

Domain-Driven Design in PHP

A Highly Practical Guide

**Carloç Buenoçvinoç Chriçtian Soronellaç Keyvan Akbary**



**BIRMINGHAM - MUMBAI**

## Domain-Driven Design in PHP

版权所有©2017 Carlos Buenosvinos，Christian Soronellas，Keyvan Akbary

版权所有. 未经出版商事先书面许可，不得将本书的任何部分复制，存储在检索系统中，或以任何形式或任何方式进行传播，除非嵌入关键文章或评论中的简短引语.

本书的准备工作已尽全力，以确保所提供信息的准确性。 但是，本书中所包含的信息不附带任何明示或暗示的担保。 作者，Packt出版公司及其经销商和分销商均不对本书直接或间接引起或声称造成的任何损害负责.

Packt Publishing致力于通过适当使用资本提供有关本书中提及的所有公司和产品的商标信息。 但Packt Publishing不能保证这些信息的准确性.

首次发布：2017年6月 Production参考：1090617

由Packt Publishing Ltd出版.

发行地点

利弗莱街35号伯明翰

B3 2PB，英国.

ISBN 978-1-78728-494-4

[www.packtpub.com](http://www.packtpub.com/)

**Authors**

Carlos Buenosvinos Christian Soronellas Keyvan Akbary

**Technical Editor**

Joel Wilfred D'souza

**Acquisition Editor**

Frank Pohlmann

**Layout co-ordinator**

Aparna Bhagat

**Indexer**

Pratik Shirodkar

我必须承认，当我第一次听说PHP领域驱动设计倡议时，我有点担心。 这种危险是双重的：首先，当浏览目录时，主题看起来像是其他几个领域驱动设计书中已经提供的内容的重复。 其次，专门针对PHP社区编写一本关于领域驱动设计的书似乎不必要地缩小了，特别是因为领域驱动设计本身不是语言特定的。 因此，这可能会阻止PHP开发人员超越自己的社区边界，特别是考虑到PHP的范围之外还有很多事情。 事实上，即使是领域驱动设计也是其中之一，因为它不是源于PHP社区.

看完这本书后，我很高兴地通知你，我的担忧已经失效！

关于我的第一个问题：当然与以前出版的领域驱动设计书籍有一些重叠。 然而作者却克制自己。 理论部分正是你需要能够理解代码示例中发生的事情。 此外，如果您从未阅读过另一本“领域驱动设计”一书，这本书将为您提供开始在代码中应用某些领域驱动设计原则和模式所需的内容，因为它本质上是实用的.

我的第二个关注 - 关于本书的PHP方面 - 已经得到很好的解决。 事实证明，在PHP世界中有很多关于领域驱动设计的内容。 本书特别针对由PHP开发人员组成的受众。 代码示例类似于真实世界的PHP项目，作者使用我们从使用Symfony或Silex的项目中了解的编程风格。 为了持久化Domain对象，使用了Doctrine ORM--它是PHP事实上的标准数据映射器 -.

本书还满足了我在PHP社区中经常见到的需求：需要具体的例子。 对于作者如何应用某些在实际项目中被误解或滥用的风险较低的想法，作者提出正确的说明并不容易。 而在本质上是哲学性的领域驱动设计中，这更具挑战性.

就本书而言，作者并不害怕展示许多有用的例子，以及一些有趣的替代解决方案。 他们不仅仅是在解决方案上挥手， 他们花时间提供详细的解释 - 例如他们谈论如何为具有大量域事件的聚合保存快照，或者何时讨论使用RabbitMQ集成有界上下文。 我不记得以前在关于Domain-Driven Design的书或文章中看到过这些东西的实现.

就我个人而言，领域驱动设计是当今软件开发中最有趣的主题之一。 有这么多的发现，并且有很多相关的主题：敏捷软件开发，TDD和BDD，还有活的文档，可视化和知识处理技术。 一旦你开始研究所有这些，你就会意识到Domain-Driven Design是一个值得研究的专业领域，因为它使你能够作为一个软件开发人员增加更多的自身价值.

所以，我想我想说的是：深入阅读本书，从中学习，然后拿起另一本书（参见本书最后的参考书目以获得有关未来阅读的建议）。 持续学习是在软件行业保持最新状态的基本部分，所以不要停留在这里.

哦，顺便说一句：如果你有机会去巴塞罗那，请确保你参加了许多PHP或Symfony活动之一。 这个社区很大，很友善，而且有很多有趣的想法。 你也会在这里找到这本书的作者。 他们都投资在当地的PHP社区，并乐意与您分享他们的见解和经验！

**Matthias Noback**

Symfony一年的作者

# 关于作者

**Carlos Buenosvinos是一位PHP Extreme程序员，拥有超过15年开发Web应用程序的经验，并拥有超过10年作为20至100人的技术主管和CTO领导团队的经验。 他是一名认证的ScrumMaster（CSM），他曾在敏捷实践方面接受过二十多家不同公司的培训，包括员工和顾问。 在技术方面，他是Zend PHP工程师，Zend Framework工程师和MySQL认证。 他还是PHP Barcelona用户组的董事会成员。 他曾与电子商务（Atrapalo和eBay），付款处理（Vendo），分类（Emagister）和B2B招聘工具（XING）合作。 他对JavaScript，DevOps和Scala感兴趣。 他喜欢开发移动，树莓派和游戏**.

Twitter: [@buenosvinos](https://twitter.com/buenosvinos)

Web: [https://carlosbuenosvinos.com](https://carlosbuenosvinos.com/)

GitHub: <https://github.com/carlosbuenosvinos>

**Christian Soronellas是一位充满激情的软件开发人员，软件熟练工和工匠学徒。 他是一位在Web开发领域拥有超过10年经验的极限程序员。 他还是Zend PHP 5.3认证工程师，Zend Framework认证工程师和SensioLabs认证Symfony开发人员。 他曾担任过自由职业者，以及Privalia，Emagister，Atrapalo和Enalquiler担任软件架构师**.

Twitter: [@theUniC](https://twitter.com/theUniC)

GitHub: <https://github.com/theUniC>

**Keyvan Akbary是一名软件开发人员，他热爱软件基础知识，工艺运动，极限编程，SOLID原则，Clean Code，设计模式和测试。 他也是一名零星的功能程序员。 他理解技术是提供价值的媒介。 作为一名自由撰稿人，在视频流媒体（Youzee）和在线市场（MyBuilder）上，他一直从事无数项目 - 除了创办众筹公司（Funddy）之外。 目前，Keyvan作为伦敦TransferWise的首席开发人员在FinTech工作**.

Twitter: @keyvanakbary

Web: [http://keyvanakbary.com](http://keyvanakbary.com/)

GitHub: <https://github.com/keyvanakbary>

# 致谢

首先，我们要感谢所有的朋友和家人。 没有他们的支持，写这本书本来就是一件更加艰巨的任务。 感谢您安排我们的时间表并照顾我们的孩子，从而腾出时间让我们专注于写作。 你太棒了，这本书的一部分也是你的.

我们是三个西班牙人，他们用英文写了一本书，所以如果你猜我们的英文不够完美，那你就是对的。 幸运的是，Edd Mann从一开始就支持我们这个语言。 他不仅是一个伟大的合作者，也是一个伟大的朋友，我们应该对他表示非常感谢。 最后的审查由专业复制编辑Natalye Childress完成。 她做了一个伟大的工作，重写我们的话，使他们可以理解。 非常感谢。 我们的书更轻松，更愉快阅读.

巴塞罗那的一组PHP开发人员为我们称之为el camino del rigor或严谨的道路辩护。 它存在于工艺运动之前，它意味着与所有对我们堆叠的东西进行斗争，以便以特殊的方式建造特殊的东西。

来自该组的两名特定开发人员和朋友是Albert Casademont和Ricard Clau，他们都是致力于社区的非凡人士。 非常感谢您帮助修改过程。 你的贡献非常有价值.

我们要感谢每一位在我们应用领域驱动设计的公司中与我们合作过的开发人员。 我们知道你在学习和应用这些概念时一直在挣扎。 你们有些人在开始时并不那么开明，但在使用了基本的积木之后，你们就成了福音传道者。 感谢您的信仰.

从我们把Leanpub的第一章开始的那一刻起，我们的书就出售了。 在开始阶段购买这本书的早期采用者为我们提供了非常需要的爱和支持来完成此项任务。 感谢继续前进的动力.

也感谢Matthias Noback对本书的前言和反馈。 由于他的贡献，最终结果会更好.

特别提到Vaughn Vernon--不仅仅因为他的工作对我们来说是一个令人难以置信的信息和灵感来源，还因为他帮助我们找到了一位出色的出版商，给了我们宝贵的建议，并与我们分享了一些想法。 非常感谢你的帮助.

最后但并非最不重要的一点是，我们想向所有报告问题，提出建议并以其他方式贡献给我们的GitHub存储库的人员表示感谢。对大家，谢谢。你已经帮助我们改进了这本书。更重要的是，你已经帮助社区成长，并帮助其他开发人员成为更好的开发人员。正如Robert C. Martin在他的书“清洁代码：敏捷软件工艺手册”中写道的那样，“你读这本书有两个原因：第一，你是一个程序员;第二，你想成为一个更好的程序员。需要更好的程序员。“感谢Jordi Abad，Jonathan Wondrusch，CésarRodríguez，Yannick Voyer，Victor Guardiola，OriolGonzález，Henry Snoek，Tom Jowitt，Sascha Schimke，Sven Herrmann，Daniel Abad，Luis Rovirosa，Luis Cordova，RaúlRamos，Juan Maturana，NilPortugués ，尼古拉·祖耶夫，费尔南多·普拉达斯，劳尔·阿拉亚，尼尔·布鲁克斯，休伯特·贝格，亚历山大·雷克斯，塞巴斯蒂安·马丘卡，尼古拉斯·奥尔加特，塞巴斯蒂安·斯托克，弗拉基米尔·赫拉邦，弗拉达斯·迪尔斯和马克·奥贝.

有关与您的图书相关的支持文件和下载，请访问www.PacktPub.com。

您知道Packt提供每本已出版书籍的电子书版本，PDF和ePub文件可用吗？ 您可以在www.PacktPub.com上升级到电子书版本，并且作为印刷书籍客户，您有权在电子书副本上享受折扣。 请通过service@packtpub.com与我们联系，了解更多详情。

在www.PacktPub.com上，您还可以阅读一系列免费技术文章，注册一系列免费时事通讯，并在Packt书籍和电子书上获得独家折扣和优惠。



<https://www.packtpub.com/mapt>

通过Mapt获得最多的按需软件技能。 Mapt让您可以完全访问所有Packt书籍和视频课程，以及行业领先的工具，以帮助您规划个人发展并推动您的职业发展.

## 为什么订阅?

Packt出版的每本书都可以全面搜索Copy and paste, 打印和书签内容

按需提供，并可通过网络浏览器访问

感谢您购买这本Packt书。 在Packt，质量是我们编辑过程的核心.

如果您想加入我们的常规审核团队，您可以发送电子邮件至customerreviews@packtpub.com。 我们通过免费电子书和视频授予我们的定期评论者以换取宝贵的反馈。 帮助我们无情地改进我们的产品！

*这本书献给我最亲爱的凡妮莎，以及瓦伦蒂娜和加布里埃拉。 感谢您的爱，您的支持和耐心.*

*– Carlos*

*给我亲爱的埃琳娜。 没有你的鼓励，你的爱和耐心，这本书是不可能的.*

* *Christian*

*给我的父母，约翰和梅赛德斯，他们让我摆脱了束缚。 这将是许多人的第一本书。 为了我的爱，克拉拉，为您提供无条件的支持和无限的耐心.*

* *Keyvan*

目录

前言 [1](#_bookmark1)

第1章：领域驱动设计入门 [7](#_bookmark2)

为什么领域驱动设计很重要 [8](#_bookmark5)

领域驱动设计的三大支柱 [8](#_bookmark7)

无处不在的语言 [9](#_bookmark8)

事件风暴 [9](#_bookmark10)

考虑领域驱动设计 [10](#_bookmark12)

整蛊部分11

战略概述11

相关活动：微服务和独立系统12

包起来 [14](#_bookmark19)

