个异常，对于资产编号（**assetId**）为 55500，产品（**productId**）为 DEMO_2_1 没有会计事务 AcctgTrans，这是因为它是自动创建的而没有收购成本（**acquireCost**）。对于大部分的企业来说并不希望如此，收购成本（**acquireCost**）字段应该总是有数据的，但是对于简单的、无需关心追踪成本和仓库信息的系统需求时，这就是其所需要的。

```
<mantle.ledger.transaction.AcctgTrans acctgTransId="55500"
    acctgTransTypeEnumId="AttInventoryIssuance"
    organizationPartyId="ORG_BIZI_RETAIL"
    transactionDate="${effectiveTime}" isPosted="Y"
    postedDate="${effectiveTime}" glFiscalTypeEnumId="GLFT_ACTUAL"
    amountUomId="USD" assetId="DEMO_1_1A" assetIssuanceId="55500"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55500"
    acctgTransEntrySeqId="01" debitCreditFlag="C" amount="7.5"
    glAccountTypeEnumId="INVENTORY_ACCOUNT" glAccountId="140000"
    reconcileStatusId="AES_NOT_RECONCILED" isSummary="N"
    productId="DEMO_1_1"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55500"
    acctgTransEntrySeqId="02" debitCreditFlag="D" amount="7.5"
    glAccountTypeEnumId="COGS_ACCOUNT" glAccountId="501000"
    reconcileStatusId="AES_NOT_RECONCILED" isSummary="N"
    productId="DEMO_1_1"/>

<mantle.ledger.transaction.AcctgTrans acctgTransId="55501"
    acctgTransTypeEnumId="AttInventoryIssuance"
    organizationPartyId="ORG_BIZI_RETAIL"
    transactionDate="${effectiveTime}" isPosted="Y"
    postedDate="${effectiveTime}" glFiscalTypeEnumId="GLFT_ACTUAL"
    amountUomId="USD" assetId="DEMO_3_1A" assetIssuanceId="55501"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55501"
    acctgTransEntrySeqId="01" debitCreditFlag="C" amount="20"
    glAccountTypeEnumId="INVENTORY_ACCOUNT" glAccountId="140000"
    reconcileStatusId="AES_NOT_RECONCILED" isSummary="N"
    productId="DEMO_3_1"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55501"
    acctgTransEntrySeqId="02" debitCreditFlag="D" amount="20"
    glAccountTypeEnumId="COGS_ACCOUNT" glAccountId="501000"
    reconcileStatusId="AES_NOT_RECONCILED" isSummary="N"
    productId="DEMO_3_1"/>
```

就像上面提到的那样，当一个装运 Shipment 进行打包（**Packed**）状态时，将会触发装运 Shipment 的订单条目的发票 Invoice 生成。下面就是含有发票条目 InvoiceItem，以及关联发票条目 InvoiceItem 记录和其相对应的订单条目 OrderItem 记录的订单条目计费（OrderItemBilling）记录的发票 Invoice 样例：

```
<mantle.account.invoice.Invoice invoiceId="55500"
    invoiceTypeEnumId="InvoiceSales" fromPartyId="ORG_BIZI_RETAIL"
    toPartyId="CustJqp" statusId="InvoicePmtRecvd"
    invoiceDate="${effectiveTime}"
    description="Invoice for Order ${cartOrderId} part 01 and Shipment
        ${shipResult.shipmentId}" currencyUomId="USD"/>

<mantle.account.invoice.InvoiceItem invoiceId="55500" invoiceItemSeqId="01"
    itemTypeEnumId="ItemProduct" productId="DEMO_1_1" quantity="1"
    amount="16.99" description="Demo Product One-One"
    itemDate="${effectiveTime}"/>
<mantle.order.OrderItemBilling orderItemBillingId="55500"
    orderId="${cartOrderId}" orderItemSeqId="01" invoiceId="55500"
    invoiceItemSeqId="01" assetIssuanceId="55500"
    shipmentId="${shipResult.shipmentId}" quantity="1" amount="16.99"/>

<mantle.account.invoice.InvoiceItem invoiceId="55500" invoiceItemSeqId="02"
    itemTypeEnumId="ItemProduct" productId="DEMO_3_1" quantity="5"
    amount="7.77" description="Demo Product Three-One"
    itemDate="${effectiveTime}"/>
<mantle.order.OrderItemBilling orderItemBillingId="55501"
    orderId="${cartOrderId}" orderItemSeqId="02" invoiceId="55500"
    invoiceItemSeqId="02" assetIssuanceId="55501"
    shipmentId="${shipResult.shipmentId}" quantity="5" amount="7.77"/>
```

```
<mantle.account.invoice.InvoiceItem invoiceId="55500" invoiceItemSeqId="03"
    itemTypeEnumId="ItemProduct" productId="DEMO_2_1" quantity="7"
    amount="12.12" description="Demo Product Two-One"
    itemDate="${effectiveTime}"/>
<mantle.order.OrderItemBilling orderItemBillingId="55502"
    orderId="${cartOrderId}" orderItemSeqId="03" invoiceId="55500"
    invoiceItemSeqId="03" assetIssuanceId="55502"
    shipmentId="${shipResult.shipmentId}" quantity="7" amount="12.12"/>

<mantle.account.invoice.InvoiceItem invoiceId="55500" invoiceItemSeqId="04"
    itemTypeEnumId="ItemShipping" quantity="1" amount="5"
    description="Ground" itemDate="${effectiveTime}"/>
<mantle.order.OrderItemBilling orderItemBillingId="55503"
    orderId="${cartOrderId}" orderItemSeqId="04" invoiceId="55500"
    invoiceItemSeqId="04" shipmentId="${shipResult.shipmentId}"
    quantity="1" amount="5"/>
```

发票是有财务影响的记录，所以它们会通过会计业务进行过账。这里对每个发票条目 InvoiceItem 都有一个业务入口（AcctgTransEntry）去记入可用的销售科目（或者运输/处理接收到的科目）的贷方，同时一个平衡的入口去记入应收账款科目的借方。

```
<mantle.ledger.transaction.AcctgTrans acctgTransId="55502"
    acctgTransTypeEnumId="AttSalesInvoice"
    organizationPartyId="ORG_BIZI_RETAIL"
    transactionDate="${effectiveTime}" isPosted="Y"
    postedDate="${effectiveTime}" glFiscalTypeEnumId="GLFT_ACTUAL"
    amountUomId="USD" otherPartyId="CustJqp" invoiceId="55500"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55502"
    acctgTransEntrySeqId="01" debitCreditFlag="C" amount="16.99"
    glAccountId="401000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N" productId="DEMO_1_1" invoiceItemSeqId="01"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55502"
    acctgTransEntrySeqId="02" debitCreditFlag="C" amount="38.85"
    glAccountId="401000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N" productId="DEMO_3_1" invoiceItemSeqId="02"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55502"
    acctgTransEntrySeqId="03" debitCreditFlag="C" amount="84.84"
    glAccountId="401000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N" productId="DEMO_2_1" invoiceItemSeqId="03"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55502"
    acctgTransEntrySeqId="04" debitCreditFlag="C" amount="5"
    glAccountId="731200" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N" invoiceItemSeqId="04"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55502"
    acctgTransEntrySeqId="05" debitCreditFlag="D" amount="145.68"
    glAccountTypeEnumId="ACCOUNTS_RECEIVABLE" glAccountId="120000"
    reconcileStatusId="AES_NOT_RECONCILED" isSummary="N"/>
```