第2章：建筑风格15

好日子15

分层架构19

模型 - 视图 - 控制器20

分层架构示例21

21型

视图23

控制器25

反向依赖：六角形建筑26

依赖倒置原理（DIP）26

应用六角形建筑27

命令查询责任隔离（CQRS）29

写模型31

阅读模型34

使写模型与读模型同步36

事件采购42

总结49

第3章：价值对象50

定义50

价值对象与实体51

货币和金钱示例52

特点54

量度，量化或描述54

不变性54

概念整体56

价值平等57

替换性59

无副作用的行为59

基本类型61

测试价值对象61

坚持价值对象63

坚持单一价值对象64

使用Ad Hoc ORM 64的嵌入式值

内含价值（Embeddables）与学说> = 2.5。\* 67

内含价值与学说<= 2.4。\* 69

序列化的LOB和Ad Hoc ORM 72

使用JMS序列化器改进序列化73

序列化LOB与学说74

学说对象映射类型74

学说自定义类型76

坚持价值对象的集合79

序列化为单列80的集合

由加入表81支持的集合

以学说加入表82支持的集合

采用Ad Hoc ORM 85的加入表作为备份集合

由数据库实体支持的集合85

NoSQL的86

PostgreSQL JSONB和MySQL JSON Type 86

安全87

总结87

第4章：实体88

介绍88

对象VS.原始类型90

身份操作92

持久性机制产生身份92

代孕身份93

活动记录与Vs.用于富域模型的数据映射器94

客户端提供标识95

应用程序生成标识96

其他有界上下文生成标识99

坚持实体99

设置原则99

映射实体100

使用带注释的代码映射实体100

使用XML映射实体102

映射实体身份103

最终映射文件105

测试实体106

DateTime是否109

将所有日期作为参数传递110

测试类别111

外部伪造113

反思115

验证116

属性验证116

整个对象验证118

解耦验证消息120

验证对象组成122

实体和域名事件122

总结123

第5章：服务125

应用程序服务126

域服务128

域服务和基础设施服务130

代码重用问题133

测试域服务135

贫血域模型与丰富的域模型136

贫血域模型破坏封装141

贫血领域模型带来了错误的代码重用意识141

如何避免贫血领域模型142

总结142

第6章：域名事件143

引言144

定义144

短篇故事145

隐喻146

真实世界示例146

特点148

命名约定148

域名事件和无处不在的语言148

不变性149

为事件建模149

学说事件152

坚持域名事件154

活动商店155

从域模型发布事件158

从实体发布域事件158

从域或应用程序服务发布您的域事件161

域名事件发布者如何工作161

设置DomainEventListeners 163

测试域事件165

将消息传播到远程有界上下文166

消息166

将域服务与REST同步176

*总结178*

*第7章：模块179*

*总体概述180*

*在PHP 180中利用模块*

*第一级命名空间180*

*PEAR样式命名空间180*

*PSR-0和PSR-4命名空间181*

*有界的上下文和应用程序184*

*在模块中构建代码185*

*设计指南187*

*基础设施层中的模块190*

*混合不同的技术196*

*应用层中的模块198*

*总结198*

*第8章：聚合199*

*引言199*

*重要概念200*

*ACID 200*

*交易201*

*隔离级别204*

*参照完整性204*

*锁定205*

*并发205*

*悲观并发控制（PCC）205*

*与教条206*

*乐观并发控制207*

*使用Elasticsearch 207*

*与教条210*

*什么是聚合？ 213*

*马丁福勒说什么...... 213*

*维基百科说什么...... 214*

*为什么要聚合？ 214*

*一点历史217*

*解剖总结217*

*综合设计规则220*

*基于业务真不变量的设计聚合220*

*小聚集比大聚合223*

*通过身份引用其他实体224*

*修改每个事务和请求的一个聚合224*

*示例应用程序服务：用户和愿望225*

*没有不变，两个聚合226*

*每个用户不超过三个愿望233*

*悲观并发控制236*

*乐观并发控制238*

*交易246*

*总结246*

*第9章：工厂247*

*工厂方法聚合根247*

*强制不变量248*

*工厂服务249*

*建筑规格249*

*构建聚合体254*

*测试工厂257*

*对象母亲258*

*测试数据生成器259*

*总结262*

*第10章：存储库263*

*定义264*

*存储库不是DAOs 264*

*面向收藏的知识库265*

*内存中实现271*

*学说ORM 272*

*对象映射273*

*学说自定义映射类型273*

*实体管理器276*

*DQL实现276*

*持久性存储库278*

*Redis实施278*

*SQL实现280*

*额外行为283*

*查询存储库284*

*规格模式285*

*内存中实现285*

*SQL实现287*

*管理交易288*

*测试库291*

*使用内存中的实现测试您的服务295*

*总结295*

*第11章：应用程序296*

*请求297*

*构建应用程序服务请求298*

*请求设计300*

*使用基元300*

*序列化300*

*没有业务逻辑300*

*没有测试300*

*应用服务剖析301*

*依赖倒置302*

*实例化应用程序服务303*

*自定义应用程序服务305*

*执行306*

*每类应用服务一类306*

*每个类306的多个应用程序服务方法*

*返回值307*

*来自聚合实例的DTO 308*

*数据变换器310*

*复合布局上的多个应用服务313*

*AJAX内容集成313*

*ESI内容集成314*

*Symfony子请求314*

*一个控制器，多个应用服务314*

*测试应用程序服务315*

*交易317*

*安全319*

*域名事件319*

*命令处理程序320*

*战术图书馆和其他选项320*

*总结321*

*第12章：集成有界上下文322*

*通过数据存储集成322*

*整合关系324*

*客户 - 供应商324*

*独立的方式324*

*循规蹈矩325*

*实现有界上下文集成325*

*现代RPC 325*

*消息队列330*

*总结335*

*第13章：使用PHP 336的六角形体系结构*

*引言336*

*第一种方法337*

*存储库和持久边缘339*

*解耦业务和持久性341*

*将我们的坚持迁移到Redis 343*

*将业务和Web框架分开344*

*使用API​​评估一个想法347*

*控制台应用程序评分349*

*测试评价一个想法UseCase 350*

*测试基础设施354*

*Arggg，这么多的依赖！ 355*

*域服务和通知Hexagon Edge 357*

*让我们回顾358*

*六角形建筑359*

*要点359*

*下一步是什么？ 359*

*第14章：参考书目360*

*第15章：结束* [362](#_bookmark469)

[Index](#_bookmark470) [363](#_bookmark470)

# 前言

2014年，在与领域驱动设计有关的两年阅读和工作之后，Carlos和Christian，朋友和同事前往柏林参加Vaughn Vernon的实施领域驱动设计研讨会。 训练非常棒，在旅行之前在脑海中旋转的所有概念突然变得非常真实。 但是，他们是一个充满Java和.NET开发人员的房间中唯一的两位PHP开发人员.

Around the same time, [php[tek]](https://tek.phparch.com/), an annual PHP conference, opened its call for papers, and Carlos sent one about Hexagonal Architecture. His talk was rejected, but Eli White — of [musketeers.me](http://musketeers.me/) and [php[architect]](https://www.phparch.com/) fame — got in touch with him a month later wondering if he was interested in writing an article about Hexagonal Architecture for the magazine php[architect]. So in June 2014, *Hexagonal Architecture with PHP* was published.

That article, which you'll find in the [Appendix](#_bookmark433), was the origin of this book.

In late 2014, Carlos and Christian talked about extending the article and sharing all their knowledge of and experience in applying Domain-Driven Design in production. They were very excited about the idea behind the book: helping the PHP community delve into Domain-Driven Design from a practical approach. At that time, concepts such as Rich Domain Models and framework-agnostic applications weren't so common in the PHP community. So in December 2014, the first commit to the GitHub book repository was pushed.

Around the same time, in a parallel universe, Keyvan co-founded Funddy, a crowdfunding platform for the masses built on top of the concepts and building blocks of Domain-Driven Design. Domain-Driven Design proved itself effective in the exploratory process and modeling of building an early-stage startup like Funddy. It also helped handle the complexity of the company, with its constantly changing environment and requirements.

And after connecting with Carlos and Christian and discussing the book, Keyvan proudly signed on as the third writer.

Together, we've written the book we wanted to have when we started with Domain-Driven Design. It's full of examples, production-ready code, shortcuts, and our recommendations based on our experiences of what worked and what didn't for our respective teams. We arrived at Domain-Driven Design via its building blocks — Tactical Patterns — which is why this book is mainly about them. Reading it will help you learn them, write them, and implement them. You'll also discover how to integrate Bounded Contexts using synchronous and asynchronous approaches, which will open your world to strategic design

— though the latter is a road you'll have to discover on your own.

This book is heavily inspired by [Implementing Domain-Driven Design](http://www.amazon.com/Implementing-Domain-Driven-Design-Vaughn-Vernon/dp/0321834577) by Vaughn Vernon (aka *the Red Book*), and [Domain-Driven Design: Tackling Complexity in the Heart of](http://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215) [Software](http://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215) by Eric Evans (aka *the Blue Book*). You should buy both books. You should read them carefully. You should love them.

## 谁应该读这本书

If you're a PHP Developer, Architect, or Tech Lead, we highly recommend this book. It will help you become a better professional. It will give you a new overview of and approach to the applications you're developing. If you're a Junior profile, getting into Value Objects, Entities, Repositories, and Domain Events is important in order to model any Domain you'll face in the future. For an average profile, understanding the benefits of Hexagonal Architecture and the boundaries between your framework and your Application is key for writing code that's easier to maintain in the real world (framework migrations, testing, etc.). More advanced readers will have fun both exploring how to use Domain Events in order to integrate Applications and delving deeper into Aggregate design.

Although Domain-Driven Design is not about technology, you still need it to make HTTP requests to access your Domain. Throughout the book, we recommend using specific PHP frameworks and libraries, such as Symfony, Silex, and Doctrine. For some examples, we also use specific technologies, such as MySQL, RabbitMQ, Redis, and Elasticsearch.

However, most important are the behind-the-scenes concepts — concepts that are applicable regardless of the technology used to implement them.

Additionally, the book is loaded with tons of details and examples, such as how to properly design and implement all the building blocks of Domain-Driven Design — including Value Objects, Entities, Services, Domain Events, Aggregates, Factories, Repositories, and Application Services — with PHP. It explains what the role of the main PHP libraries and frameworks used in Domain-Driven Design are. The book also teaches how to apply Hexagonal Architecture within your application, regardless of whether you use an open source framework or your own one. Finally, it shows how to integrate Bounded Contexts using REST frameworks and messaging mechanisms. If you're interested in any of these subjects, this book is for you.

## DDD and PHP Community

In 2016, Carlos and Christian went to the first official Domain-Driven Design conference, [DDD Europe](http://dddeurope.com/). They were really happy to see some PHP open source leaders, such as Marco Pivetta (Doctrine) and Sebastian Bergmann (PHPUnit), attending the conference.

Domain-Driven Design arrived in the PHP community two years prior to that conference. However, there's still a lack of documentation and real code examples. Why? We think not many people have worked with this kind of approach in production yet — even people in other more established communities such as Java. Maybe this is because their project complexity is low, or maybe it's because they don't know how to do it. Whatever the reason, this book is written for the community. One of our goals is to teach you how you can write an application that solves your Domain issues without being coupled to specific frameworks or technologies.

## Summary of Chapters

The book is arranged with each chapter exploring a separate tactical building block of Domain-Driven Design. It also includes an introduction to Domain-Driven Design, information on how to integrate different Bounded Contexts or applications, and an appendix.

### 第1章：领域驱动设计入门

What is Domain-Driven Design about? What role does it play in complex systems? Is it worth learning about and exploring? What are the main concepts a developer needs to know when jumping into it?

### Chapter 2: Architectural Styles

Bounded Contexts can be implemented in different ways and using different approaches. However, two styles are getting more popular, and they are Hexagonal Architecture and CQRS + ES. In this chapter, we'll see these two main Architectural Styles, understand what their main strengths are, and discover when to use them.

### Chapter 3: Value Objects

Value Objects are the basic pieces for rich modeling. We'll learn what their properties are and what makes them so important. We'll figure out how to persist them using Doctrine and custom ORMs. We'll show how to properly validate and unit test them. And finally, we'll see what a test case of testing immutability looks like.

### Chapter 4: Entities

Entities are Domain-Driven Design building blocks that are uniquely identified and mutable. We'll see how to create and validate them and how to properly map them using a custom ORM and Doctrine. We'll also assess whether or not annotations are the best mapping approach for Entities and look at the different strategies for generating an Identity.

### Chapter 5: Domain Services

In this chapter, you'll learn about what a Domain Service is and when to use it. We'll review what Anemic Domain Models and Rich Domain Models are. Lastly, we'll deal with Infrastructure issues when writing Domain Services.

### Chapter 6: Domain-Events

Domain Events are a great Inversion of Control (IoC) mechanism. In Domain-Driven Design, they're important for communicating different Bounded Contexts asynchronously, improving your Application performance using eventual consistency, and decoupling your Application from its Infrastructure.

### Chapter 7: Modules

With so many tactical building blocks, it's a bit difficult to know where to place them in code, especially if you're dealing with a framework like Symfony. We'll review how PHP namespaces can be used for implementing Modules. We'll also discover different

hierarchies of folders for organizing Domain Model code, Application Code, and Infrastructure Code.

### Chapter 8: Aggregates

Aggregates are probably the most difficult part of tactical Domain-Driven Design. We'll look at the key concepts when dealing with them and discover how to design them. We'll also propose a practical scenario where two Aggregates become one when adding a business rule, and we'll demonstrate how the rest of the objects must be refactored.

### Chapter 9: Factories

Factory Methods and objects help us keep business invariants, which is why they're so important in Domain-Driven Design. Here, we'll also explore the relationship between Factories and Aggregates.

### Chapter 10: Repositories

Repositories are key for retrieving and adding Entities and Aggregates to collections. We'll review the different types of Repositories and learn how to implement them using Doctrine, custom ORMs, and Redis.

### Chapter 11: Application

An Application is the thin layer that connects outside clients to your Domain. In this chapter, we'll show you how to write your Application Services so that they're easy to test and keep thin. We'll also review how to prepare request objects, define dependencies, and return results.

### Chapter 12: Integrating Bounded Contexts

We'll explore the different tactical approaches to communicate Bounded Contexts and see real implementations. REST is our suggestion for synchronous communication, and messaging with RabbitMQ is our suggestion for asynchronous communication.

### Appendix: Hexagonal Architecture with PHP

Here is where you'll find the original article written by Carlos and published by php[architect] in June 2014.

## 代码和示例

The authors have created an organization at GitHub called [Domain-Driven Design in PHP](https://github.com/dddinphp), which is where all the code examples from this book, additional snippets, and some complete sample projects are available. For example, you can find [Last Wishes](https://github.com/dddinphp/last-wishes), a simple Domain-Driven Design-style application showing different examples explained in this book. Additionally, you'll find our [*CQRS Blog*](https://github.com/dddinphp/blog-cqrs), along with [Gamify](https://github.com/dddinphp/last-wishes-gamify), a Bounded Context that adds gamification capabilities to *Last Wishes*.

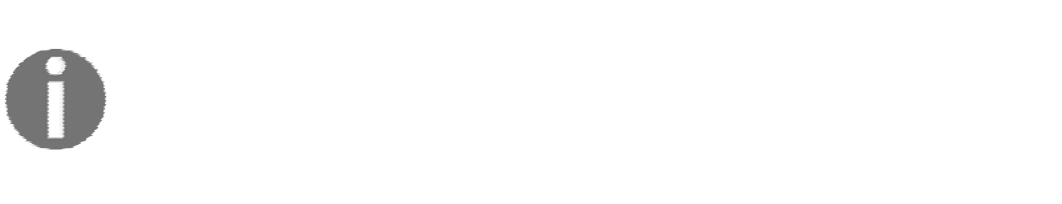
Finally, if you find any issue or fix or have a suggestion or comment while reading this book, you can create an issue in the [DDD in PHP Book Issues](https://github.com/dddinphp/book-issues) repository. We fix them as they come in. If you're interested, we also urge you to watch our projects and provide feedback.

# Getting Started with Domain-

****

Driven Design

那么，什么是大惊小怪？ 如果你已经阅读了Vaughn Vernon和Eric Evans关于这个主题的书籍，那么你可能对我们将要说的话很熟悉，因为我们从他们的定义和解释中大量借用。 领域驱动设计（DDD）是帮助我们成功理解和构建软件模型设计的一种方法。 它为我们提供了战略和战术建模工具，以帮助设计符合我们业务目标的高质量软件。



本书的主要目标是向您展示PHP代码示例的域驱动设计战术模式。 如果您想了解更多关于战略模式和主要领域驱动设计的知识，您应该阅读由Vaughn Vernon蒸馏的域驱动设计或由域驱动的设计参考：Eric Evans的定义和模式总结.

更重要的是，领域驱动设计不是关于技术。 相反，它是围绕商业发展知识并使用技术来提供价值。 只有当你有能力理解你的公司工作的业务时，你才能够参与到软件模型发现过程中来产生一种无处不在的语言.

## 为什么领域驱动设计很重要

软件不仅仅是代码。 如果你仔细想想，代码很少是我们职业的最终目标。 代码只是解决业务问题的媒介。 那么，为什么要谈论不同的语言呢？ 领域驱动设计强调确保企业和软件使用相同的语言。 一旦打破障碍，不需要翻译或繁琐的同步，信息不会丢失。 每个人都致力于发现业务领域，而不仅仅是编码员。 最终的软件是通用语言的唯一真理.

领域驱动设计还为策略和战术设计提供了一个框架 - 具有战略意义，能够根据业务价值确定最重要的开发领域，以及构建战场测试构建模块和模式的工作领域模型的策略.

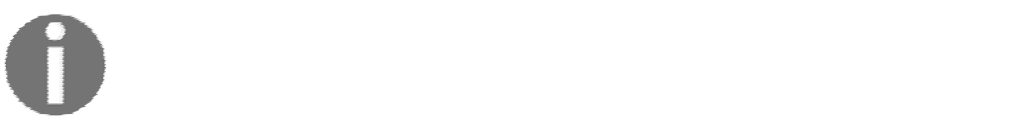
## 领域驱动设计的三大支柱

领域驱动设计是一种交付软件的方法，它着重于三大支柱:

1. **无处不在的语言：领域专家和软件开发人员共同为正在开发的业务领域构建通用语言。 没有我们与他们相对; 它总是我们。 开发软件是商业投资，而不仅仅是成本。 构建泛在语言所涉及的努力有助于在所有团队成员中传播深入的领域洞察力**.
2. **战略设计：领域驱动设计解决业务方向背后的策略，而不仅仅是技术方面。 它有助于确定内部关系和预警反馈系统。 在技术方面，战略设计通过为如何实现面向服务的体系结构提供动力来保护每项业务服务**.
3. **战术设计：领域驱动设计为迭代软件交付提供了工具和构建模块。 战术设计工具产生的软件不仅是正确的，而且也是可测试的，且不易出错**.

**无处不在的语言**

除了第12章集成有界上下文外，无处不在的语言是领域驱动设计的主要优势之一.



**根据上下文**

*就目前而言，认为有界上下文是围绕系统的概念边界。 边界内无所不在的语言具有特定的语境意义。 在这种情况之外的概念可以有不同的含义.*

所以，如何找到，探索和捕捉这种非常特殊的语言，下面的指针会突出显示这一点:

确定关键的业务流程，其投入和产出

创建术语和定义的术语表

用某种文档捕捉重要的软件概念

与其他团队分享并扩展所收集的知识（开发人员和领域专家）

自从领域驱动设计诞生以来，出现了改进泛在语言构建过程的新技术。 现在经常使用的最重要的事件是事件风暴.

#### 事件风暴

Alberto Brandolini在一篇博客文章中解释了Event Storming及其优点，他的描述比我们的要简单得多。Event Storming是一种快速探索复杂业务领域的研讨会形式:

它功能强大：它让我和许多从业者能够在几个小时而不是几个星期内提出一个完整业务流程的综合模型.

它是有吸引力的：整个想法是将问题和人员带到人们的面前

谁知道在同一个房间里的答案，并一起建立一个模型.

它非常高效：结果模型完全符合领域驱动设计实现方式（特别适合采用事件源方法），并且可以快速确定上下文和聚合边界.

很简单：符号非常简单。 没有复杂的UML可能会使讨论中的参与者截断.

这很有趣：我一直在领导研讨会很愉快，人们充满活力，交付的时间比预期的要多。 正确的问题出现了，气氛是正确的.

如果你想了解更多关于暴力事件的信息，请查看Brandolini的书，介绍EventStorming.

## 考虑领域驱动设计

领域驱动设计不是一个银弹; 与软件中的所有内容一样，这取决于上下文。 作为一个经验法则，使用它来简化您的域，但从不增加更多的复杂性.

如果您的应用程序是以数据为中心的，并且您的用例主要处理数据库中的行并执行CRUD操作（即创建，读取，更新和删除），则不需要域驱动设计。 相反，你公司唯一需要的是在你的数据库前面有一个奇特的面孔.

如果您的应用程序少于30个用例，那么使用像Symfony或Laravel这样的框架来处理您的业务逻辑可能会更简单.

然而，如果你的应用程序有超过30个用例，你的系统可能会转向可怕的大泥球。 如果您确定系统的复杂性会增加，那么您应该考虑使用领域驱动设计来应对这种复杂性.

如果您知道您的应用程序将会增长并且可能经常发生变化，那么Domain-Driven Design肯定会帮助您管理复杂性并随着时间的推移重构您的模型.

如果您不理解您正在开发的域名，因为它是新的，而且之前没有人投资过解决方案，这可能意味着您开始应用域驱动设计已足够复杂。 在这种情况下，您需要与域名专家密切合作才能正确使用模型.

## 棘手的部分

应用领域驱动设计并不容易。 它需要时间和精力来绕过业务领域，术语，研究和与领域专家的合作，而不是编码术语。 您需要承担领域专家的参与才能参与这个过程。 这需要开放和健康的持续对话来将他们的口语模型化为软件。 最重要的是，我们必须努力避免技术上的思考，认真思考对象的行为和无处不在的语言.

## Strategical Overview

In order to provide a general overview of the strategical side of Domain-Driven Design, we'll use an approach from *Jimmy Nilsson's* book, [Applying Domain-Driven Design and](https://www.amazon.com/Applying-Domain-Driven-Design-Patterns-Examples/dp/0321268202) [Patterns](https://www.amazon.com/Applying-Domain-Driven-Design-Patterns-Examples/dp/0321268202). Consider two different spaces: the problem space and the solution space.

In the problem space, Domain-Driven Design uses Domains and Subdomains to group and organize what companies want to solve. In the case of an **Online Travel Agency** (**OTA**), the problem is about dealing with things like flight tickets and booking hotels. Such a Domain can be organized into different Subdomains such as Pricing, Inventory, User Management, and so on.

In the solution space, Domain-Driven Design provides two patterns: Bounded Contexts and Context Maps. The goal is to define how to provide an implementation to all the identified Subdomains by defining their interactions and the details of those interactions. Continuing with the OTA example, each of the Subdomains will be solved with a Bounded Context implementation — for example, consider a custom Web Application developed by a team for the Pricing Management Subdomain, and an off-the-shelf solution for the User Management Subdomain. The Context Map will show how each Bounded Context is related to the rest. Inside the Context Map, we can see what type of relation two Bounded Contexts have (example: customer-supplier, partners). The ideal approach is to have each Subdomain implemented by one Bounded Context, but that's not always possible. In terms of implementation, when following Domain-Driven Design, you'll end up with distributed architectures. As you may already know, distributed architectures are more complex than monolithic ones, so why is this approach interesting, especially for big and complex companies? Is it really worth it? Well, it is.

Distributed architectures are proven to increase overall company productivity because they define boundaries for your product that can be developed by focused teams.

If your Domain — the problem you need to solve — is not complex, applying the strategical part of Domain-Driven Design can add unnecessary overhead and slow down your development speed.

If you want to know more about the strategical part of Domain-Driven Design, you should take a look at the first three chapters of *Vaughn Vernon's* book, [Implementing Domain-](http://www.amazon.com/Implementing-Domain-Driven-Design-Vaughn-Vernon-ebook/dp/B00BCLEBN8) [Driven Design](http://www.amazon.com/Implementing-Domain-Driven-Design-Vaughn-Vernon-ebook/dp/B00BCLEBN8), or the book [Domain-Driven Design: Tackling Complexity in the Heart](http://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215)) [of Software](http://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215)) by *Eric Evans*, both of which specifically focus on this aspect.

## Related Movements: Microservices and Self- Contained Systems

There are other movements promoting architectures that follow the same principles Domain-Driven Design is promoting. Microservices and Self-Contained Systems are good examples of this. *James Lewis* and *Martin Fowler* define Microservices in the [Microservices](http://martinfowler.com/microservices/) [Resource Guide](http://martinfowler.com/microservices/):

The Microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and are also independently deployable using fully automated machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and also use different data storage technologies.

If you want to know more about Microservices, their guide is a good place to start.How is this related to Domain-Driven Design? As explained in *Sam Newman's* book, [Building](http://www.amazon.com/Building-Microservices-Sam-Newman/dp/1491950358) [Microservices](http://www.amazon.com/Building-Microservices-Sam-Newman/dp/1491950358), Microservices are implementations of Domain-Driven Design Bounded Contexts.

In addition to Microservices, another related movement is **Self-Contained Systems** (**SCS**). According to the [Self-Contained Systems](http://scs-architecture.org/) website:

The Self-contained System approach is an architecture that focuses on a separation of the functionality into many independent systems, making the complete logical system a collaboration of many smaller software systems. This avoids the problem of large monoliths that grow constantly and eventually become unmaintainable.

Over the past few years, we have seen its benefits in many mid-sized and large- scale projects. The idea is to break a large system apart into several smaller self- contained system, or SCSs, that follow certain rules.

The website also spells out seven characteristics of SCS:

Each SCS is an autonomous web application. For the SCS's domain all data, the logic to process that data and all code to render the web interface is contained within the SCS. An SCS can fulfill its primary use cases on its own, without having to rely on other systems being available.

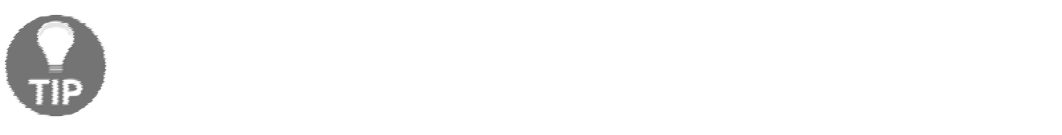
Each SCS is owned by one team. This does not necessarily mean that only one team might change the code, but the owning team has the final say on what goes into the code base, for example by merging pull-requests.

Communication with other SCSs or 3rd party systems is asynchronous wherever possible. Specifically, other SCSs or external systems should not be accessed synchronously within the SCS's own request/response cycle. This decouples the systems, reduces the effects of failure, and thus supports autonomy. The goal is decoupling concerning time: An SCS should work even if other SCSs are temporarily offline. This can be achieved even if the communication on the technical level is synchronous, example by replicating data or buffering requests.

An SCS can have an optional service API. Because the SCS has its own web UI it can interact with the user — without going through a UI service. However, an API for mobile clients or for other SCSs might still be useful.

Each SCS must include data and logic. To really implement any meaningful features both are needed. An SCS should implement features by itself and must therefore include both.

An SCS should make its features usable to end-users by its own UI. Therefore the SCS should have no shared UI with other SCSs. SCSs might still have links to each other. However, asynchronous integration means that the SCS should still work even if the UI of another SCS is not available. To avoid tight coupling an SCS should share no business code with other SCSs. It might be fine to create a pull- request for an SCS or use common libraries, example: database drivers or oAuth clients.



**Exercise**

Discuss the pros and cons of such distributed architectures with your workmates. Think about using different languages, deployment processes, infrastructure responsibilities, and so on.

## Wrap-Up

During this chapter you've learned:

Domain-Driven Design is not about technology; it's actually about providing value in the field you're working in by focusing on the model. Everyone takes part in the process of discovering the Domain, and developers and Domain Experts team up to build the knowledge base by sharing the same language, the Ubiquitous Language.

Domain-Driven Design provides tactical and strategic modeling tools to design high-quality software. Strategic design targets the business direction, helps in defining the internal relationships, and technically protects each business service by defining strong boundaries. Tactical design provides useful building blocks for iterative design.

Domain-Driven Design only makes sense in certain contexts. It's not a silver bullet for every problem in software, so whether or not you use it highly depends on the amount of complexity you're dealing with.

Domain-Driven Design is a long-term investment; it requires active effort. Domain Experts will be required to collaborate closely with developers, and developers will have to think in terms of the business. In the end, the business customer is the one who has to be pleased.

Implementing Domain-Driven Design requires effort. If it were easy, everybody would be writing high-quality code. Get ready, because you'll soon learn how to write code that, when read, will perfectly describe the business your company operates on. Enjoy this journey!

# 建筑风格

****

In order to be able to build complex applications, one of the key requirements is having an architectural design that fits the application's needs. One advantage of Domain-Driven Design is that it's not tied to any particular architecture style. Instead, we're free to choose the architecture that best fits the needs of every Bounded Context inside the Core Domain, which offers a diverse set of architectural choices for every specific Domain problem.

For example, an Order Processing System can use Event Sourcing to track all the different order operations; a Product Catalog can use CQRS to expose the product details to the different clients; and a Content Management System can use plain Hexagonal Architecture to expose requirements such as blogs, static pages, and so on.

This chapter presents an introduction to every relevant architecture style in the land of PHP, following the evolution from traditional old school PHP code to a more sophisticated architecture. Please note that although there are many other existing architecture styles, such as Data Fabric or SOA, we found some of them a bit too complex to introduce from the PHP perspective.

## The Good Old Days

Before the release of PHP 4, the language didn't embrace the Object-Oriented paradigm. Back then, the usual way of writing applications was by using procedures and global state. Concepts like **Separation of Concerns** (**SoC**) and **Model-View-Controller** (**MVC**) were alien among the PHP community.

下面的例子是以这种传统方式编写的应用程序，其中应用程序由许多与HTML代码混合的前端控制器组成。 在此期间，基础设施，演示文稿，UI和域层代码都被纠缠在一起：

include DIR

. '/bootstrap.php';

$link = mysql\_connect('localhost', 'a\_username', '4\_p4ssw0rd');

if (!$link) {

die('Could not connect: ' . mysql\_error());

}

mysql\_set\_charset('utf8', $link); mysql\_select\_db('my\_database', $link);

$errormsg = null ;

if (isset($\_POST['submit'] && isValid($\_POST['post'])) {

$post = getFrom($\_POST['post']); mysql\_query('START TRANSACTION', $link);

$sql = sprintf(

"INSERT INTO posts (title, content) VALUES ('%s','%s')", mysql\_real\_escape\_string($post['title']), mysql\_real\_escape\_string($post['content']

));

$result = mysql\_query($sql, $link); if ($result) {

mysql\_query('COMMIT', $link);

} else {

mysql\_query('ROLLBACK', $link);

$errormsg = 'Post could not be created! :(';

}

}

$result = mysql\_query('SELECT id, title, content FROM posts', $link);

>

<html>

<head></head>

<body>

<php if (null !== $errormsg) : >

<div class="alert error"><php echo $errormsg; ></div>

<php else: >

<div class="alert success">

Bravo! Post was created successfully!

</div>

<php endif; >

<table>

<thead><tr><th>ID</th><th>TITLE</th>

<th>ACTIONS</th></tr></thead>

<tbody>

<php while($post = mysql\_fetch\_assoc($result)) : >

<tr>

<td><php echo $post['id']; ></td>

<td><php echo $post['title']; ></td>

<td><php editPostUrl($post['id']); ></td>

</tr>

<php endwhile; >

</tbody>

</table>

</body>

</html>

<php mysql\_close($link); >

这种编码方式通常被称为第一章中提到的大泥球。 但是，这种风格的改进是将网页的页眉和页脚封装在各自的单独文件中，这些文件包含在页眉和页脚文件中。 这避免了重复和有利的重用：

include DIR

. '/bootstrap.php';

$link = mysql\_connect('localhost', 'a\_username', '4\_p4ssw0rd');

if (!$link) {

die('Could not connect: ' . mysql\_error());

}

mysql\_set\_charset('utf8', $link); mysql\_select\_db('my\_database', $link);

$errormsg = null;

if (isset($\_POST['submit'] && isValid($\_POST['post'])) {

$post = getFrom($\_POST['post']); mysql\_query('START TRANSACTION', $link);

$sql = sprintf(

"INSERT INTO posts(title, content) VALUES('%s','%s')", mysql\_real\_escape\_string($post['title']), mysql\_real\_escape\_string($post['content'])

);

$result = mysql\_query($sql, $link); if ($result) {

mysql\_query('COMMIT', $link);

} else {

mysql\_query('ROLLBACK', $link);

$errormsg = 'Post could not be created! :(';

}

}

$result = mysql\_query('SELECT id, title, content FROM posts', $link);

>

<php include DIR

. '/header.php'; >

<php if (null !== $errormsg) : >

<div class="alert error"><php echo $errormsg; ></div>

<php else: >

<div class="alert success">

Bravo! Post was created successfully!