最后的操作时捕获信用卡支付的结果，记入 PaymentGatewayResponse 记录中并更新支付 Payment 状态为已交货（**Delivered**）。这里同样有个现金科目和应收账款科目的 AcctgTrans 记录。

```
<mantle.account.payment.Payment paymentId="${setInfoOut.paymentId}"
    statusId="PmntDelivered"/>
<mantle.account.payment.PaymentApplication paymentApplicationId="55500"
    paymentId="${setInfoOut.paymentId}" invoiceId="55500"
    amountApplied="145.68" appliedDate="${effectiveTime}"/>
<mantle.account.method.PaymentGatewayResponse
    paymentGatewayResponseId="55501" paymentOperationEnumId="PgoCapture"
    paymentId="${setInfoOut.paymentId}" paymentMethodId="CustJqpCc"
    amount="145.68" amountUomId="USD" transactionDate="${effectiveTime}"
    resultSuccess="Y" resultDeclined="N" resultNsf="N"
    resultBadExpire="N" resultBadCardNumber="N"/>

<mantle.ledger.transaction.AcctgTrans acctgTransId="55503"
    acctgTransTypeEnumId="AttIncomingPayment"
    organizationPartyId="ORG_BIZI_RETAIL"
    transactionDate="${effectiveTime}" isPosted="Y"
    glFiscalTypeEnumId="GLFT_ACTUAL" amountUomId="USD"
    otherPartyId="CustJqp" paymentId="${setInfoOut.paymentId}"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55503"
    acctgTransEntrySeqId="01" debitCreditFlag="C" amount="145.68"
    glAccountId="120000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55503"
    acctgTransEntrySeqId="02" debitCreditFlag="D" amount="145.68"
    glAccountId="122000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N"/>
```

随着货品运达，付款到账和所有业务都已经被过账，订单到收款的业务过程就完成了。

工作计划到收款

这个过程的 Spock 测试套件在 WorkPlanToCashBasicFlow.groovy 文件中。这是 HiveMind 项目管理应用支撑的主要过程。

例子中相关安装数据将在下面展示其中一部分，但你可以在 ZzaGlAccountsDemoData.xml，ZzbOrganizationDemoData.xml，ZzcProductDemoData.xml 文件中找到其余的那些。

➢ *供应商*

对于一个服务型企业的系统来说，首先要做的事情就是设置供应商，服务企业本身。它是个内部组织（OrgInternal）角色的组织（Organization）类型的当事人（Party）。它同样具有供应商（VendorBillFrom）角色。它还有一个完整的会计核算配置，通过使用 **init#PartyAccountingConfiguration** 服务从'DefaultSettings' Party（ZzaGlAccountsDemoData.xml 文件中的定义）中拷贝而来。使用 **create#Account** 服务去创建一个供应商组织的代表（为了 AR/AP 等），通常是一个有用户帐号（UserAccount）的个人（Person）类型的当事人（Party）。

```
long effectiveTime = System.currentTimeMillis()
ec.user.loginUser("john.doe", "moqui", null)
// set an effective date so data check works, etc
ec.user.setEffectiveTime(new Timestamp(effectiveTime))
effectiveThruDate = ec.l10n.parseTimestamp(
    ec.l10n.formatValue(ec.user.nowTimestamp,'yyyy-MM-dd HH:mm'),
        'yyyy-MM-dd HH:mm')
Map vendorResult = ec.service.sync()
    .name("mantle.party.PartyServices.create#Organization")
    .parameters([roleTypeId:'VendorBillFrom',
        organizationName:'Test Vendor']).call()
```

```
Map vendorCiResult = ec.service.sync()
    .name("mantle.party.ContactServices.store#PartyContactInfo")
    .parameters([partyId:vendorResult.partyId,
        postalContactMechPurposeId:'PostalPayment',
        telecomContactMechPurposeId:'PhonePayment',
        emailContactMechPurposeId:'EmailPayment', countryGeoId:'USA',
        address1:'51 W. Center St.', unitNumber:'1234', city:'Orem',
        stateProvinceGeoId:'USA_UT', postalCode:'84057',
        postalCodeExt:'4605', countryCode:'+1', areaCode:'801',
        contactNumber:'123-4567', emailAddress:'vendor.ar@test.com'])
     .call()
ec.service.sync().name("create#mantle.party.PartyRole")
    .parameters([partyId:vendorResult.partyId, roleTypeId:'OrgInternal'])
    .call()
ec.service.sync()
    .name("mantle.ledger.LedgerServices.init#PartyAccountingConfiguration")
    .parameters([sourcePartyId:'DefaultSettings',
        organizationPartyId:vendorResult.partyId]).call()

Map vendorRepResult = ec.service.sync()
    .name("mantle.party.PartyServices.create#Account")
    .parameters([firstName:'Vendor', lastName:'TestRep',
        emailAddress:'vendor.rep@test.com', username:'vendor.rep',
        newPassword:'moqui1!', newPasswordVerify:'moqui1!',
        loginAfterCreate:'false']).call()
Map repRelResult = ec.service.sync()
    .name("create#mantle.party.PartyRelationship")
    .parameters([relationshipTypeEnumId:'PrtRepresentative',
        fromPartyId:vendorRepResult.partyId, fromRoleTypeId:'Manager',
        toPartyId:vendorResult.partyId, toRoleTypeId:'VendorBillFrom',
        fromDate:ec.user.nowTimestamp]).call()
```

下面的是供应商组织 Organization 的记录及其联系信息：

```
<mantle.party.Party partyId="${vendorResult.partyId}"
    partyTypeEnumId="PtyOrganization"/>
<mantle.party.Organization partyId="${vendorResult.partyId}"
    organizationName="Test Vendor"/>
<mantle.party.PartyRole partyId="${vendorResult.partyId}"
    roleTypeId="OrgInternal"/>
<mantle.party.PartyRole partyId="${vendorResult.partyId}"
    roleTypeId="VendorBillFrom"/>

<mantle.party.contact.ContactMech
    contactMechId="${vendorCiResult.postalContactMechId}"
    contactMechTypeEnumId="CmtPostalAddress"/>
<mantle.party.contact.PostalAddress
    contactMechId="${vendorCiResult.postalContactMechId}"
    address1="51 W. Center St." unitNumber="1234" city="Orem"
    stateProvinceGeoId="USA_UT" countryGeoId="USA" postalCode="84057"
    postalCodeExt="4605"/>
<mantle.party.contact.PartyContactMech partyId="${vendorResult.partyId}"
    contactMechId="${vendorCiResult.postalContactMechId}"
    contactMechPurposeId="PostalPayment" fromDate="${effectiveTime}"/>
<mantle.party.contact.ContactMech
    contactMechId="${vendorCiResult.telecomContactMechId}"
    contactMechTypeEnumId="CmtTelecomNumber"/>
<mantle.party.contact.PartyContactMech partyId="${vendorResult.partyId}"
    contactMechId="${vendorCiResult.telecomContactMechId}"
    contactMechPurposeId="PhonePayment" fromDate="${effectiveTime}"/>
<mantle.party.contact.TelecomNumber
    contactMechId="${vendorCiResult.telecomContactMechId}" countryCode="+1"
    areaCode="801" contactNumber="123-4567"/>
<mantle.party.contact.ContactMech
    contactMechId="${vendorCiResult.emailContactMechId}"
    contactMechTypeEnumId="CmtEmailAddress"
    infoString="vendor.ar@test.com"/>
<mantle.party.contact.PartyContactMech partyId="${vendorResult.partyId}"
    contactMechId="${vendorCiResult.emailContactMechId}"
    contactMechPurposeId="EmailPayment" fromDate="${effectiveTime}"/>
```

这里还有供应商的会计核算配置记录。本例中各种配置记录（GlAccountTypeDefault，ItemTypeGlAccount，GlAccountOrganization，PaymentTypeGlAccount 等）是一个小的选集，同时还有很多其他的源于'DefaultSettings' Party 的副本。

```
<mantle.ledger.transaction.GlJournal
    glJournalId="${vendorResult.partyId}Error"
    glJournalName="Error Journal for ${vendorResult.partyId}"
    organizationPartyId="${vendorResult.partyId}"/>
<mantle.ledger.config.PartyAcctgPreference
    organizationPartyId="${vendorResult.partyId}"
    taxFormEnumId="TxfUsIrs1120" cogsMethodEnumId="CogsActualCost"
    baseCurrencyUomId="USD" invoiceSequenceEnumId="InvSqStandard"

    orderSequenceEnumId="OrdSqStandard"
    errorGlJournalId="${vendorResult.partyId}Error"/>
<mantle.ledger.config.GlAccountTypeDefault
    glAccountTypeEnumId="ACCOUNTS_RECEIVABLE"
    organizationPartyId="${vendorResult.partyId}" glAccountId="120000"/>
<mantle.ledger.config.GlAccountTypeDefault
    glAccountTypeEnumId="ACCOUNTS_PAYABLE"
    organizationPartyId="${vendorResult.partyId}" glAccountId="210000"/>
<mantle.ledger.config.PaymentMethodTypeGlAccount
    paymentMethodTypeEnumId="PmtCompanyCheck"
    organizationPartyId="${vendorResult.partyId}" glAccountId="111100"/>
<mantle.ledger.config.ItemTypeGlAccount glAccountId="402000" direction="O"
    itemTypeEnumId="ItemTimeEntry"
    organizationPartyId="${vendorResult.partyId}"/>
<mantle.ledger.config.ItemTypeGlAccount glAccountId="550000" direction="I"
    itemTypeEnumId="ItemTimeEntry"
    organizationPartyId="${vendorResult.partyId}"/>
<mantle.ledger.config.ItemTypeGlAccount itemTypeEnumId="ItemExpTravAir"
    direction="E" glAccountId="681000"
    organizationPartyId="${vendorResult.partyId}"/>
<mantle.ledger.account.GlAccountOrganization glAccountId="120000"
    organizationPartyId="${vendorResult.partyId}"/>
<mantle.ledger.account.GlAccountOrganization glAccountId="210000"
    organizationPartyId="${vendorResult.partyId}"/>
<mantle.ledger.config.PaymentTypeGlAccount
    paymentTypeEnumId="PtInvoicePayment"
    organizationPartyId="${vendorResult.partyId}" isPayable="N"
    isApplied="Y" glAccountId="120000"/>
<mantle.ledger.config.PaymentTypeGlAccount
    paymentTypeEnumId="PtInvoicePayment"
    organizationPartyId="${vendorResult.partyId}" isPayable="Y"
    isApplied="Y" glAccountId="210000"/>
```