</div>

<php endif; >

<table>

<thead>

<tr>

<th>ID</th>

<th>TITLE</th>

<th>ACTIONS</th>

</tr>

</thead>

<tbody>

<php while($post = mysql\_fetch\_assoc($result)): >

<tr>

<td><php echo $post['id']; ></td>

<td><php echo $post['title']; ></td>

<td><php editPostUrl($post['id']); ></td>

</tr>

<php endwhile; >

</tbody>

</table>

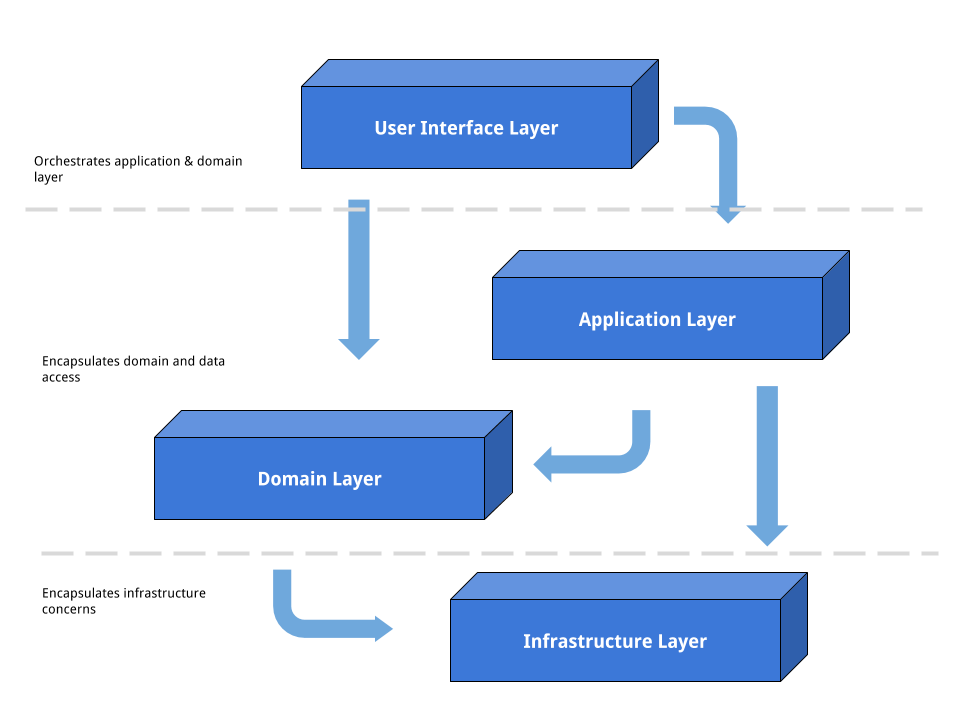
<php include DIR

. '/footer.php'; >

现在，尽管它非常沮丧，但仍然有应用程序使用这种程序编码方式。 这种风格的架构的主要缺点是没有真正的关注点分离 - 以这种方式开发应用程序的维护和成本与其他知名和经过验证的架构相比急剧增加.

## 分层架构

从代码可维护性和重用的角度来看，让代码更容易维护的最好方法是分解概念，即为每个不同的问题创建图层。 在我们之前的例子中，很容易形成不同的层次：一个用于封装数据访问和操作，另一个用于处理基础架构问题，最后一个用于封装前两者的编排。 分层体系结构的基本规则是每层必须与其下面的图层紧密耦合，如下图所示:



Layered Architecture for SoC

分层架构真正寻求的是分离应用程序的不同组件。 例如，就前面的示例而言，博客帖子表示必须完全独立于作为概念实体的博客文章。 作为概念实体的博客文章可以与一个或多个表示相关联，而不是与特定表示紧密结合。 这通常被称为分离关注.

寻求相同目的的另一种架构范例和模式是模型 - 视图 - 控制器模式。 它最初被认为是广泛用于构建桌面GUI应用程序，现在它主要用于Web应用程序，这要归功于Symfony，Zend Framework和CodeIgniter等流行的Web框架.

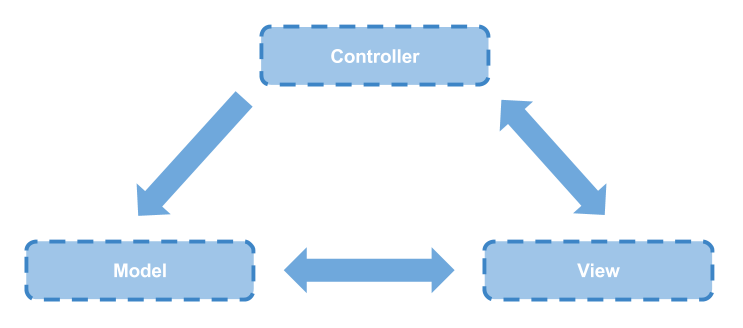
### 模型 - 视图 - 控制器

模型 - 视图 - 控制器是将应用程序分为三个主要层的体系结构模式和范例，在以下几点中进行描述:

**模型：捕获并集中所有域模型的行为。 该层独立于数据表示来管理所有数据，逻辑和业务规则。 据说模型层是每个MVC应用程序的核心和灵魂**.

**控制器：协调其他层之间的交互，在模型上触发动作以更新其状态，并刷新与模型相关的表示。 此外，控制器可以将消息发送到视图层以更改特定的模型表示**.

**视图：公开模型图层的不同表示形式，并提供触发模型状态变化的方法**.



The MVC pattern

### 分层架构示例

#### 模型(model)

继续前面的例子，我们提到不同的问题应该分解。 为了做到这一点，所有的图层都应该在我们最初的纠结代码中进行标识。

在整个过程中，我们需要特别注意符合Model层的代码，这将成为应用程序的跳动核心:

class Post

{

private $title; private $content;

public static function writeNewFrom($title, $content)

{

return new static($title, $content);

}

private function construct($title, $content)

{

$this->setTitle($title);

$this->setContent($content);

}

private function setTitle($title)

{

if (empty($title)) {

throw new RuntimeException('Title cannot be empty');

}

$this->title = $title;

}

private function setContent($content)

{

if (empty($content)) {

throw new RuntimeException('Content cannot be empty');

}

$this->content = $content;

}

}

class PostRepository

{

private $db;

public function construct()

{

$this->db = new PDO( 'mysql:host=localhost;dbname=my\_database', 'a\_username',

'4\_p4ssw0rd', [

PDO::MYSQL\_ATTR\_INIT\_COMMAND => 'SET NAMES utf8mb4',

]

);

}

public function add(Post $post)

{

$this->db->beginTransaction();

try {

$stm = $this->db->prepare(

'INSERT INTO posts (title, content) VALUES (, )'

);

$stm->execute([

$post->title(),

$post->content(),

]);

$this->db->commit();

} catch (Exception $e) {

$this->db->rollback();

throw new UnableToCreatePostException($e);

}

}

}

Model层现在由Post类和PostRepository类定义。 Post类表示博客文章，而PostRepository类表示可用博客文章的整个集合。 此外，模型内部还需要另一个层（协调和编排域模型行为的层）。 进入应用层：

class PostService

{

public function createPost($title, $content)

{

$post = Post::writeNewFrom($title, $content); (new PostRepository())->add($post);

return $post;

}

}

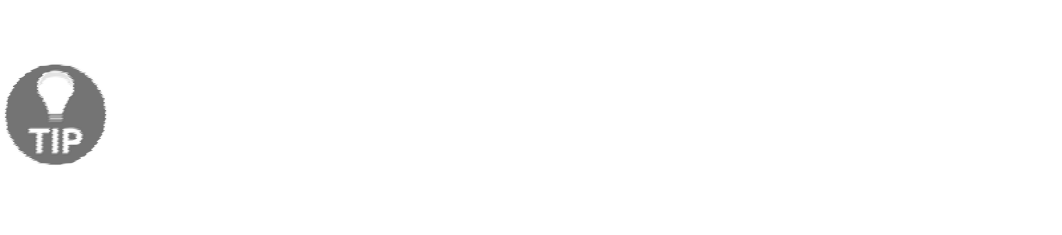
PostService类是所谓的应用程序服务，其目的是编排和组织域行为。 换句话说，应用服务是使事情发生的应用服务，它们是域模型的直接客户。 没有其他类型的对象应该能够直接与模型图层的内部图层进行对话.

**视图(view)**

该视图是一个既可以发送模型图层，又可以从控制器图层发送和接收消息的图层。 其主要目的是在UI级别向用户呈现模型，并在每次更新模型时刷新UI中的表示。

一般来说，View层接收一个对象 - 通常是一个数据传输

对象（DTO）而不是模型层的实例 - 从而收集所有需要的信息以成功表示。 对于PHP，有几个模板引擎可以帮助将模型表示从模型本身和控制器中分离出来。 目前最流行的一种叫做Twig。 让我们看看View层如何与Twig一起看.



**DTO而不是模型实例？**

**这是一个古老而活跃的话题。 为什么要创建一个DTO，而不是将模型的一个实例提供给View层？ 主要原因和简短答案同样是分离关注。 让View检查并使用Model实例会导致View层和Model层紧密耦合。 实际上，Model层的更改可能会破坏所有使用更改的Model实例的视图**.

{% extends "base.html.twig" %}

{% block content %}

{% if errormsg is defined %}

<div class="alert error">{{ errormsg }}</div>

{% else %}

<div class="alert success">

Bravo! Post was created successfully!

</div>

{% endif %}

<table>

<thead>

<tr>

<th>ID</th>

<th>TITLE</th>

<th>ACTIONS</th>

</tr>

</thead>

<tbody>

{% for post in posts %}

<tr>

<td>{{ post.id }}</td>

<td>{{ post.title }}</td>

<td><a href="{{ editPostUrl(post.id) }}">Edit Post</a></td>

</tr>

{% endfor %}

</tbody>

</table>

{% endblock %}

大多数情况下，当模型触发状态更改时，它还会通知相关视图，以便刷新UI。 在典型的Web场景中，由于客户端 - 服务器的性质，模型及其表示之间的同步可能有点棘手。 在这些类型的环境中，通常需要一些JavaScript定义的交互来维持同步。 出于这个原因，像下面那样的JavaScript MVC框架近年来已经广泛流行:

[AngularJS](https://angularjs.org/) [Ember.js](http://emberjs.com/) [Marionette.js](http://marionettejs.com/) [React](https://facebook.github.io/react/)

#### The Controller

Controller层负责组织和编排视图和模型。 它从View层接收消息并触发模型行为，以执行所需的操作。 此外，它将消息发送到视图以显示模型表示。 这两个操作都由应用层执行，它负责编排，组织和封装域行为.

就PHP中的Web应用程序而言，控制器通常会理解一组类，为了实现其目的，请“说出HTTP”。 换句话说，他们收到一个HTTP请求并返回一个HTTP响应：

class PostsController

{

public function updateAction(Request $request)

{

if (

) {

$request->request->has('submit') && Validator::validate($request->request->post)

$postService = new PostService();

try {

$postService->createPost(

$request->request->get('title'),

$request->request->get('content')

);

$this->addFlash( 'notice',

'Post has been created successfully!'

);

} catch (Exception $e) {

$this->addFlash( 'error',

'Unable to create the post!'

);

}

}

return $this->render('posts/update-result.html.twig');

}

}

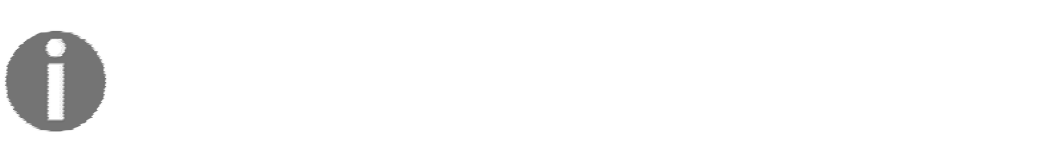
### 反向依赖：六角形建筑

遵循分层体系结构的基本规则，在实现包含基础设施问题的域接口时存在风险.

作为一个例子，在MVC中，前一个例子中的PostRepository类应该放在域模型中。 但是，将正确的基础设施细节置于我们域名的中间会违反关注的分离。 这可能是有问题的; 很难避免违反分层体系结构的基本规则，这会导致一种代码风格，如果域层意识到技术实现时可能难以测试.

#### 依赖倒置原则（DIP）

我们如何解决这个问题？ 由于域模型层依赖于具体的基础设施实现，所以可以通过将基础设施层重新定位在其他三层之上来应用依赖性反转原理或DIP.



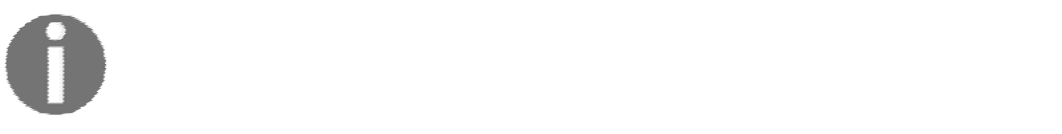
**依赖倒置原则**

高级模块不应该依赖于低级模块。 两者都应该依赖于抽象.

抽象不应该依赖细节。 细节应该取决于抽象。 罗伯特C.马丁

通过使用依赖性反转原则，体系结构模式发生变化，而基础结构层（可称为低级模块）现在取决于UI，应用层和Domain层（它们是高级别模块。 依赖性已被颠倒过来.

但是，什么是六角形建筑，它在这一切中如何适合？六角形建筑（也称为港口和适配器）由Alistair Cockburn在其着作Hexagonal Architecture中定义。它将应用程序描述为一个六边形，其中每一侧代表一个带有一个或多个适配器的端口。端口是一个带有可插入适配器的连接器，可将外部输入转换为内部应用程序可以理解的内容。在DIP方面，一个端口将是一个高级模块，而一个适配器将是一个低级模块。此外，如果应用程序需要向外部发送消息，它还将使用带有适配器的端口发送它并将其转换为外部可以理解的内容。因此，六角结构在应用中提出了对称性的概念，也是架构模式发生变化的主要原因。它通常表示为六角形，因为谈论顶层或底层不再有意义。相反，六角形建筑主要从外部和内部进行谈判。



Matthias Noback在YouTube上有很棒的视频，他谈到了六角形建筑。 你可能想看看其中一个更详细的信息.

#### 应用六角形建筑

继续使用博客示例应用程序，我们需要的第一个概念是外部世界可以与应用程序交谈的端口。 在这种情况下，我们将使用HTTP端口及其相应的适配器。 外部将使用端口向应用程序发送消息。 该博客示例使用数据库来存储整个博客文章的集合，因此为了允许应用程序从数据库中检索博客文章，需要一个端口:

interface PostRepository

{

public function byId(PostId $id); public function add(Post $post);

}

该接口公开了应用程序将通过检索关于博客帖子的信息的端口，并且该端口将位于域层中。 现在需要此端口的适配器。 适配器负责定义使用特定技术检索博客文章的方式:

class PDOPostRepository implements PostRepository

{

private $db;

public function construct(PDO $db)

{

$this->db = $db;

}

public function byId(PostId $id)

{

$stm = $this->db->prepare(

'SELECT \* FROM posts WHERE id = '

);

$stm->execute([$id->id()]);

return recreateFrom($stm->fetch());

}

public function add(Post $post)

{

$stm = $this->db->prepare(

'INSERT INTO posts (title, content) VALUES (, )'

);

$stm->execute([

$post->title(),

$post->content(),

]);

}

}

一旦我们定义了端口及其适配器，最后一步就是重构PostService

类，以便它使用它们。 这可以通过使用依赖注入轻松实现:

class PostService

{

private $postRepository;

public function construct(PostRepositor $postRepository)

{

$this->postRepository = $postRepository;

}

public function createPost($title, $content)

{

$post = Post::writeNewFrom($title, $content);

$this->postRepository->add($post);

return $post;

}

}

这只是六角形建筑的一个简单例子。 这是一个灵活的架构，可促进分散关注，如分层架构。 由于具有通过端口与外部通信的内部应用程序，因此它还促进了对称性。 从现在开始，这将是用于构建和解释CQRS和事件采购的基础架构.