下面的是供应商代表 Person 以及他的联系信息。注意这里的 **passwordSalt** 是随机生成的，所以 SHA-256 加密的密码和源于其他地方运行的密码有区别。

```
<mantle.party.Party partyId="${vendorRepResult.partyId}"
    partyTypeEnumId="PtyPerson" disabled="N"/>
<mantle.party.Person partyId="${vendorRepResult.partyId}"
    firstName="Vendor" lastName="TestRep"/>
<moqui.security.UserAccount userId="${vendorRepResult.userId}"
    username="vendor.rep" userFullName="Vendor TestRep"
    passwordHashType="SHA-256" passwordSetDate="${effectiveTime}"
    disabled="N" requirePasswordChange="N"
    emailAddress="vendor.rep@test.com" passwordSalt="{.rqlPt8x"
    partyId="${vendorRepResult.partyId}" currentPassword="32ce60c14d9e72c1
        fb17938ede30fe9de04390409cce7310743c2716a2c7bf89"/>
<mantle.party.contact.ContactMech
    contactMechId="${vendorRepResult.emailContactMechId}"
```

```
    contactMechTypeEnumId="CmtEmailAddress"
    infoString="vendor.rep@test.com"/>
<mantle.party.contact.PartyContactMech partyId="${vendorRepResult.partyId}"
    contactMechId="${vendorRepResult.emailContactMechId}"
    contactMechPurposeId="EmailPrimary" fromDate="${effectiveTime}"/>
<mantle.party.PartyRelationship
    partyRelationshipId="${repRelResult.partyRelationshipId}"
    relationshipTypeEnumId="PrtRepresentative"
    fromPartyId="${vendorRepResult.partyId}" fromRoleTypeId="Manager"
    toPartyId="${vendorResult.partyId}" toRoleTypeId="VendorBillFrom"
    fromDate="${effectiveTime}"/>
```

- ***工人和价格***

下面的代码为一个工人创建了一个个人（Person）类型的当事人 Party 和用户帐户（UserAccount），例如工作于任务上的某人。它同样创建了两条 RateAmount 记录，一条$60 是供应商对客户的计费，而另一条$40 是作为外部承包人的工人对供应商的计费。工人通过一条 PrtAgent 类型的 PartyRelationship 记录去关联作为代理商的供应商。

```
Map workerResult = ec.service.sync()
    .name("mantle.party.PartyServices.create#Account")
    .parameters([firstName:'Test', lastName:'Worker',
        emailAddress:'worker@test.com', username:'worker',
        newPassword:'moqui1!', newPasswordVerify:'moqui1!',
        loginAfterCreate:'false']).call()
Map workerRelResult = ec.service.sync()
    .name("create#mantle.party.PartyRelationship")
    .parameters([relationshipTypeEnumId:'PrtAgent',
        fromPartyId:workerResult.partyId, fromRoleTypeId:'Worker',
        toPartyId:vendorResult.partyId, toRoleTypeId:'VendorBillFrom',
        fromDate:ec.user.nowTimestamp]).call()
Map clientRateResult = ec.service.sync()
    .name("create#mantle.humanres.rate.RateAmount")
    .parameters([rateTypeEnumId:'RatpStandard',
        ratePurposeEnumId:'RaprClient', timePeriodUomId:'TF_hr',
        emplPositionClassId:'Programmer', fromDate:'2010-02-03 00:00:00',
        rateAmount:'60.00', rateCurrencyUomId:'USD',
        partyId:workerResult.partyId]).call()
Map vendorRateResult = ec.service.sync()
    .name("create#mantle.humanres.rate.RateAmount")
    .parameters([rateTypeEnumId:'RatpStandard',
        ratePurposeEnumId:'RaprVendor', timePeriodUomId:'TF_hr',
        emplPositionClassId:'Programmer', fromDate:'2010-02-03 00:00:00',
        rateAmount:'40.00', rateCurrencyUomId:'USD',
        partyId:workerResult.partyId]).call()
```

下面的是工人当事人 Party 以及计费价格的记录：

```
<mantle.party.Party partyId="${workerResult.partyId}"
    partyTypeEnumId="PtyPerson" disabled="N"/>
<mantle.party.Person partyId="${workerResult.partyId}" firstName="Test"
    lastName="Worker"/>
<moqui.security.UserAccount userId="${workerResult.userId}"
    username="worker" userFullName="Test Worker" passwordHashType="SHA-256"
    passwordSetDate="${effectiveTime}" disabled="N"
    requirePasswordChange="N" emailAddress="worker@test.com"
    partyId="${workerResult.partyId}" passwordSalt="{.rqlPt8x"
    currentPassword="32ce60c14d9e72c1fb17938ede30fe9de04390409cce7310743
        c2716a2c7bf89"/>
```

```
<mantle.party.contact.ContactMech
    contactMechId="${workerResult.emailContactMechId}"
    contactMechTypeEnumId="CmtEmailAddress" infoString="worker@test.com"/>
<mantle.party.contact.PartyContactMech partyId="${workerResult.partyId}"
    contactMechId="${workerResult.emailContactMechId}"
    contactMechPurposeId="EmailPrimary" fromDate="${effectiveTime}"/>
<mantle.party.PartyRelationship
    partyRelationshipId="${workerRelResult.partyRelationshipId}"
    relationshipTypeEnumId="PrtAgent" fromPartyId="${workerResult.partyId}"
    fromRoleTypeId="Worker" toPartyId="${vendorResult.partyId}"
    toRoleTypeId="VendorBillFrom" fromDate="${effectiveTime}"/>

<mantle.humanres.rate.RateAmount
    rateAmountId="${clientRateResult.rateAmountId}"
    rateTypeEnumId="RatpStandard" ratePurposeEnumId="RaprClient"
    timePeriodUomId="TF_hr" partyId="${workerResult.partyId}"
    emplPositionClassId="Programmer" fromDate="2010-02-03 00:00:00"
    rateAmount="60.00" rateCurrencyUomId="USD"/>
<mantle.humanres.rate.RateAmount
    rateAmountId="${vendorRateResult.rateAmountId}"
    rateTypeEnumId="RatpStandard" ratePurposeEnumId="RaprVendor"
    timePeriodUomId="TF_hr" partyId="${workerResult.partyId}"
    emplPositionClassId="Programmer" fromDate="2010-02-03 00:00:00"
    rateAmount="40.00" rateCurrencyUomId="USD"/>
```

- ***客户***

下面的代码是创建客户（CustomerBillTo）Organization，以及一个带有联系信息等的客户代表（带有一个 PrtRepresentative 类型的 PartyRelationship）的 Person。

```
Map clientResult = ec.service.sync()
    .name("mantle.party.PartyServices.create#Organization")
    .parameters([roleTypeId:'CustomerBillTo',
        organizationName:'Test Client']).call()
Map clientCiResult = ec.service.sync()
    .name("mantle.party.ContactServices.store#PartyContactInfo")
    .parameters([partyId:clientResult.partyId,
        postalContactMechPurposeId:'PostalBilling',

        telecomContactMechPurposeId:'PhoneBilling',
        emailContactMechPurposeId:'EmailBilling', countryGeoId:'USA',
        address1:'1350 E. Flamingo Rd.', unitNumber:'1234',
        city:'Las Vegas', stateProvinceGeoId:'USA_NV', postalCode:'89119',
        postalCodeExt:'5263', countryCode:'+1', areaCode:'702',
        contactNumber:'123-4567',   emailAddress:'client.ap@test.com'])
    .call()

Map clientRepResult = ec.service.sync()
    .name("mantle.party.PartyServices.create#Account")
    .parameters([firstName:'Client', lastName:'TestRep',
        emailAddress:'client.rep@test.com', username:'client.rep',
        newPassword:'moqui1!', newPasswordVerify:'moqui1!',
        loginAfterCreate:'false']).call()
Map repRelResult = ec.service.sync()
    .name("create#mantle.party.PartyRelationship")
    .parameters([relationshipTypeEnumId:'PrtRepresentative',
        fromPartyId:clientRepResult.partyId,
        fromRoleTypeId:'ClientBilling', toPartyId:clientResult.partyId,
        toRoleTypeId:'CustomerBillTo', fromDate:ec.user.nowTimestamp])
    .call()
```

下面的是客户，联系信息以及客户代表的记录：

```
<mantle.party.Party partyId="${clientResult.partyId}"
    partyTypeEnumId="PtyOrganization"/>
<mantle.party.Organization partyId="${clientResult.partyId}"
    organizationName="Test Client"/>
<mantle.party.PartyRole partyId="${clientResult.partyId}"
    roleTypeId="CustomerBillTo"/>

<mantle.party.contact.ContactMech
    contactMechId="${clientCiResult.postalContactMechId}"
    contactMechTypeEnumId="CmtPostalAddress"/>
<mantle.party.contact.PostalAddress
    contactMechId="${clientCiResult.postalContactMechId}"
    address1="1350 E. Flamingo Rd." unitNumber="1234" city="Las Vegas"
    stateProvinceGeoId="USA_NV" countryGeoId="USA" postalCode="89119"
    postalCodeExt="5263"/>
<mantle.party.contact.PartyContactMech partyId="${clientResult.partyId}"
    contactMechId="${clientCiResult.postalContactMechId}"
    contactMechPurposeId="PostalBilling" fromDate="${effectiveTime}"/>
<mantle.party.contact.ContactMech
    contactMechId="${clientCiResult.telecomContactMechId}"
    contactMechTypeEnumId="CmtTelecomNumber"/>
<mantle.party.contact.PartyContactMech partyId="${clientResult.partyId}"
    contactMechId="${clientCiResult.telecomContactMechId}"
    contactMechPurposeId="PhoneBilling" fromDate="${effectiveTime}"/>
<mantle.party.contact.TelecomNumber
    contactMechId="${clientCiResult.telecomContactMechId}" countryCode="+1"
    areaCode="702" contactNumber="123-4567"/>
<mantle.party.contact.ContactMech
    contactMechId="${clientCiResult.emailContactMechId}"
    contactMechTypeEnumId="CmtEmailAddress"
    infoString="client.ap@test.com"/>
<mantle.party.contact.PartyContactMech partyId="${clientResult.partyId}"
    contactMechId="${clientCiResult.emailContactMechId}"
    contactMechPurposeId="EmailBilling" fromDate="${effectiveTime}"/>

<mantle.party.Party partyId="${clientRepResult.partyId}"
    partyTypeEnumId="PtyPerson" disabled="N"/>
<mantle.party.Person partyId="${clientRepResult.partyId}"
    firstName="Client" lastName="TestRep"/>
<moqui.security.UserAccount userId="${clientRepResult.userId}"
    username="client.rep" userFullName="Client TestRep"
    passwordHashType="SHA-256" passwordSetDate="${effectiveTime}"
    disabled="N" requirePasswordChange="N"
    emailAddress="client.rep@test.com"
    partyId="${clientRepResult.partyId}" passwordSalt="{.rqlPt8x"
    currentPassword="32ce60c14d9e72c1fb17938ede30fe9de04390409cce7310743
        c2716a2c7bf89"/>
<mantle.party.contact.ContactMech
    contactMechId="${clientRepResult.emailContactMechId}"
    contactMechTypeEnumId="CmtEmailAddress"
    infoString="client.rep@test.com"/>
<mantle.party.contact.PartyContactMech partyId="${clientRepResult.partyId}"
    contactMechId="${clientRepResult.emailContactMechId}"
    contactMechPurposeId="EmailPrimary" fromDate="${effectiveTime}"/>
<mantle.party.PartyRelationship
    partyRelationshipId="${repRelResult.partyRelationshipId}"
    relationshipTypeEnumId="PrtRepresentative"
    fromPartyId="${clientRepResult.partyId}" fromRoleTypeId="ClientBilling"
    toPartyId="${clientResult.partyId}" toRoleTypeId="CustomerBillTo"
    fromDate="${effectiveTime}"/>
```

- ***项目和里程碑***

下面的代码创建了一个带有客户和供应商集合的项目（Project）类型的 WorkEffort，而且指派并创建了作为 Worker 的工人 Person。注意，用于工作分配的 WorkEffortParty 记录有一个 **emplPositionClassId** 字段用于表示 Programmer 的类型，它同时用于为计费价格查找上面创建的 RateAmount 记录。

```
ec.service.sync().name("mantle.work.ProjectServices.create#Project")
    .parameters([workEffortId:'TEST', workEffortName:'Test Project',
        statusId:'WeInProgress', clientPartyId:clientResult.partyId,
        vendorPartyId:vendorResult.partyId]).call()
ec.service.sync().name("create#mantle.work.effort.WorkEffortParty")
    .parameters([workEffortId:'TEST', partyId:workerResult.partyId,
        roleTypeId:'Worker', emplPositionClassId:'Programmer',
        fromDate:'2013-11-01', statusId:'PRTYASGN_ASSIGNED']).call()
```

下面的是项目和客户（CustomerBillTo）的记录，并有供应商（VendorBillFrom）和工人（Worker）信息与之关联：

```
<mantle.work.effort.WorkEffort workEffortId="TEST"
    workEffortTypeEnumId="WetProject" statusId="WeInProgress"
    workEffortName="Test Project"/>
<mantle.work.effort.WorkEffortParty workEffortId="TEST"
    partyId="EX_JOHN_DOE" roleTypeId="Manager" fromDate="${effectiveTime}"
    statusId="PRTYASGN_ASSIGNED"/>
<mantle.work.effort.WorkEffortParty workEffortId="TEST"
    partyId="${clientResult.partyId}" roleTypeId="CustomerBillTo"
    fromDate="${effectiveTime}"/>
<mantle.work.effort.WorkEffortParty workEffortId="TEST"
    partyId="${vendorResult.partyId}" roleTypeId="VendorBillFrom"
    fromDate="${effectiveTime}"/>
<mantle.work.effort.WorkEffortParty workEffortId="TEST"
    partyId="${workerResult.partyId}" roleTypeId="Worker"
    fromDate="1383282000000" statusId="PRTYASGN_ASSIGNED"
    emplPositionClassId="Programmer"/>
```

WorkEffort.**statusId** 字段被审计日志记录的，并且有 EntityAuditLog 记录去跟踪状态从计划中（In Planning）变化到执行中（In Progress）：

```
<moqui.entity.EntityAuditLog auditHistorySeqId="55911"
    changedEntityName="mantle.work.effort.WorkEffort"
    changedFieldName="statusId" pkPrimaryValue="TEST"
    oldValueText="WeInPlanning" newValueText="WeInProgress"
    changedDate="${effectiveTime}" changedByUserId="EX_JOHN_DOE"/>
```

下一步，我们将为项目创建一对里程碑信息：

```
ec.service.sync().name("mantle.work.ProjectServices.create#Milestone")
    .parameters([rootWorkEffortId:'TEST', workEffortId:'TEST-MS-01',
        workEffortName:'Test Milestone 1', estimatedStartDate:'2013-11-01',
        estimatedCompletionDate:'2013-11-30', statusId:'WeInProgress'])
    .call()

ec.service.sync().name("mantle.work.ProjectServices.create#Milestone")
    .parameters([rootWorkEffortId:'TEST', workEffortId:'TEST-MS-02',
        workEffortName:'Test Milestone 2', estimatedStartDate:'2013-12-01',
        estimatedCompletionDate:'2013-12-31', statusId:'WeApproved'])
    .call()
```