有关此体系结构的更多示例，可以查看附录，PHP中的六角形体系结构。 有关更详细的示例，您应该跳到第11章应用程序，它解释了事务性和其他横切关注点等高级主题.

### 命令查询责任分离（CQRS）

六角形体系结构是一个很好的基础体系结构，但它有一些局限性。 例如，复杂的用户界面可能需要以多种形式（第8章，聚合）显示聚合信息，或者他们可能需要从多个聚合中获取数据。 在这种情况下，我们最终可能会在存储库中找到很多finder方法（可能与应用程序中存在的UI视图数量相同）。 或者，也许我们可以决定将这种复杂性转移到应用程序服务，使用复杂的结构来累积来自多个聚合的数据。 这是一个例子:

interface PostRepository

{

public function save(Post $post); public function byId(PostId $id); public function all();

public function byCategory(CategoryId $categoryId); public function byTag(TagId $tagId);

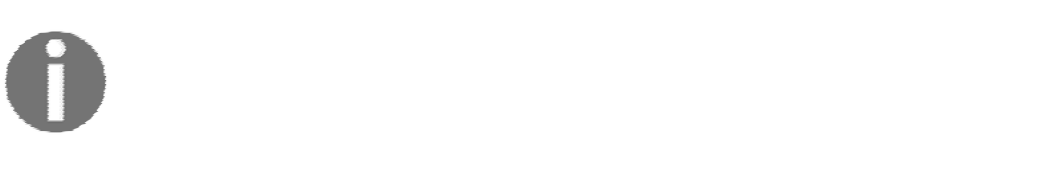
public function withComments(PostId $id); public function groupedByMonth();

// ...

}

当这些技术被滥用时，构建UI视图会变得非常痛苦。 我们应该评估应用服务返回域模型实例和返回某种DTO之间的权衡。 通过后一种选择，我们避免了域模型和基础架构代码（Web控制器，CLI控制器等）之间的紧密耦合，.

幸运的是，还有另一种方法。 如果问题存在多个不同视图，我们可以将它们从域模型中排除，并开始将它们视为纯粹的基础结构问题。 该选项基于设计原则，命令查询分离（CQS）。 这一原则由Bertrand Meyer定义，并且反过来，它催生了一种新的架构模式，命名查询责任隔离（CQRS），由Greg Young.



**命令查询分离（CQS）**

*提出一个问题不应该改变答案 - Bertrand Meyer这个设计原则规定每种方法都应该是a*

*执行操作的命令或将数据返回给调用者的查询，但不是维基百科*

CQRS寻求更加激进的关注分离，将模型分成两部分:

写模型：也称为命令模型，它执行写操作并负责真正的域行为.

读取模型：它负责读取应用程序中的内容，并将它们视为应该超出域模型的内容。

每当有人触发Write Model的命令时，就会执行写入所需数据存储的操作。 此外，它还会触发Read Model更新，以便在Read Model上显示最新的更改。

严格的分离会导致另一个问题：最终一致性。 Read Model的一致性现在受写模型执行的命令的限制。 换句话说，读取模型最终是一致的。 也就是说，每当写模型执行一个命令时，它将根据写模型上的最新更新拉起一个负责更新读模型的进程。 有一段时间，用户界面可能会向用户显示陈旧的信息。 在网络场景中，这经常发生，因为我们受当前技术的限制。

考虑一下Web应用程序前面的缓存系统。 每当用新信息更新数据库时，缓存层上的数据可能会过时，所以每次更新时都应该有一个更新缓存系统的进程。 缓存系统最终是一致的.

这些以CQRS术语讲话的过程称为写模型预测，或者只是预测。 我们将写模型投影到读模型上。 这个过程可以是同步的也可以是异步的，具体取决于你的需要，这可以归功于另一种有用的战术设计模式 - 章节域名事件 - 本书稍后将详细解释。 写模型预测的基础是收集所有已发布的域事件，并使用来自事件的所有信息更新读模型.

#### 写模型

这是Domain行为的真正持有者。 继续我们的例子，Repository接口将被简化为以下内容:

interface PostRepository

{

public function save(Post $post); public function byId(PostId $id);

}

现在PostRepository已经从所有的阅读关注中解放出来，除了一个：The

byId函数负责通过ID加载Aggregate，以便我们可以对其进行操作。 一旦完成，所有查询方法也从Post模型中剥离下来，只留下命令方法。 这意味着我们将有效地摆脱所有getter方法和任何其他暴露Post Aggregate信息的方法。 相反，域名事件将被发布，以便能够通过订阅它们来触发写模型预测:

class AggregateRoot

{

private $recordedEvents = [];

protected function

recordApplyAndPublishThat( DomainEvent $domainEvent

) {

$this->recordThat($domainEvent);

$this->applyThat($domainEvent);

$this->publishThat($domainEvent);

}

protected function recordThat(DomainEvent $domainEvent)

{

$this->recordedEvents[] = $domainEvent;

}

protected function applyThat(DomainEvent $domainEvent)

{

$modifier = 'apply' . get\_class($domainEvent);

$this->$modifier($domainEvent);

}

protected function publishThat(DomainEvent $domainEvent)

{

DomainEventPublisher::getInstance()->publish($domainEvent);

}

public function recordedEvents()

{

return $this->recordedEvents;

}

public function clearEvents()

{

$this->recordedEvents = [];

}

}

class Post extends AggregateRoot

{

private $id; private $title; private $content;

private $published = false; private $categories;

private function construct(PostId $id)

{

$this->id = $id;

$this->categories = new Collection();

}

public static function writeNewFrom($title, $content)

{

$postId = PostId::create();

$post = new static($postId);

$post->recordApplyAndPublishThat(

new PostWasCreated($postId, $title, $content)

);

}

public function publish()

{

$this->recordApplyAndPublishThat( new PostWasPublished($this->id)

);

}

public function categorizeIn(CategoryId $categoryId)

{

$this->recordApplyAndPublishThat(

new PostWasCategorized($this->id, $categoryId)

);

}

public function changeContentFor($newContent)

{

$this->recordApplyAndPublishThat(

new PostContentWasChanged($this->id, $newContent)

);

}

public function changeTitleFor($newTitle)

{

$this->recordApplyAndPublishThat(

new PostTitleWasChanged($this->id, $newTitle)

);

}

}

所有触发状态更改的操作均通过域事件实现。 对于发布的每个域事件，都有一个应用方法负责反映状态更改:

class Post extends AggregateRoot

{

// ...

protected function applyPostWasCreated( PostWasCreated $event

) {

$this->id = $event->id();

$this->title = $event->title();

$this->content = $event->content();

}

protected function applyPostWasPublished( PostWasPublished $event

) {

$this->published = true;

}

protected function applyPostWasCategorized( PostWasCategorized $event

) {

$this->categories->add($event->categoryId());

}

protected function applyPostContentWasChanged( PostContentWasChanged $event

) {

$this->content = $event->content();

}

protected function applyPostTitleWasChanged( PostTitleWasChanged $event

) {

$this->title = $event->title();

}

}

#### 读取模型

读取模型也称为查询模型，是一个从域关注中解除的纯粹的非规范化数据模型。 事实上，对于CQRS，所有读取的问题都被视为报告过程，这是一个基础设施问题。 一般来说，使用CQRS时，读取模型需要满足用户界面的需求以及复合UI的视图的复杂程度。 在根据关系数据库定义读取模型的情况下，最简单的方法是在数据库表和UI视图之间设置一对一的关系。 这些数据库表和UI视图将使用从写入侧发布的域事件触发的写入模型预测进行更新：

-- Definition of a UI view of a single post with its comments CREATE TABLE single\_post\_with\_comments (

id INTEGER NOT NULL,

post\_id INTEGER NOT NULL, post\_title VARCHAR(100) NOT NULL, post\_content TEXT NOT NULL, post\_created\_at DATETIME NOT NULL, comment\_content TEXT NOT NULL

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4\_unicode\_ci;

-- Set up some data

INSERT INTO single\_post\_with\_comments VALUES

|  |  |  |
| --- | --- | --- |
| (1, | 1, | "Layered" , "Some content", NOW(), "A comment"), |
| (2, | 1, | "Layered" , "Some content", NOW(), "The comment"), |
| (3, | 2, | "Hexagonal" , "Some content", NOW(), "No comment"), |
| (4, | 2, | "Hexagonal", "Some content", NOW(), "All comments"), |
| (5, | 3, | "CQRS", "Some content", NOW(), "This comment"), |
| (6, | 3, | "CQRS", "Some content", NOW(), "That comment"); |

-- Query it

SELECT \* FROM single\_post\_with\_comments WHERE post\_id = 1;

这种架构风格的一个重要特征是，由于应用程序的真实状态是由写模型处理的，所以读模型应该完全一次性使用。 这意味着使用写模型投影可以在需要时删除和重新创建读模型.

在这里，我们可以在博客应用程序中看到一些可能的视图示例:

SELECT \* FROM

posts\_grouped\_by\_month\_and\_year ORDER BY month DESC,year ASC;

SELECT \* FROM

posts\_by\_tags WHERE tag = "ddd";

SELECT \* FROM

posts\_by\_author WHERE author\_id = 1;

需要指出的是，CQRS不会将读取模型的定义和实现限制为关系数据库。 它完全取决于正在构建的应用程序的需求。 它可以是关系数据库，面向文档的数据库，键值存储或任何最适合您应用程序需求的内容。 在博客文章应用之后，我们将使用Elasticsearch--一个面向文档的数据库 - 来实现读取模式l:

class PostsController

{

public function listAction()

{

$client = new ElasticsearchClientBuilder::create()->build();

$response = $client-> search([ 'index' => 'blog-engine', 'type' => 'posts',

'body' => [

'sort' => [

'created\_at' => ['order' => 'desc']

]

]

]);

return [

'posts' => $response

];

}

}

Read Model代码已经大幅简化为针对Elasticsearch索引的单个查询。

这表明Read Model不需要一个对象关系映射器，因为这可能是矫枉过正的。 但是，写模型可能会受益于使用对象关系映射器，因为这样可以根据应用程序的需要组织和构建读模型.

#### 将写入模型与读取模型同步

棘手的部分来了。 我们如何将读取模型与写入模型同步？ 我们已经表示，我们将通过使用写模型事务中捕获的域事件来完成此任务。 对于捕获的每种类型的域事件，都会执行特定的投影。 因此，域名事件和投影之间的一对一关系将被设置.

让我们来看一个配置投影的例子，以便我们可以得到一个更好的想法。 首先，我们需要为投影定义一个框架:

interface Projection

{

public function listensTo(); public function project($event);

}

因此，为PostWasCreated事件定义Elasticsearch投影将如此简单:

namespace Infrastructure\Projection\Elasticsearch;

use Elasticsearch\Client; use PostWasCreated;

class PostWasCreatedProjection implements Projection

{

private $client;

public function construct(Client $client)

{

$this->client = $client;

}

public function listensTo()

{

return PostWasCreated::class;

}

public function project($event)

{

$this->client->index([ 'index' => 'posts', 'type' => 'post',

'id' => $event->getPostId(), 'body' => [

'content' => $event->getPostContent(),

// ...

]

]);

}

}

投影仪实现是一种专门的域事件监听器。 它和默认的域事件监听器之间的主要区别在于投影机对一组域事件作出反应，而不是一个：

namespace Infrastructure\Projection;

class Projector

{

private $projections = [];

public function register(array $projections)

{

foreach ($projections as $projection) {

$this->projections[$projection->eventType()] = $projection;

}

}

public function project( array $events)

{

foreach ($events as $event) {

if (isset($this->projections[get\_class($event)])) {

$this->projections[get\_class($event)]

->project($event);

}

}

}

}

以下代码显示投影机和事件之间的流程将如何显示:

$client = new ElasticsearchClientBuilder::create()->build();

$projector = new Projector();

$projector->register([

new Infrastructure\Projection\Elasticsearch\ PostWasCreatedProjection($client),

new Infrastructure\Projection\Elasticsearch\ PostWasPublishedProjection($client),

new Infrastructure\Projection\Elasticsearch\ PostWasCategorizedProjection($client),

new Infrastructure\Projection\Elasticsearch\ PostContentWasChangedProjection($client),

new Infrastructure\Projection\Elasticsearch\ PostTitleWasChangedProjection($client),

]);

$events = [

new PostWasCreated(/\* ... \*/), new PostWasPublished(/\* ... \*/),

new PostWasCategorized(/\* ... \*/), new PostContentWasChanged(/\* ... \*/), new PostTitleWasChanged(/\* ... \*/),

];

$projector->project($event);

此代码是同步的，但是如果需要，该进程可以是异步的。 通过在视图层中放置一些警报，您可以让客户意识到这种不同步的数据。

对于下一个示例，我们将结合使用amqplib PHP扩展和ReactPHP：

// Connect to an AMQP broker

$cnn = new AMQPConnection();

$cnn->connect();

// Create a channel

$ch = new AMQPChannel($cnn);

// Declare a new exchange

$ex = new AMQPExchange($ch);

$ex->setName('events');

$ex->declare();

// Create an event loop

$loop = ReactEventLoopFactory::create();

// Create a producer that will send any waiting messages every half a second

$producer = new Gos\Component\React\AMQPProducer($ex, $loop, 0.5);

$serializer = JMS\Serializer\SerializerBuilder::create()->build();

$projector = new AsyncProjector($producer, $serializer);

$events = [

new PostWasCreated(/\* ... \*/), new PostWasPublished(/\* ... \*/),

new PostWasCategorized(/\* ... \*/), new PostContentWasChanged(/\* ... \*/), new PostTitleWasChanged(/\* ... \*/),

];

$projector->project($event);

为此，我们需要一台异步投影仪。 这是一个天真的实现:

namespace Infrastructure\Projection;

use Gos\Component\React\AMQPProducer; use JMS\Serializer\Serializer;

class AsyncProjector

{

private $producer; private $serializer;

public function construct( Producer $producer, Serializer $serializer

) {

$this->producer = $producer;

$this->serializer = $serializer;

}

public function project(array $events)

{

foreach ($events as $event) {

$this->producer->publish(

$this->serializer->serialize(

$event, 'json'

)

);

}

}

}

而RabbitMQ交易所的事件消费者看起来就像这样:

// Connect to an AMQP broker

$cnn = new AMQPConnection();

$cnn-> connect();

// Create a channel

$ch = new AMQPChannel($cnn);

// Create a new queue

$queue = new AMQPQueue($ch);

$queue->setName('events');

$queue->declare();

// Create an event loop

$loop = React\EventLoop\Factory::create();

$serializer = JMS\Serializer\SerializerBuilder::create()->build();

$client = new Elasticsearch\ClientBuilder::create()->build();

$projector = new Projector();

$projector->register([

new Infrastructure\Projection\Elasticsearch\ PostWasCreatedProjection($client),

new Infrastructure\Projection\Elasticsearch\ PostWasPublishedProjection($client),

new Infrastructure\Projection\Elasticsearch\ PostWasCategorizedProjection($client),

new Infrastructure\Projection\Elasticsearch\ PostContentWasChangedProjection($client),

new Infrastructure\Projection\Elasticsearch\ PostTitleWasChangedProjection($client),

]);

// Create a consumer

$consumer = new Gos\Component\ReactAMQP\Consumer($queue, $loop, 0.5, 10);

// Check for messages every half a second and consume up to 10 at a time.