这里是里程碑的记录。它们是 WetMilestone 类型的，并使用 **rootWorkEffortId** 字段去关联项目信息。

```
<mantle.work.effort.WorkEffort workEffortId="TEST-MS-01"
    rootWorkEffortId="TEST" workEffortTypeEnumId="WetMilestone"
    statusId="WeInProgress" workEffortName="Test Milestone 1"
    estimatedStartDate="2013-11-01 00:00:00.0"
    estimatedCompletionDate="2013-11-30 00:00:00.0"/>
<mantle.work.effort.WorkEffort workEffortId="TEST-MS-02"
    rootWorkEffortId="TEST" workEffortTypeEnumId="WetMilestone"
    statusId="WeApproved" workEffortName="Test Milestone 2"
    estimatedStartDate="2013-12-01 00:00:00.0"
    estimatedCompletionDate="2013-12-31 00:00:00.0"/>
```

- ***任务和时间条目***

这些服务调用创建了 3 个带有目的，状态，优先级，预计工作时间等的任务：

```
ec.service.sync().name("mantle.work.TaskServices.create#Task")
    .parameters([rootWorkEffortId:'TEST', parentWorkEffortId:null,
        workEffortId:'TEST-001', milestoneWorkEffortId:'TEST-MS-01',
        workEffortName:'Test Task 1', estimatedCompletionDate:'2013-11-15',
        statusId:'WeApproved', assignToPartyId:workerResult.partyId,
        priority:3, purposeEnumId:'WepTask', estimatedWorkTime:10,
        description:'Will be really great when it\'s done'])
    .call()
ec.service.sync().name("mantle.work.TaskServices.create#Task")
    .parameters([rootWorkEffortId:'TEST', parentWorkEffortId:'TEST-001',
        workEffortId:'TEST-001A', milestoneWorkEffortId:'TEST-MS-01',
        workEffortName:'Test Task 1A',
        estimatedCompletionDate:'2013-11-15', statusId:'WeInPlanning',
        assignToPartyId:workerResult.partyId, priority:4,
        purposeEnumId:'WepNewFeature', estimatedWorkTime:2,
        description:'One piece of the puzzle'])
    .call()
ec.service.sync().name("mantle.work.TaskServices.create#Task")
    .parameters([rootWorkEffortId:'TEST', parentWorkEffortId:'TEST-001',
        workEffortId:'TEST-001B', milestoneWorkEffortId:'TEST-MS-01',
        workEffortName:'Test Task 1B',
        estimatedCompletionDate:'2013-11-15', statusId:'WeApproved',
        assignToPartyId:workerResult.partyId, priority:4,
        purposeEnumId:'WepFix', estimatedWorkTime:2,
        description:'Broken piece of the puzzle'])
    .call()
```

下面的是这些服务调用生成的记录，包括一个工作投入记录，它有一个 **rootWorkEffortId** 去关联产品，同时还有个 WorkEffortAssoc 记录去关联里程碑。这里同样对每个任务还有个 WorkEffortParty 记录去关联工人。注意这里的 **estimatedCompletionDate** 以毫秒为单位的，并以公历进行格式化的。这种方式同样适用于所有的实体 XML 导出数据，以避免时区等问题。

```
<mantle.work.effort.WorkEffort workEffortId="TEST-001"
    rootWorkEffortId="TEST" workEffortTypeEnumId="WetTask"
    purposeEnumId="WepTask" resolutionEnumId="WerUnresolved"
    statusId="WeApproved" priority="3" workEffortName="Test Task 1"
    description="Will be really great when it's done"

    estimatedCompletionDate="1384495200000" estimatedWorkTime="10"
    remainingWorkTime="10" timeUomId="TF_hr"/>
<mantle.work.effort.WorkEffortParty workEffortId="TEST-001"
    partyId="${workerResult.partyId}" roleTypeId="Worker"
    fromDate="${effectiveTime}" statusId="PRTYASGN_ASSIGNED"/>
```

```
<mantle.work.effort.WorkEffortAssoc workEffortId="TEST-MS-01"
    toWorkEffortId="TEST-001" workEffortAssocTypeEnumId="WeatMilestone"
    fromDate="${effectiveTime}"/>

<mantle.work.effort.WorkEffort workEffortId="TEST-001A"
    parentWorkEffortId="TEST-001" rootWorkEffortId="TEST"
    workEffortTypeEnumId="WetTask" purposeEnumId="WepNewFeature"
    resolutionEnumId="WerUnresolved" statusId="WeInPlanning" priority="4"
    workEffortName="Test Task 1A" description="One piece of the puzzle"
    estimatedCompletionDate="1384495200000" estimatedWorkTime="2"
    remainingWorkTime="2" timeUomId="TF_hr"/>
<mantle.work.effort.WorkEffortParty workEffortId="TEST-001A"
    partyId="${workerResult.partyId}" roleTypeId="Worker"
    fromDate="${effectiveTime}" statusId="PRTYASGN_ASSIGNED"/>
<mantle.work.effort.WorkEffortAssoc workEffortId="TEST-MS-01"
    toWorkEffortId="TEST-001A" workEffortAssocTypeEnumId="WeatMilestone"
    fromDate="${effectiveTime}"/>

<mantle.work.effort.WorkEffort workEffortId="TEST-001B"
    parentWorkEffortId="TEST-001" rootWorkEffortId="TEST"
    workEffortTypeEnumId="WetTask" purposeEnumId="WepFix"
    resolutionEnumId="WerUnresolved" statusId="WeApproved" priority="4"
    workEffortName="Test Task 1B" description="Broken piece of the puzzle"
    estimatedCompletionDate="1384495200000" estimatedWorkTime="2"
    remainingWorkTime="2" timeUomId="TF_hr"/>
<mantle.work.effort.WorkEffortParty workEffortId="TEST-001B"
    partyId="${workerResult.partyId}" roleTypeId="Worker"
    fromDate="${effectiveTime}" statusId="PRTYASGN_ASSIGNED"/>
<mantle.work.effort.WorkEffortAssoc workEffortId="TEST-MS-01"
    toWorkEffortId="TEST-001B" workEffortAssocTypeEnumId="WeatMilestone"
    fromDate="${effectiveTime}"/>
```

下面的代码首先更新这三个任务的状态为执行中（In Progress）。

然后这里有 3 种不同的服务调用方式去记录工作在任务上时间的例子，这些常用的选项都是用户可能会去使用于记录时间的。第一种指定了已经工作的小时数（**hours**）以及剩余的工作时间（**remainingWorkTime**），同时时间条目 TimeEntry 的起始和截止日期的计算基于截止时间（**thruDate**）被当作为当前日期/时间。第二种调用方式有已经工作的小时数（**hours**）以及休息时间（**breakHours**），并再次没有起始/截止日期，这种情况下，截止时间 **thruDate** 就是当前日期/时间，同时起始时间 **fromDate** 就是 **thruDate** 减去（**hours** + **breakHours**）。在第三种服务调用中指定了休息时间（**breakHours**），同时起始时间 **fromDate**，截止时间 **thruDate** 以及已工作时间 **hours** 都基于此进行计算。

最后将设置所有 3 个任务的状态都为已完成（Completed）。

```
ec.service.sync().name("mantle.work.TaskServices.update#Task")
    .parameters([workEffortId:'TEST-001', statusId:'WeInProgress']).call()
ec.service.sync().name("mantle.work.TaskServices.update#Task")
    .parameters([workEffortId:'TEST-001A', statusId:'WeInProgress']).call()
ec.service.sync().name("mantle.work.TaskServices.update#Task")
    .parameters([workEffortId:'TEST-001B', statusId:'WeInProgress']).call()


ec.service.sync().name("mantle.work.TaskServices.add#TaskTime")
    .parameters([workEffortId:'TEST-001', partyId:workerResult.partyId,
        rateTypeEnumId:'RatpStandard', remainingWorkTime:3, hours:6,
        fromDate:null, thruDate:null, breakHours:null]).call()

ec.service.sync().name("mantle.work.TaskServices.add#TaskTime")
    .parameters([workEffortId:'TEST-001A', partyId:workerResult.partyId,
        rateTypeEnumId:'RatpStandard', remainingWorkTime:1, hours:1.5,
        fromDate:null, thruDate:null, breakHours:0.5]).call()

ec.service.sync().name("mantle.work.TaskServices.add#TaskTime")
    .parameters([workEffortId:'TEST-001B', partyId:workerResult.partyId,
        rateTypeEnumId:'RatpStandard', remainingWorkTime:0.5, hours:null,
        fromDate:"2013-11-03 12:00:00", thruDate:"2013-11-03 15:00:00",
        breakHours:1]).call()

ec.service.sync().name("mantle.work.TaskServices.update#Task")
    .parameters([workEffortId:'TEST-001', statusId:'WeComplete',
        resolutionEnumId:'WerCompleted']).call()
ec.service.sync().name("mantle.work.TaskServices.update#Task")
    .parameters([workEffortId:'TEST-001A', statusId:'WeComplete',
        resolutionEnumId:'WerCompleted']).call()
ec.service.sync().name("mantle.work.TaskServices.update#Task")
    .parameters([workEffortId:'TEST-001B', statusId:'WeComplete',
        resolutionEnumId:'WerCompleted']).call()
```

下面是已更新的 WorkEffort 记录，其中已更新的字段包括解决方式，状态以及剩余的和实际工作时间。同样下面还有每个任务的时间条目 TimeEntry 记录。注意 **rateAmountId** 字段自动的基于工人当事人 Party 的大部分关联的 RateAmount 记录进行填充。费率用于显示费率和总的时间条目 TimeEntry 的花费，同时当工人和客户的发票条目被创建时，发票条目上稍后会计算出总额（就像下面两条发票和支付部分中展示的那样）。

```
<mantle.work.effort.WorkEffort workEffortId="TEST-001"
    resolutionEnumId="WerCompleted" statusId="WeComplete"
    estimatedWorkTime="10" remainingWorkTime="3" actualWorkTime="6"/>
<mantle.work.time.TimeEntry timeEntryId="55900"
    partyId="${workerResult.partyId}" rateTypeEnumId="RatpStandard"
    rateAmountId="${clientRateResult.rateAmountId}"
    vendorRateAmountId="${vendorRateResult.rateAmountId}"
    fromDate="${effectiveThruDate.time-(6*60*60*1000)}"
    thruDate="${effectiveThruDate.time}" hours="6"
    workEffortId="TEST-001"/>

<mantle.work.effort.WorkEffort workEffortId="TEST-001A"
    resolutionEnumId="WerCompleted" statusId="WeComplete"
    estimatedWorkTime="2" remainingWorkTime="1" actualWorkTime="1.5"/>
<mantle.work.time.TimeEntry timeEntryId="55901"
    partyId="${workerResult.partyId}" rateTypeEnumId="RatpStandard"
    rateAmountId="${clientRateResult.rateAmountId}"
    vendorRateAmountId="${vendorRateResult.rateAmountId}"
    fromDate="${effectiveThruDate.time-(2*60*60*1000)}"
    thruDate="${effectiveThruDate.time}" hours="1.5" breakHours="0.5"
    workEffortId="TEST-001A"/>
```

```
<mantle.work.effort.WorkEffort workEffortId="TEST-001B"
    resolutionEnumId="WerCompleted" statusId="WeComplete"
    estimatedWorkTime="2" remainingWorkTime="0.5" actualWorkTime="2"/>
<mantle.work.time.TimeEntry timeEntryId="55902"
    partyId="${workerResult.partyId}" rateTypeEnumId="RatpStandard"
    rateAmountId="${clientRateResult.rateAmountId}"
    vendorRateAmountId="${vendorRateResult.rateAmountId}"
    fromDate="1383501600000" thruDate="1383512400000" hours="2"
    breakHours="1" workEffortId="TEST-001B"/>
```

- ***需要的请求和任务***

下面的代码展示了如何去创建一个指派给工人的支持请求，从已提交状态更新为已评估，为这个请求创建一个任务，完成任务，然后完成这个请求的一系列过程。

```
Map createReqResult = ec.service.sync()
    .name("mantle.request.RequestServices.create#Request")
    .parameters([clientPartyId:clientResult.partyId,
        assignToPartyId:workerResult.partyId, requestName:'Test Request 1',
        description:'Description of Test Request 1', priority:7,
        requestTypeEnumId:'RqtSupport', statusId:'ReqSubmitted',
        responseRequiredDate:'2013-11-15 15:00:00']).call()
ec.service.sync().name("mantle.request.RequestServices.update#Request")
    .parameters([requestId:createReqResult.requestId,
        statusId:'ReqReviewed']).call()

Map createReqTskResult = ec.service.sync()
    .name("mantle.work.TaskServices.create#Task")
    .parameters([rootWorkEffortId:'TEST',
        workEffortName:'Test Request 1 Task',
        estimatedCompletionDate:'2013-11-15', statusId:'WeApproved',
        assignToPartyId:workerResult.partyId, priority:7,
        purposeEnumId:'WepTask', estimatedWorkTime:2,
        description:'']).call()
ec.service.sync().name("create#mantle.request.RequestWorkEffort")
    .parameters([workEffortId:createReqTskResult.workEffortId,
        requestId:createReqResult.requestId]).call()
ec.service.sync().name("mantle.work.TaskServices.update#Task")
    .parameters([workEffortId:createReqTskResult.workEffortId,
        statusId:'WeComplete', resolutionEnumId:'WerCompleted']).call()

ec.service.sync().name("mantle.request.RequestServices.update#Request")
    .parameters([requestId:createReqResult.requestId,
        statusId:'ReqCompleted']).call()
```

下面的是 Request 记录以及其关联的带有工人和客户（顾客）的 RequestParty 记录。这里同样还有任务的 WorkEffort，工人的 WorkEffortParty 记录以及其用于关联 Request 的 RequestWorkEffort 记录。

```
<mantle.request.Request requestId="${createReqResult.requestId}"
    requestTypeEnumId="RqtSupport" statusId="ReqCompleted"
    requestName="Test Request 1"
    description="Description of Test Request 1" priority="7"
    responseRequiredDate="1384549200000"
    requestResolutionEnumId="RrUnresolved" filedByPartyId="EX_JOHN_DOE"/>
<mantle.request.RequestParty requestId="${createReqResult.requestId}"
    partyId="${workerResult.partyId}" roleTypeId="Worker"
    fromDate="${effectiveTime}"/>
<mantle.request.RequestParty requestId="${createReqResult.requestId}"
    partyId="${clientResult.partyId}" roleTypeId="CustomerBillTo"
    fromDate="${effectiveTime}"/>

<mantle.work.effort.WorkEffort
    workEffortId="${createReqTskResult.workEffortId}"
    rootWorkEffortId="TEST" workEffortTypeEnumId="WetTask"
    purposeEnumId="WepTask" resolutionEnumId="WerCompleted"
    statusId="WeComplete" priority="7" workEffortName="Test Request 1 Task"
    estimatedCompletionDate="1384495200000" estimatedWorkTime="2"
    remainingWorkTime="2" timeUomId="TF_hr"/>
<mantle.work.effort.WorkEffortParty
    workEffortId="${createReqTskResult.workEffortId}"
    partyId="${workerResult.partyId}" roleTypeId="Worker"
    fromDate="${effectiveTime}" statusId="PRTYASGN_ASSIGNED"/>
<mantle.request.RequestWorkEffort requestId="${createReqResult.requestId}"
    workEffortId="${createReqTskResult.workEffortId}"/>
```

- ***工人的发票和付款***

工人对服务供应商的发票 Invoice（内部组织运行的系统）有费用和时间条目两部分。服务 **create#ProjectExpenseInvoice** 从项目的 WorkEffort（ID: TEST）中获取大部分的 Invoice 设置信息（包括供应商，支付给，当事人信息），同时将工人指定为 **fromPartyId**。

一旦发票已创建，随后的两个服务调用将会添加发票费用的条目，然后调用 **create#ProjectInvoiceItems** 服务为工人当事人在 TEST 项目中的所有时间条目去添加发票条目，并设置发票条目的 **ratePurposeEnumId** 为 RaprVendor，这样价格和其他明细构成了工人对供应商的发票（相对于供应商对客户的发票）。接着我们标记发票为已接收 Received。这应该是由一个供应商组织的代表去完成，比如发票收票方的当事人。