$consumer->on( 'consume',

function ($envelope, $queue) use ($projector, $serializer) {

$event = $serializer->unserialize($envelope->getBody(), 'json');

$projector->project($event);

}

);

$loop->run();

从现在开始，它可以像使所有需要的存储库使用投影机的一个实例，然后使其调用投影过程一样简单：

class DoctrinePostRepository implements PostRepository

{

private $em; private $projector;

public function construct(EntityManager $em, Projector $projector)

{

$this->em = $em;

$this->projector = $projector;

}

public function save(Post $post)

{

$this->em->transactional(

function (EntityManager $em) use ($post)

{

$em->persist($post);

foreach ($post->recordedEvents() as $event) {

$em->persist($event);

}

}

);

$this->projector->project($post->recordedEvents());

}

public function byId(PostId $id)

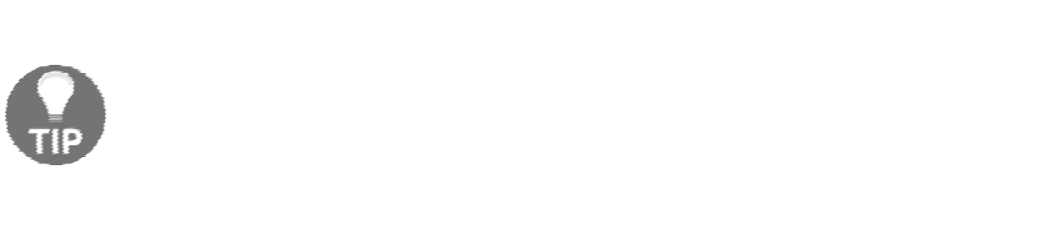
{

return $this->em->find($id);

}

}

Post实例和记录的事件在同一事务中被触发并保持。 这确保不会有事件丢失，因为如果事务成功，我们将它们投影到读取模型。 因此，写模型和读模型之间不存在矛盾.

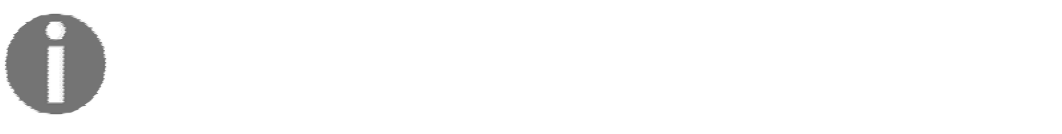


**要ORM还是不要ORM**

在实现CQRS时最常见的问题之一是如果真的需要一个对象关系映射器（ORM）。 我们坚信在写模型中使用ORM非常好，并具有使用工具的所有优点，这有助于我们在使用关系数据库的情况下节省大量工作。 但是我们不应该忘记，我们仍然需要在关系数据库中坚持并检索Write Model的状态.

## 事件采购(事件源)

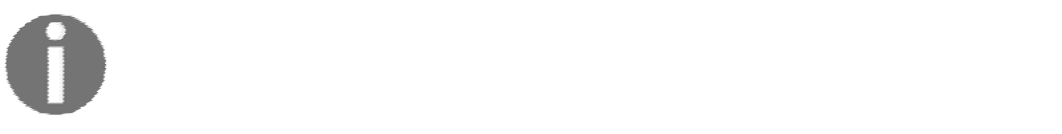
CQRS是一个强大而灵活的架构。 在收集和保存域事件（在聚合操作期间发生）方面，它还有一个额外的好处，可以让您详细了解域中发生的事情。 域名事件是关键战术模式之一，因为它们在域中有重要意义，因为它们描述了过去的事件.



**记录太多事件要小心**

越来越多的事件是一种气味。 它可能会揭示域上事件记录的上瘾，这很可能是由业务激励的。 作为一个经验法则，请记住保持简单.

通过使用CQRS，我们能够记录域层中发生的所有相关事件。 域模型的状态可以通过再现我们先前记录的域事件来表示。 我们只需要一个以一致的方式存储所有这些事件的工具。 我们需要一个活动商店.



事件采购背后的基本思想是将聚合的状态表示为一系列线性事件

使用CQRS，我们部分实现了以下功能：Post实体通过使用域事件来更改其状态，但它已被保留，如已解释的，从而将对象映射到数据库表.

事件采购更进一步。 如果我们使用数据库表来存储所有博客帖子的状态，使用另一个存储所有博客帖子评论的状态等等，使用事件源代码将允许我们使用单个数据库表：单个附件 - 唯一的数据库表，它将存储由域模型中的所有聚合发布的所有域事件。 是的，你没看错。 单个数据库表.

With this model in mind, tools like object-relational mappers are no longer needed. The only tool needed would be a simple database abstraction layer by which events can be appended:

interface EventSourcedAggregateRoot

{

public static function reconstitute(EventStream $events);

}

class Post extends AggregateRoot implements EventSourcedAggregateRoot

{

public static function reconstitute(EventStream $history)

{

$post = new static($history->getAggregateId());

foreach ($events as $event) {

$post->applyThat($event);

}

return $post;

}

}

现在，Post Aggregate有一种方法，当给定一组事件（或换句话说，一个事件流）时，它可以一步一步地重播状态，直到达到当前状态，所有这些都在保存之前完成。 下一步将构建PostRepository端口的适配器，该端口将从Post Aggregate中提取所有发布的事件，并将它们附加到所有事件都附加到的数据存储中。 这就是我们所说的活动商店:

class EventStorePostRepository implements PostRepository

{

private $eventStore; private $projector;

public function construct($eventStore, $projector)

{

$this->eventStore = $eventStore;

$this->projector = $projector;

}

public function save(Post $post)

{

$events = $post->recordedEvents();

$this->eventStore->append(new EventStream(

$post->id(),

$events)

);

$post->clearEvents();

$this->projector->project($events);

}

}

这是PostRepository的实现在我们使用事件存储库来保存Post Post Aggregate发布的所有事件时的外观。 现在我们需要一种方法来从其事件历史记录中恢复聚合。 由Post Aggregate实现并用于从触发事件重建博客帖子状态的重构方法派上用场:

class EventStorePostRepository implements PostRepository

{

public function byId(PostId $id)

{

return Post::reconstitute(

$this->eventStore->getEventsFor($id)

);

}

}

活动商店是履行保存和恢复活动流所有责任的主力。 其公共API由两个简单的方法组成：它们

是append和getEventsFrom。 前者将事件流附加到事件存储，后者加载事件流以允许聚合重建.

我们可以使用键值实现来存储所有事件:

class EventStore

{

private $redis; private $serializer;

public function construct($redis, $serializer)

{

$this->redis = $redis;

$this->serializer = $serializer;

}

public function append(EventStream $eventstream)

{

foreach ($eventstream as $event) {

$data = $this->serializer->serialize(

$event, 'json'

);

$date = (new DateTimeImmutable())->format('YmdHis');

$this->redis->rpush(

'events:' . $event->getAggregateId(),

$this->serializer->serialize([ 'type' => get\_class($event), 'created\_on' => $date, 'data' => $data

],'json')

);

}

}

public function getEventsFor($id)

{

$serializedEvents = $this->redis->lrange('events:' . $id, 0, -1);

$eventStream = [];

foreach($serializedEvents as $serializedEvent){

$eventData = $this->serializerdeserialize(

$serializedEvent, 'array',

'json'

);

$eventStream[] = $this->serializer->deserialize(

$eventData['data'],

$eventData['type'], 'json'

);

}

return new EventStream($id, $eventStream);

}

}

This event store implementation is built upon [Redis](http://redis.io/), a widely used key-value store. The events are appended in a list using the prefix events: In addition, before persisting the events, we extract some metadata like the event class or the creation date, as it will come in handy later.

Obviously, in terms of performance, it's expensive for an Aggregate to go over its full event history to reach its final state all of the time. This is especially the case when an event stream has hundreds or even thousands of events. The best way to overcome this situation is to take a snapshot from the Aggregate and replay only the events in the event stream that occurred after the snapshot was taken. A snapshot is just a simple serialized version of the Aggregate state at any given moment. It can be based on the number of events of the Aggregate's event stream, or it can be time based. With the first approach, a snapshot will be taken every *n* triggered events (every 50, 100, or 200 events, for example). With the second approach, a snapshot will be taken every *n* seconds.

To follow the example, we'll use the first way of snapshotting. In the event's metadata, we store an additional field, the version, from which we'll start replaying the Aggregate history:

class SnapshotRepository

{

public function byId($id)

{

$key = 'snapshots:' . $id;

$metadata = $this->serializer->unserialize(

$this->redis->get($key)

);

if (null === $metadata) { return;

}

return new Snapshot(

$metadata['version'],

$this->serializer->unserialize(

$metadata['snapshot']['data'],

$metadata['snapshot']['type'], 'json'

)

);

}

public function save($id, Snapshot $snapshot)

{

$key = 'snapshots:' . $id;

$aggregate = $snapshot->aggregate();

$snapshot = [

'version' => $snapshot->version(), 'snapshot' => [

'type' => get\_class($aggregate),

'data' => $this->serializer->serialize(

$aggregate, 'json'

)

]

];

$this->redis->set($key, $snapshot);

}

}

And now we need to refactor the EventStore class so that it starts using the

SnapshotRepository to load the Aggregate with acceptable performance times:

class EventStorePostRepository implements PostRepository

{

public function byId(PostId $id)

{

$snapshot = $this->snapshotRepository->byId($id);

if (null === $snapshot) { return Post::reconstitute(

$this->eventStore->getEventsFrom($id)

);

}

$post = $snapshot->aggregate();

$post->replay(

$this->eventStore->fromVersion($id, $snapshot->version())

);

return $post;

}

}

We just need to take Aggregate snapshots periodically. We could do this synchronously or asynchronously by a process responsible for monitoring the event store. The following code is a simple example demonstrating the implementation of Aggregate snapshotting:

class EventStorePostRepository implements PostRepository

{

public function save(Post $post)

{

$id = $post->id();

$events = $post->recordedEvents();

$post->clearEvents();

$this->eventStore->append(new EventStream($id, $events));

$countOfEvents =$this->eventStore->countEventsFor($id);

$version = $countOfEvents / 100;

if (!$this->snapshotRepository->has($post->id(), $version)) {

$this->snapshotRepository->save(

$id,

new Snapshot(

$post, $version

)

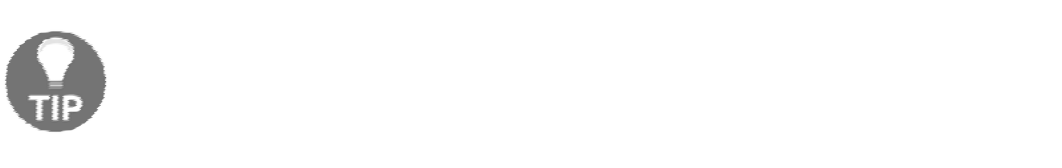
);

}

$this->projector->project($events);

}

}



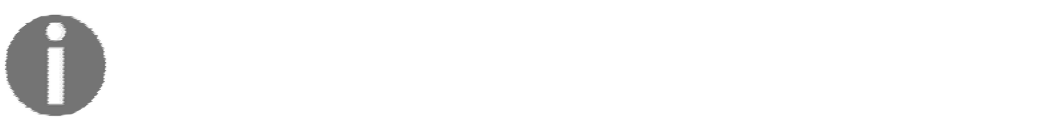
**To ORM or Not To ORM**

It's clear from the use case of this architectural style that using an ORM just to persist / fetch events would be overkill. Even if we use a relational database for storing them, we only need to persist / fetch events from the data store.

## Wrap-Up

As there are plenty of options for architectural styles, you may have gotten a bit confused in this chapter. You'll have to consider the tradeoffs for each one of them in order to choose wisely. One thing is clear: the Big Ball of Mud approach is not an option, as the code will rot pretty fast. Layered Architecture is a better option, but it presents some disadvantages, like tight coupling between layers. Arguably, the most balanced option would be Hexagonal Architecture, as it can be used as a foundational base architecture, and it promotes a high- level degree of decoupling and symmetry between the inside and outside of the application. This is what we recommend for most scenarios.

We've also seen CQRS and Event Sourcing as relatively flexible architectures that will help you in fighting serious complexity. CQRS and Event Sourcing both have their places, but don't let the *coolness factor* distract you from the value they provide. As they both come with some overhead, you should have a technical reason for justifying their use. These architectural styles are indeed really useful, and the heuristics to start using them can be discovered in the number of finders on the Repositories for CQRS and the volume of triggered events for Event Sourcing. If the number of finder methods starts growing and Repositories become difficult to maintain, then it's time to consider the use of CQRS, in order to split read and write concerns. And after that, if the volume of events on each Aggregate operation tends to grow and the business is interested in more granular information, then an option to consider is whether a move toward Event Sourcing might pay off.



**Extracted from a paper by Brian Foote and Joseph Yoder:**

*A BIG BALL OF MUD is haphazardly structured, sprawling, sloppy, duct-tape and bailing wire,* [spaghetti code jungle](http://www.laputan.org/mud/mud.html#BigBallOfMud).

# 值对象

****

通过使用self关键字，我们不需要Value对象是Domain-Driven Design的基本构建块，它们用于在代码中为无处不在的语言的概念建模。 价值对象不仅仅是你的领域中的一件东西 - 它测量，量化或描述一些东西。 价值对象可以被看作是小而简单的对象 - 例如金钱或日期范围 - 它们的平等不是基于身份，而是基于持有的内容.

例如，产品价格可以使用Value Object进行建模。 在这种情况下，它不代表一件事物，而是一种价值，它使我们能够衡量一件产品的价值。 这些对象的内存占用是微不足道的（由它们的组成部分计算），而且开销很小。 因此，即使用于表示相同的值，新实例创建也优于参考重用。 然后基于两个实例的字段的可比性来检查平等.

## 定义

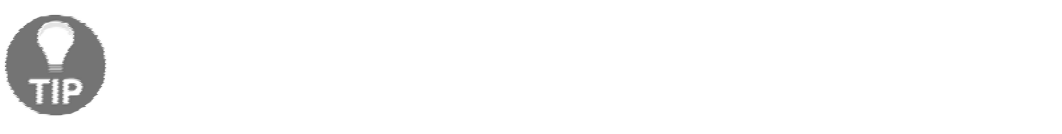
Ward Cunningham将价值对象定义为:

对某物的测量或描述。 值对象的例子是数字，日期，钱和字符串。 通常，它们是相当广泛使用的小物体。 他们的身份是基于他们的状态而不是他们的对象身份。 这样，您可以拥有同一个概念性值对象的多个副本。 每张5美元的钞票都有自己的身份（由于其序列号），但现金经济依靠每5美元钞票具有与每5美元钞票相同的价值.

马丁福勒定义一个价值对象为:

一个小对象，如Money或日期范围对象。 他们的关键特性是他们遵循价值语义而不是引用语义。 你通常可以告诉他们，因为他们的平等观念不是基于身份，相反，如果两个价值对象的所有领域都相等，那么两个价值对象是平等的。 尽管所有字段都相同，但如果子集是唯一的，则不需要比较所有字段 - 例如，货币对象的货币代码就足以测试相等性。 一般的启发式就是Value Objects应该完全不可变。 如果你想改变一个值对象，你应该用一个新对象替换对象，而不允许更新值对象本身的值 - 可更新值对象导致混叠问题.

值对象的例子是数字，文本字符串，日期，时间，人的全名（由名字，中间名，姓氏和标题组成），货币，颜色，电话号码和邮政地址.