最后一个服务调用，**create#InvoicePayment**，记录了一个发票已交货的支票支付。

```
expInvResult = ec.service.sync()
    .name("mantle.account.InvoiceServices.create#ProjectExpenseInvoice")
    .parameters([workEffortId:'TEST', fromPartyId:workerResult.partyId])
    .call()
ec.service.sync().name("create#mantle.account.invoice.InvoiceItem")
    .parameters([invoiceId:expInvResult.invoiceId,
        itemTypeEnumId:'ItemExpTravAir', description:'United SFO-LAX',
        itemDate:'2013-11-02', quantity:1, amount:345.67]).call()
ec.service.sync().name("create#mantle.account.invoice.InvoiceItem")
    .parameters([invoiceId:expInvResult.invoiceId,
        itemTypeEnumId:'ItemExpTravLodging',
        description:'Fleabag Inn 2 nights', itemDate:'2013-11-04',
        quantity:1, amount:123.45]).call()

ec.service.sync()
    .name("mantle.account.InvoiceServices.create#ProjectInvoiceItems")
    .parameters([invoiceId:expInvResult.invoiceId,
        workerPartyId:workerResult.partyId, ratePurposeEnumId:'RaprVendor',
        workEffortId:'TEST',
        thruDate:new Timestamp(effectiveTime + 1)]).call()

ec.service.sync().name("update#mantle.account.invoice.Invoice")
    .parameters([invoiceId:expInvResult.invoiceId,
        statusId:'InvoiceReceived']).call()

Map expPmtResult = ec.service.sync()
    .name("mantle.account.PaymentServices.create#InvoicePayment")
    .parameters([invoiceId:expInvResult.invoiceId,
        statusId:'PmntDelivered', amount:'849.12',
        paymentMethodTypeEnumId:'PmtCompanyCheck',
        effectiveDate:'2013-11-10 12:00:00', paymentRefNum:'1234',
        comments:'Delivered by Fedex']).call()
```

下面的是创建的发票记录，包括费用条目以及三个时间条目项（每条对应一个任务的时间条目）：

```
<mantle.account.invoice.Invoice invoiceId="${expInvResult.invoiceId}"
    invoiceTypeEnumId="InvoiceSales" fromPartyId="${workerResult.partyId}"
    toPartyId="${vendorResult.partyId}" statusId="InvoicePmtSent"
    invoiceDate="${effectiveTime}" currencyUomId="USD"/>
<mantle.account.invoice.InvoiceItem invoiceId="${expInvResult.invoiceId}"
    invoiceItemSeqId="01" itemTypeEnumId="ItemExpTravAir" quantity="1"
    amount="345.67" description="United SFO-LAX" itemDate="1383368400000"/>
<mantle.account.invoice.InvoiceItem invoiceId="${expInvResult.invoiceId}"
    invoiceItemSeqId="02" itemTypeEnumId="ItemExpTravLodging" quantity="1"
    amount="123.45" description="Fleabag Inn 2 nights"
    itemDate="1383544800000"/>
<mantle.account.invoice.InvoiceItem invoiceId="${expInvResult.invoiceId}"
    invoiceItemSeqId="03" itemTypeEnumId="ItemTimeEntry" quantity="6"
    amount="40" itemDate="${effectiveThruDate.time-(6*60*60*1000)}"/>
<mantle.work.time.TimeEntry timeEntryId="55900"
    vendorInvoiceId="${expInvResult.invoiceId}"
    vendorInvoiceItemSeqId="03"/>
<mantle.account.invoice.InvoiceItem invoiceId="${expInvResult.invoiceId}"
    invoiceItemSeqId="04" itemTypeEnumId="ItemTimeEntry" quantity="1.5"
    amount="40" itemDate="${effectiveThruDate.time-(2*60*60*1000)}"/>
<mantle.work.time.TimeEntry timeEntryId="55901"
    vendorInvoiceId="${expInvResult.invoiceId}"
    vendorInvoiceItemSeqId="04"/>
<mantle.account.invoice.InvoiceItem invoiceId="${expInvResult.invoiceId}"
    invoiceItemSeqId="05" itemTypeEnumId="ItemTimeEntry" quantity="2"
    amount="40" itemDate="1383501600000"/>
<mantle.work.time.TimeEntry timeEntryId="55902"
    vendorInvoiceId="${expInvResult.invoiceId}"
    vendorInvoiceItemSeqId="05"/>
```

这里的是这个发票 GL 总帐过账的会计事务，每个发票条目统一入口，同时有应付款账号的平衡条目：

```
<mantle.ledger.transaction.AcctgTrans acctgTransId="55900"
    acctgTransTypeEnumId="AttPurchaseInvoice"
    organizationPartyId="${vendorResult.partyId}"
    transactionDate="${effectiveTime}" isPosted="Y"
    postedDate="${effectiveTime}" glFiscalTypeEnumId="GLFT_ACTUAL"
    amountUomId="USD" otherPartyId="${workerResult.partyId}"
    invoiceId="${expInvResult.invoiceId}"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55900"
    acctgTransEntrySeqId="01" debitCreditFlag="D" amount="345.67"
    glAccountId="681000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N" invoiceItemSeqId="01"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55900"
    acctgTransEntrySeqId="02" debitCreditFlag="D" amount="123.45"
    glAccountId="681000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N" invoiceItemSeqId="02"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55900"
    acctgTransEntrySeqId="03" debitCreditFlag="D" amount="240"
    glAccountId="550000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N" invoiceItemSeqId="03"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55900"
    acctgTransEntrySeqId="04" debitCreditFlag="D" amount="60"
    glAccountId="550000" reconcileStatusId="AES NOT RECONCILED"
    isSummary="N" invoiceItemSeqId="04"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55900"
    acctgTransEntrySeqId="05" debitCreditFlag="D" amount="80"
    glAccountId="550000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N" invoiceItemSeqId="05"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55900"
    acctgTransEntrySeqId="06" debitCreditFlag="C" amount="849.12"
    glAccountTypeEnumId="ACCOUNTS_PAYABLE" glAccountId="210000"
    reconcileStatusId="AES_NOT_RECONCILED" isSummary="N"/>
<mantle.work.effort.WorkEffortInvoice invoiceId="${expInvResult.invoiceId}"
    workEffortId="TEST"/>
```

下面的是源于供应商（内部的组织）对工人的支票支付记录，支付应用将其应用于发票以及支付的会计事务上：

```
<mantle.account.payment.Payment paymentId="${expPmtResult.paymentId}"
    paymentTypeEnumId="PtInvoicePayment"
    fromPartyId="${vendorResult.partyId}"
    toPartyId="${workerResult.partyId}"
    paymentMethodTypeEnumId="PmtCompanyCheck" statusId="PmntDelivered"
    effectiveDate="1384106400000" paymentRefNum="1234"
    comments="Delivered by Fedex" amount="849.12" amountUomId="USD"/>
<mantle.account.payment.PaymentApplication
    paymentApplicationId="${expPmtResult.paymentApplicationId}"
    paymentId="${expPmtResult.paymentId}"
    invoiceId="${expInvResult.invoiceId}" amountApplied="849.12"
    appliedDate="${effectiveTime}"/>

<mantle.ledger.transaction.AcctgTrans acctgTransId="55901"
    acctgTransTypeEnumId="AttOutgoingPayment"
    organizationPartyId="${vendorResult.partyId}"
    transactionDate="${effectiveTime}" isPosted="Y"
    postedDate="${effectiveTime}" glFiscalTypeEnumId="GLFT_ACTUAL"
    amountUomId="USD" otherPartyId="${workerResult.partyId}"
    paymentId="${expPmtResult.paymentId}"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55901"
    acctgTransEntrySeqId="01" debitCreditFlag="D" amount="849.12"
    glAccountId="210000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55901"
    acctgTransEntrySeqId="02" debitCreditFlag="C" amount="849.12"
    glAccountId="111100" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N"/>
```

- ***客户发票和付款***

当所有包括工人的费用和项目设置等事情都设置完成，调用 **create#ProjectInvoiceItems** 服务为在 TEST 项目中工人当事人的所有时间条目去添加发票条目，并设置发票条目的 **ratePurposeEnumId** 为 RaprClient，这样价格和其他明细构成了供应商对客户的发票（相对于工人对供应商的发票）。传递到服务上的 **thruDate** 参数用于告知服务去获得项目中到这个日期/时间尚未结清的所有费用和时间条目。然后我们标记发票状态为已完成 Finalized，并将触发发票的总帐过账。

```
clientInvResult = ec.service.sync()
    .name("mantle.account.InvoiceServices.create#ProjectInvoiceItems")
    .parameters([ratePurposeEnumId:'RaprClient', workEffortId:'TEST',
        thruDate:new Timestamp(effectiveTime + 1)]).call()
ec.service.sync().name("update#mantle.account.invoice.Invoice")
    .parameters([invoiceId:clientInvResult.invoiceId,
        statusId:'InvoiceFinalized']).call()
```

下面的是带有时间条目和费用发票条目的供应商对客户的发票记录，以及 InvoiceItemAssoc 记录，它关联了供应商对客户发票上的费用项和作为原始记录的工人对供应商发票的费用项（这是费用如何被记录以及它们如何被标记的方式）。

```
<mantle.account.invoice.Invoice invoiceId="${clientInvResult.invoiceId}"
    invoiceTypeEnumId="InvoiceSales" fromPartyId="${vendorResult.partyId}"
    toPartyId="${clientResult.partyId}" statusId="InvoiceFinalized"
    invoiceDate="${effectiveTime}" currencyUomId="USD"
    description="Invoice for projectTest Project [TEST] "/>
<mantle.account.invoice.InvoiceItem
    invoiceId="${clientInvResult.invoiceId}" invoiceItemSeqId="01"
    itemTypeEnumId="ItemTimeEntry" quantity="6" amount="60"
    itemDate="${effectiveThruDate.time-(6*60*60*1000)}"/>
<mantle.work.time.TimeEntry timeEntryId="55900"
    invoiceId="${clientInvResult.invoiceId}" invoiceItemSeqId="01"/>
<mantle.account.invoice.InvoiceItem
    invoiceId="${clientInvResult.invoiceId}" invoiceItemSeqId="02"
    itemTypeEnumId="ItemTimeEntry" quantity="1.5" amount="60"
    itemDate="${effectiveThruDate.time-(2*60*60*1000)}"/>
<mantle.work.time.TimeEntry timeEntryId="55901"
    invoiceId="${clientInvResult.invoiceId}" invoiceItemSeqId="02"/>
<mantle.account.invoice.InvoiceItem
    invoiceId="${clientInvResult.invoiceId}" invoiceItemSeqId="03"
    itemTypeEnumId="ItemTimeEntry" quantity="2" amount="60"
    itemDate="1383501600000"/>
<mantle.work.time.TimeEntry timeEntryId="55902"
    invoiceId="${clientInvResult.invoiceId}" invoiceItemSeqId="03"/>
<mantle.account.invoice.InvoiceItem
    invoiceId="${clientInvResult.invoiceId}" invoiceItemSeqId="04"
    itemTypeEnumId="ItemExpTravAir" quantity="1" amount="345.67"
    description="United SFO-LAX" itemDate="1383368400000"/>
<mantle.account.invoice.InvoiceItemAssoc invoiceItemAssocId="55900"
    invoiceId="${expInvResult.invoiceId}" invoiceItemSeqId="01"
    toInvoiceId="${clientInvResult.invoiceId}" toInvoiceItemSeqId="04"
    invoiceItemAssocTypeEnumId="IiatBillThrough" quantity="1"
    amount="345.67"/>
<mantle.account.invoice.InvoiceItem
    invoiceId="${clientInvResult.invoiceId}" invoiceItemSeqId="05"
    itemTypeEnumId="ItemExpTravLodging" quantity="1" amount="123.45"
    description="Fleabag Inn 2 nights" itemDate="1383544800000"/>
<mantle.account.invoice.InvoiceItemAssoc invoiceItemAssocId="55901"
    invoiceId="${expInvResult.invoiceId}" invoiceItemSeqId="02"
    toInvoiceId="${clientInvResult.invoiceId}" toInvoiceItemSeqId="05"
    invoiceItemAssocTypeEnumId="IiatBillThrough" quantity="1"
    amount="123.45"/>
```

下面的是发票的 GL 总帐过账的会计事务记录，所有发票条目统一入口，同时有应收款账号的平衡条目。注意时间条目和费用条目的不同 **glAccountId** 值。

```
<mantle.ledger.transaction.AcctgTrans acctgTransId="55902"
    acctgTransTypeEnumId="AttSalesInvoice"
    organizationPartyId="${vendorResult.partyId}"
    transactionDate="${effectiveTime}" isPosted="Y"
    postedDate="${effectiveTime}" glFiscalTypeEnumId="GLFT_ACTUAL"
    amountUomId="USD" otherPartyId="${clientResult.partyId}"
    invoiceId="${clientInvResult.invoiceId}"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55902"
    acctgTransEntrySeqId="01" debitCreditFlag="C" amount="360"
    glAccountId="402000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N" invoiceItemSeqId="01"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55902"
    acctgTransEntrySeqId="02" debitCreditFlag="C" amount="90"
    glAccountId="402000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N" invoiceItemSeqId="02"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55902"
    acctgTransEntrySeqId="03" debitCreditFlag="C" amount="120"
    glAccountId="402000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N" invoiceItemSeqId="03"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55902"
    acctgTransEntrySeqId="04" debitCreditFlag="C" amount="345.67"
    glAccountId="681000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N" invoiceItemSeqId="04"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55902"
    acctgTransEntrySeqId="05" debitCreditFlag="C" amount="123.45"
    glAccountId="681000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N" invoiceItemSeqId="05"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55902"
    acctgTransEntrySeqId="06" debitCreditFlag="D" amount="1,039.12"
    glAccountTypeEnumId="ACCOUNTS_RECEIVABLE" glAccountId="120000"
    reconcileStatusId="AES_NOT_RECONCILED" isSummary="N"/>
```

这个服务调用是用于记录一个已交付的支付，它是以公司支票形式进行支付这个发票的，并且是自动的从客户对供应商进行创建的：

```
Map clientPmtResult = ec.service.sync()
    .name("mantle.account.PaymentServices.create#InvoicePayment")
    .parameters([invoiceId:clientInvResult.invoiceId,
        statusId:'PmntDelivered', amount:1039.12,
        paymentMethodTypeEnumId:'PmtCompanyCheck',
        effectiveDate:'2013-11-12 12:00:00', paymentRefNum:'54321'])
    .call()
```

这里的第一条记录展示了发票的状态更新为支付已接收。然后我们还有支付记录以及发票的支付应用信息。在此之后的是总帐过账的支付会计事务记录。

```
<mantle.account.invoice.Invoice invoiceId="${clientInvResult.invoiceId}"
    statusId="InvoicePmtRecvd"/>
<mantle.account.payment.Payment paymentId="${clientPmtResult.paymentId}"
    paymentTypeEnumId="PtInvoicePayment"
    fromPartyId="${clientResult.partyId}"
    toPartyId="${vendorResult.partyId}"
    paymentMethodTypeEnumId="PmtCompanyCheck" statusId="PmntDelivered"
    effectiveDate="1384279200000" paymentRefNum="54321" amount="1,039.12"
    amountUomId="USD"/>
<mantle.account.payment.PaymentApplication
    paymentApplicationId="${clientPmtResult.paymentApplicationId}"
    paymentId="${clientPmtResult.paymentId}"
    invoiceId="${clientInvResult.invoiceId}" amountApplied="1,039.12"
    appliedDate="${effectiveTime}"/>

<mantle.ledger.transaction.AcctgTrans acctgTransId="55903"
    acctgTransTypeEnumId="AttIncomingPayment"
    organizationPartyId="${vendorResult.partyId}"
    transactionDate="${effectiveTime}" isPosted="Y"
    postedDate="${effectiveTime}" glFiscalTypeEnumId="GLFT_ACTUAL"
    amountUomId="USD" otherPartyId="${clientResult.partyId}"
    paymentId="${clientPmtResult.paymentId}"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55903"
    acctgTransEntrySeqId="01" debitCreditFlag="C" amount="1,039.12"
    glAccountId="120000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55903"
    acctgTransEntrySeqId="02" debitCreditFlag="D" amount="1,039.12"
    glAccountId="111100" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N"/>
```