**Exercise**

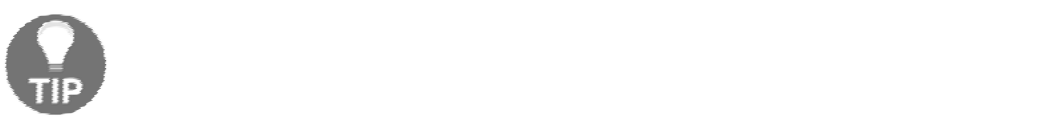
尝试在当前的域中查找更多潜在值对象的示例.

## 价值对象与实体

考虑维基百科的以下示例，以更好地理解值对象和实体之间的差异:

**价值对象：当人们交换美元钞票时，他们通常不会区分每个独特的钞票; 他们只关心美元钞票的面值。 在这种情况下，美元账单就是价值对象。 但是，美联储可能会关注每一个独特的法案; 在这种情况下，每个账单将是一个实体**.

**实体：大多数航空公司在每次航班上都独特地区分每个座位。 每个席位在这种情况下都是一个实体。 不过，西南航空，易捷航空和瑞安航空并没有区分每个座位; 所有座位都一样。 在这种情况下，一个席位实际上是一个价值对象。**



**Exercise**

考虑地址的概念（街道，号码，邮政编码等）。 什么是地址可能被建模为实体而不是值对象的可能上下文？ 与同行讨论你的发现.

## 货币和金钱示例

货币和金钱价值对象可能是解释价值对象最常用的例子，这要归功于Money模式。 此设计模式为避免浮点舍入问题的问题建模提供了一个解决方案，从而可以执行确定性计算.

在现实世界中，货币以与米和码描述距离单位相同的方式描述货币单位。 每种货币都用三个字母的大写ISO代码表示:

class Currency

{

private $isoCode;

public function construct($anIsoCode)

{

$this->setIsoCode($anIsoCode);

}

private function setIsoCode($anIsoCode)

{

if (!preg\_match('/^[A-Z]{3}$/', $anIsoCode)) { throw new InvalidArgumentException();

}

$this->isoCode = $anIsoCode;

}

public function isoCode()

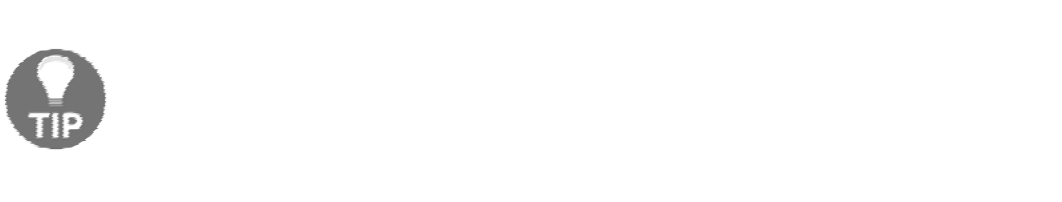
{

return $this->isoCode;

}

}

Value Objects的主要目标之一也是面向对象设计的圣杯：封装。 通过遵循这种模式，您将得到一个专门的位置来放置给定概念的所有验证，比较逻辑和行为.



**Extra Validations for Currency**

In the previous code example, we can build a Currency with an AAA Currency ISO code. That isn't valid at all. Write a more specific rule that will check if the ISO Code is valid. A full list of valid currency ISO codes can be found [here](http://www.xe.com/iso4217.php). If you need help, take a look at the [Money](https://github.com/moneyphp/money) packagist library.

Money is used to measure a specific amount of currency. It's modeled using an amount and a currency. Amount, in the case of the Money pattern, is implemented using an integer representation of the Currency's least-valuable fraction — For example in the case of USD or EUR, cents.

As a bonus, you might also notice that we're using [self encapsulation](http://martinfowler.com/bliki/SelfEncapsulation.html) to set the ISO code, which centralizes changes in the Value Object itself:

class Money

{

private $amount; private $currency;

public function construct($anAmount, Currency $aCurrency)

{

$this->setAmount($anAmount);

$this->setCurrency($aCurrency);

}

private function setAmount($anAmount)

{

$this->amount = (int) $anAmount;

}

private function setCurrency(Currency $aCurrency)

{

$this->currency = $aCurrency;

}

public function amount()

{

return $this->amount;

}

public function currency()

{

return $this->currency;

}

}

Now that you know the formal definition of Value Objects, let's dive deeper into some of the powerful features they offer.

## Characteristics

While modeling an Ubiquitous Language concept in code, you should always favor Value Objects over Entities. Value Objects are easier to create, test, use, and maintain.

Keeping this in mind, you can determine whether the concept in question can be modeled as a Value Object if:

It measures, quantifies, or describes a thing in the Domain It can be kept immutable

It models a conceptual whole by composing related attributes as an integral unit It can be compared with others through value equality

It is completely replaceable when the measurement or description changes It supplies its collaborators with side-effect-free behavior

### 量度，量化或描述

As discussed before, a Value Object should not be considered just a *thing* in your Domain. As a value, it measures, quantifies, or describes a concept in the Domain.

In our example, the Currency object describes what type of Money it is. The Money object measures or quantifies units of a given currency.

### 不变性

This is one of the most important aspects to grasp. Object values shouldn't be able to be altered over their lifetime. Because of this immutability, Value Objects are easy to reason and test and are free of undesired/unexpected side effects. As such, Value Objects should be created through their constructors. In order to build one, you usually pass the required primitive types or other Value Objects through this constructor.

Value Objects are always in a valid state; that's why we create them in a single atomic step. Empty constructors with multiple setters and getters move the creation responsibility to the client, resulting in the [Anemic Domain Model](http://www.martinfowler.com/bliki/AnemicDomainModel.html), which is considered an anti-pattern.

It's also good to point out that it's not recommended to hold references to Entities in your Value Objects. Entities are mutable, and holding references to them could lead to undesirable side effects occurring in the Value Object.

在具有方法重载的语言（如Java）中，可以创建具有相同名称的多个构造函数。 每个构造函数都提供了不同的选项来构建相同类型的结果对象。 在PHP中，我们可以通过工厂方法提供类似的功能。 这些特定的工厂方法也被称为语义构造函数。 fromMoney的主要目标是提供比普通构造函数更多的上下文含义。 更激进的方法建议将构造方法设为私有，并使用语义构造函数构建每个实例.

在我们的Money对象中，我们可以添加一些有用的工厂方法，如下所示:

class Money

{

// ...

public static function fromMoney(Money $aMoney)

{

return new self(

$aMoney->amount(),

$aMoney->currency()

);

}

public static function ofCurrency(Currency $aCurrency)

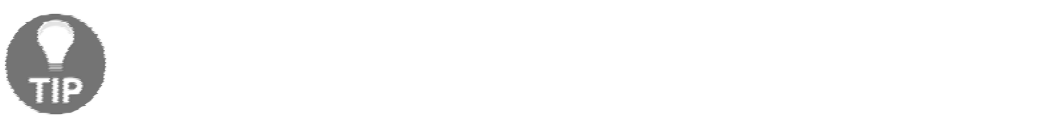
{

return new self(0, $aCurrency);

}

}

By using the self keyword, we don't couple the code with the class name. As such, a change to the class name or namespace won't affect these factory methods. This small implementation detail helps when refactoring the code at a later date.



**static vs. self**

Using static over self can result in undesirable issues when a Value Object inherits from another Value Object.

Due to this immutability, we must consider how to handle mutable actions that are common place in a stateful context. If we require a state change, we now have to return a brand new Value Object representation with this change. If we want to increase the amount of, for example, a Money Value Object, we're required to instead return a new Money instance with the desired modifications.

Fortunately, it's relatively simple to abide by this rule, as shown in the example below:

class Money

{

// ...

public function increaseAmountBy($anAmount)

{

return new self(

$this->amount() + $anAmount,

$this->currency()

);

}

}

The Money object returned by increaseAmountBy is different from the Money client object that received the method call. This can be observed in the example comparability checks below:

$aMoney = new Money(100, new Currency('USD'));

$otherMoney = $aMoney->increaseAmountBy(100); var\_dump($aMoney === otherMoney); // bool(false)

$aMoney = $aMoney->increaseAmountBy(100); var\_dump($aMoney === $otherMoney); // bool(false)

### 概念整体

So why not just implement something similar to the following example, avoiding the need for a new Value Object class altogether?

class Product

{

private id; private name;

/\*\*

\* @var int

\*/

private $amount;

/\*\*

\* @var string

\*/

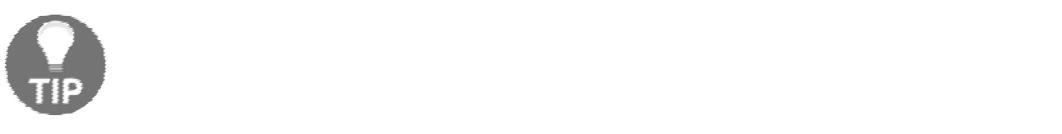
private $currency;

// ...

}

This approach has some noticeable flaws, if say, for example, you want to validate the ISO. It doesn't really make sense for the Product to be responsible for the Currency's ISO validation (thus violating the Single Responsibility Principle). This is highlighted even more so if you want to reuse the accompanying logic in other parts of your Domain (to abide by the DRY principle).

With these factors in mind, this use case is a perfect candidate for being abstracted out into a Value Object. Using this abstraction not only gives you the opportunity to group related properties together, but it also allows you to create higher-order concepts and a more concrete Ubiquitous Language.



**Exercise**

Discuss with a peer whether or not an email could be considered a Value Object. Does the context it's used in matter?

### 价值平等

As discussed at the beginning of the chapter, two Value Objects are equal if the content they measure, quantify, or describe is the same.

For example, imagine two Money objects representing 1 USD. Can we consider them equal? In the *real world*, are two bills of 1 USD valued the same? Of course they are. Directing our attention back to the code, the Value Objects in question refer to separate instances of Money. However, they both represent the same value, which makes them equal.

In regards to PHP, it's commonplace to compare two Value Objects using the == operator. Examining the [PHP Documentation](http://php.net/manual/en/language.oop5.object-comparison.php) definition of the operator highlights an interesting behavior:

When using the comparison operator ==, object variables are compared in a simple manner, namely: Two object instances are equal if they have the same attributes and values, and are instances of the same class.

This behavior works in agreement with our formal definition of a Value Object. However, as an exact class match predicate is present, you should be wary when handling subtyped Value Objects.

Keeping this in mind, the even stricter === operator doesn't help us, unfortunately:

When using the identity operator ===, object variables are identical if and only if they refer to the same instance of the same class.

The following example should help confirm these subtle differences:

$a = new Currency('USD');

$b = new Currency('USD');

var\_dump($a == $b); // bool(true) var\_dump($a === $b); // bool(false)

$c = new Currency('EUR');

var\_dump($a == $c); // bool(false) var\_dump($a === $c); // bool(false)

A solution is to implement a conventional equals method in each Value Object. This method is tasked with checking the type and equality of its composite attributes. Abstract data type comparability is easy to implement using the built-in type hinting in PHP. You can also use the get\_class() function to aid in the comparability check if necessary.

The language, however, is unable to decipher what equality truly means in your Domain concept, meaning it's your responsibility to provide the answer. In order to compare

the Currency objects, we just need to confirm that both their associated ISO codes are the same. The === operator does the job pretty well in this case:

class Currency

{

// ...

public function equals(Currency $currency)

{

return $currency->isoCode() === $this->isoCode();

}

}

Because Money objects use Currency objects, the equals method needs to perform this comparability check, along with comparing the amounts:

class Money

{

// ...

public function equals(Money $money)

{

return

$money->currency()->equals($this->currency()) &&

$money->amount() === $this->amount();

}

}

### 替换性

Consider a Product Entity that contains a Money Value Object used to quantify its price. Additionally, consider two Product Entities with an identical price — for example 100 USD. This scenario could be modeled using the two individual Money objects or two references pointing to a single Value Object.

Sharing the same Value Object can be risky; if one is altered, both will reflect the change. This behavior can be considered an unexpected side effect. For example, if Carlos was hired on February 20, and we know that Christian was also hired on the same day, we may set Christian's hire date to be the same instance as Carlos's. If Carlos then changes the month of his hire date to May, Christian's hire date changes too. Whether it's correct or not, it's not what people expect.

Due to the problems highlighted in this example, when holding a reference to a Value Object, it's recommended to replace the object as a whole rather than modifying its value:

$this−>price = new Money(100, new Currency('USD'));

//...

$this->price = $this->price->increaseAmountBy(200);

This kind of behavior is similar to how basic types such as strings work in PHP. Consider the function strtolower. It returns a new string rather than modifying the original one. No reference is used; instead, a new value is returned.

### 无副作用的行为

If we want to include some additional behavior — like an add method — in our Money class, it feels natural to check that the input fits any preconditions and maintains any invariance. In our case, we only wish to add monies with the same currency:

class Money

{

// ...

public function add(Money $money)

{

if ($money->currency() !== $this->currency()) { throw new InvalidArgumentException();

}

$this->amount += $money->amount();

}

}

If the two currencies don't match, an exception is raised. Otherwise, the amounts are added. However, this code has some undesirable pitfalls. Now imagine we have a mysterious method call to otherMethod in our code:

class Banking

{

public function doSomething()

{

$aMoney = new Money(100, new Currency('USD'));

$this->otherMethod($aMoney);//mysterious call

// ...

}

}

Everything is fine until, for some reason, we start seeing unexpected results when we're returning or finished with otherMethod. Suddenly, $aMoney no longer contains 100 USD. What happened? And what happens if otherMethod internally uses our previously defined add method? Maybe you're unaware that add mutates the state of the Money instance. This is what we call a side effect. You must avoid generating side effects. You must not mutate your arguments. If you do, the developer using your objects may experience strange behaviors. They'll complain, and they'll be correct.

So how can we fix this? Simple — by making sure that the Value Object remains immutable, we avoid this kind of unexpected problem. An easy solution could be returning a new instance for every potentially mutable operation, which the add method does:

class Money

{

// ...

public function add(Money $money)

{

if (!$money->currency()->equals($this->currency())) { throw new \InvalidArgumentException();

}

return new self(

$money->amount() + $this->amount(),

$this->currency()

);

}

}

With this simple change, immutability is guaranteed. Each time two instances of Money are added together, a new resulting instance is returned. Other classes can perform any number of changes without affecting the original copy. Code free of side effects is easy to understand, easy to test, and less error prone.

## 基本类型

Consider the following code snippet:

$a = 10;

$b = 10; var\_dump($a == $b);

// bool(true) var\_dump($a === $b);

// bool(true)

$a = 20; var\_dump($a);

// integer(20)

$a = $a + 30; var\_dump($a);

// integer(50);

Although $a and $b are different variables stored in different memory locations, when compared, they're the same. They hold the same value, so we consider them equal. You can change the value of $a from 10 to 20 at any time that you want, making the new value 20 and eliminating the 10. You can replace integer values as much as you want without consideration of the previous value because you're not modifying it; you're just replacing it. If you apply any operation — such as addition (That is. $a + $b) — to these variables, you get another new value that can be assigned to another variable or a previously defined one. When you pass $a to another function, except when explicitly passed by reference, you're passing a value. It doesn't matter if $a gets modified within that function, because in your current code, you'll still have the original copy. Value Objects behave as basic types.

## 测试值对象

Value Objects are tested in the same way normal objects are. However, the immutability and side-effect-free behavior must be tested too. A solution is to create a copy of the Value Object you're testing before performing any modifications. Assert both are equal using the implemented equality check. Perform the actions you want to test and assert the results.

Finally, assert that the original object and copy are still equal.

Let's put this into practice and test the side-effect-free implementation of our add method in the Money class:

class MoneyTest extends *Framework*TestCase

{

/\*\*

\* @test

\*/

public function copiedMoneyShouldRepresentSameValue()

{

$aMoney = new Money(100, new Currency('USD'));

$copiedMoney = Money::fromMoney($aMoney);

$this->assertTrue($aMoney->equals($copiedMoney));

}

/\*\*

\* @test

\*/

public function originalMoneyShouldNotBeModifiedOnAddition()

{

$aMoney = new Money(100, new Currency('USD'));

$aMoney->add(new Money(20, new Currency('USD')));

$this->assertEquals(100, $aMoney->amount());

}

/\*\*

\* @test

\*/

public function moniesShouldBeAdded()

{

$aMoney = new Money(100, new Currency('USD'));

$newMoney = $aMoney->add(new Money(20, new Currency('USD')));

$this->assertEquals(120, $newMoney->amount());

}

// ...

}

## Persisting Value Objects

Value Objects are not persisted on their own; they're typically persisted within an Aggregate. Value Objects shouldn't be persisted as complete records, though that's an option in some cases. Instead, it's best to use Embedded Value or Serialize LOB patterns. Both patterns can be used when persisting your objects with an open source ORM such as Doctrine, or with a bespoke ORM. As Value Objects are small, Embedded Value is usually the best choice because it provides an easy way to query Entities by any of the attributes the Value Object has. However, if querying by those fields isn't important to you, serialize strategies can be very easy to implement.

Consider the following Product Entity with string id, name, and price (Money Value Objects) attributes. We've intentionally decided to simplify this example, with the id being a string and not a Value Object:

class Product

{

private $productId; private $name; private $price;

public function construct(

$aProductId,

$aName,

Money $aPrice

) {

$this->setProductId($aProductId);

$this->setName($aName);

$this->setPrice($aPrice);

}

// ...

}

Assuming you have a [Chapter 10](#_bookmark315), *Repositories* for persisting Product Entities, an implementation to create and persist a new Product could look like this:

$product = new Product(

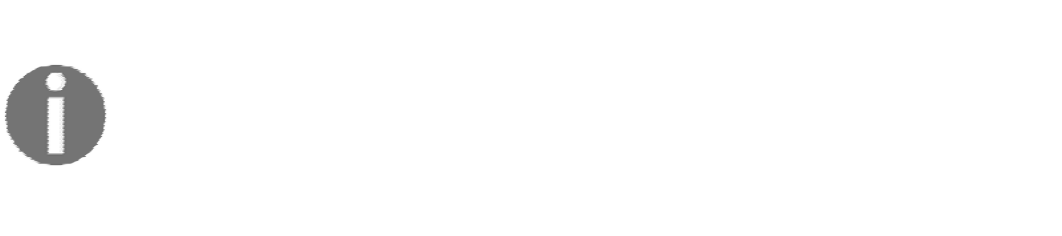
$productRepository->nextIdentity(), 'Domain-Driven Design in PHP',

new Money(999, new Currency('USD'))

);

$productRepository−>persist(product);

Now let's look at both the ad hoc ORM and the Doctrine implementations that could be used to persist a Product Entity containing Value Objects. We'll highlight the application of the Embedded Value and Serialized LOB patterns, along with the differences between persisting a single Value Object and a collection of them.



**Why Doctrine?**

The [Doctrine](http://www.doctrine-project.org/projects/orm.html) is a great ORM. It solves 80 percent of the requirements a PHP application faces. It has a great community. With a correctly tuned setup, it can perform the same or even better than a bespoke ORM (without losing maintainability). We recommend using Doctrine in most cases when dealing with Entities and business logic. It will save you a lot of time and headaches.

### Persisting Single Value Objects

Many different options are available for persisting a single Value Object. These range from using Serialize LOB or Embedded Value as mapping strategies, to using an Ad Hoc ORM or an open source alternative, such as Doctrine. We consider an Ad Hoc ORM to be a custom- built ORM that your company may have developed in order to persist Entities in a database. In our scenario, the Ad Hoc ORM code is going to be implemented using the

[DBAL](http://docs.doctrine-project.org/projects/doctrine-dbal/en/latest/) library. According to the [official documentation](http://docs.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/introduction.html), The **Doctrine Database Abstraction** & **Access Layer** (**DBAL**) offers a lightweight and thin runtime layer around a PDO-like API and a lot of additional, horizontal features like database schema introspection and manipulation through an OO API.

#### Embedded Value with an Ad Hoc ORM

If we're dealing with an Ad Hoc ORM using the Embedded Value pattern, we need to create a field in the Entity table for each attribute in the Value Object. In this case, two extra columns are needed when persisting a Product Entity — one for the amount of the Value Object, and one for its currency ISO code:

CREATE TABLE `products` ( id INT NOT NULL,

name VARCHAR( 255) NOT NULL,

price\_amount INT NOT NULL, price\_currency VARCHAR( 3) NOT NULL

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4\_unicode\_ci;

For persisting the Entity in the database, our [Chapter 10](#_bookmark315), *Repositories* has to map each of the fields of the Entity and the ones from the Money Value Object.

If you're using an Ad hoc ORM Repository based on DBAL—let's call it DbalProductRepository—you must take care of creating the INSERT statement, binding the parameters, and executing the statement:

class DbalProductRepository extends DbalRepository implements ProductRepository

{

public function add(Product $aProduct)

{

$sql = 'INSERT INTO products VALUES (, , , )' ;

$stmt = $this->connection()->prepare($sql);

$stmt->bindValue(1, $aProduct->id());

$stmt->bindValue(2, $aProduct->name());

$stmt->bindValue(3, $aProduct->price()->amount());

$stmt->bindValue(4, $aProduct

->price()->currency()->isoCode());

$stmt->execute();

// ...

}

}

After executing this snippet of code to create a Products Entity and persist it into the database, each column is filled with the desired information:

mysql> select \* from products \G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* id: 1

name: Domain-Driven Design in PHP price\_amount: 999

price\_currency: USD

1 row in set (0.00 sec)

As you can see, you can map your Value Objects and query parameters in an Ad hoc manner in order to persist your Value Objects. However, everything is not as easy as it seems. Let's try to fetch the persisted Product with its associated Money Value Object. A common approach would be to execute a SELECT statement and return a new Entity:

class DbalProductRepository extends DbalRepository implements ProductRepository

{

public function productOfId($anId)

{

$sql = 'SELECT \* FROM products WHERE id = ';

$stmt = $this->connection()->prepare($sql);

$stmt->bindValue(1, $anId);

$res = $stmt->execute();

// ...

return new Product(

$row['id'],

$row['name'], new Money(

$row['price\_amount'],

new Currency($row['price\_currency'])

)

);

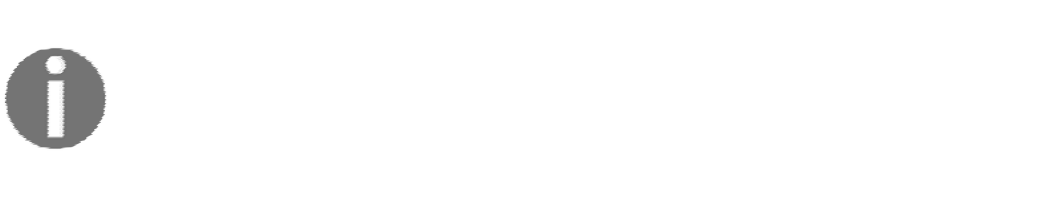
}

}

There are some benefits to this approach. First, you can easily read, step by step, how the persistence and subsequent creations occur. Second, you can perform queries based on any of the attributes of the Value Object. Finally, the space required to persist the Entity is just what is required — no more and no less.

However, using the ad hoc ORM approach has its drawbacks. As explained in the [Chapter](#_bookmark178) [6](#_bookmark178), *Domain-Events*, Entities (in Aggregate form) should fire an Event in the constructor if your Domain is interested in the Aggregate's creation. If you use the new operator, you'll be firing the Event as many times as the Aggregate is fetched from the database.

This is one of the reasons why Doctrine uses internal proxies and serialize and unserialize methods to reconstitute an object with its attributes in a specific state without using its constructor. An Entity should only be created with the new operator once in its lifetime:



**Constructors**

Constructors don't need to include a parameter for each attribute in the object. Think about a blog post. A constructor may need an id and a title; however, internally it can also be setting its status attribute to draft. When publishing the post, a publish method should be called in order to alter its status accordingly and set a published date.

If your intention is still to roll out your own ORM, be ready to solve some fundamental problems such as Events, different constructors, Value Objects, lazy load relations, and so on. That's why we recommend giving Doctrine a try for Domain-Driven Design applications.

Besides, in this instance, you need to create a DbalProduct Entity that extends from the Product Entity and is able to reconstitute the Entity from the database without using the new operator, instead using a static factory method.

#### 内含价值（Embeddables）与学说> = 2.5。\*

The latest stable Doctrine release is currently *version 2.5* and it comes with support for mapping Value Objects, thereby removing the need to do this yourself as in *Doctrine 2.4*. Since December 2015, Doctrine also has support for nested embeddables. The support is not 100 percent, but it's high enough to give it a try. In case it doesn't work for your scenario, take a look at the next section. For official documentation, check the Doctrine [Embeddables](http://doctrine-orm.readthedocs.org/en/latest/tutorials/embeddables.html) [reference](http://doctrine-orm.readthedocs.org/en/latest/tutorials/embeddables.html). This option, if implemented correctly, is definitely the one we recommend most. It would be the simplest, most elegant solution, that also provides search support in your *DQL* queries.

Because the Product, Money, and Currency classes have already been shown, the only thing remaining is to show the Doctrine mapping files:

<xml version="1.0" encoding="utf-8">

<doctrine-mapping

[xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping"](http://doctrine-project.org/schemas/orm/doctrine-mapping) [xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"](http://www.w3.org/2001/XMLSchema-instance) xsi:schemaLocation="

<http://doctrine-project.org/schemas/orm/doctrine-mapping> https://raw.github.com/doctrine/doctrine2/master/doctrine-mapping.xsd">

<entity

name="Product" table="product">

<id

name="id" column="id" type="string" length="255">

<generator strategy="NONE">

</generator>

</id>

<field

name="name" type="string" length="255"

/>

<embedded

name="price" class="Ddd\Domain\Model\Money"

/>

</entity>

</doctrine-mapping>

In the product mapping, we're defining price as an instance variable that will hold a Money instance. At the same time, Money is designed with an amount and a Currency instance:

<xml version="1.0" encoding="utf-8">

<doctrine-mapping

[xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping"](http://doctrine-project.org/schemas/orm/doctrine-mapping) [xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"](http://www.w3.org/2001/XMLSchema-instance) xsi:schemaLocation="

<http://doctrine-project.org/schemas/orm/doctrine-mapping> https://raw.github.com/doctrine/doctrine2/master/doctrine-mapping.xsd">

<embeddable

name="Ddd\Domain\Model\Money">

<field

name="amount" type="integer"

/>

<embedded

name="currency" class="Ddd\Domain\Model\Currency"

/>

</embeddable>

</doctrine-mapping>

Finally, it's time to show the Doctrine mapping for our Currency Value Object:

<xml version="1.0" encoding="utf-8">

<doctrine-mapping

[xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping"](http://doctrine-project.org/schemas/orm/doctrine-mapping) [xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"](http://www.w3.org/2001/XMLSchema-instance) xsi:schemaLocation="

<http://doctrine-project.org/schemas/orm/doctrine-mapping> https://raw.github.com/doctrine/doctrine2/master/doctrine-mapping.xsd">

<embeddable

name="Ddd\Domain\Model\Currency">

<field

name="iso" type="string"

length="3"

/>

</embeddable>

</doctrine-mapping>

As you can see, the above code has a standard embeddable definition with just one string field that holds the ISO code. This approach is the easiest way to use embeddables and is much more effective. By default, Doctrine names your columns by prefixing them using the Value Object name. You can change this behavior to meet your needs by changing the column-prefix attribute in the XML notation.

#### Embedded Value with Doctrine <= 2.4.\*

If you're still stuck in *Doctrine 2.4*, you may wonder what an acceptable solution for using Embedded Values with *Doctrine < 2.5* is. We need to now surrogate all the Value Object attributes in the Product Entity, which means creating new artificial attributes that will hold the information of the Value Object. With this in place, we can map all those new attributes using Doctrine. Let's see what impact this has on the Product Entity:

class Product

{

private $productId; private $name; private $price;

private $surrogateCurrencyIsoCode; private $surrogateAmount;

public function construct($aProductId, $aName, Money $aPrice)

{

$this->setProductId($aProductId);

$this->setName($aName);

$this->setPrice($aPrice);

}

private function setPrice(Money $aMoney)

{

$this->price = $aMoney;

$this->surrogateAmount = $aMoney->amount();

$this->surrogateCurrencyIsoCode =

$aMoney->currency()->isoCode();

}

private function price()

{

if (null === $this->price) {

$this->price = new Money(

$this->surrogateAmount,

new Currency($this->surrogateCurrency)

);

}

return $this->price;

}

// ...

}

As you can see, there are two new attributes: one for the amount, and another for the ISO code of the currency. We've also updated the setPrice method in order to keep attribute consistency when setting it. On top of this, we updated the price getter in order to return the Money Value Object built from the new fields. Let's see how the corresponding XML Doctrine mapping should be changed:

<xml version="1.0" encoding="utf-8">

<doctrine-mapping

[xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping"](http://doctrine-project.org/schemas/orm/doctrine-mapping) [xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"](http://www.w3.org/2001/XMLSchema-instance) xsi:schemaLocation="

<http://doctrine-project.org/schemas/orm/doctrine-mapping> https://raw.github.com/doctrine/doctrine2/master/doctrine-mapping.xsd">

<entity

name="Product" table="product">

<id

name="id" column="id" type="string" length="255" >

<generator strategy="NONE">

</generator>

</id>

<field

name="name" type="string" length="255"

/>

<field

name="surrogateAmount" type="integer"

column="price\_amount"

/>

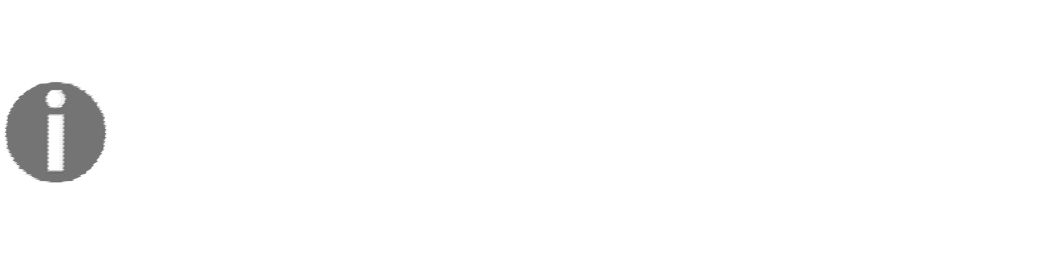
<field

name="surrogateCurrencyIsoCode" type="string" column="price\_currency"

/>

</entity>

</doctrine-mapping>



**Surrogate Attributes**

These two new fields don't strictly belong to the Domain, as they don't refer to Infrastructure details. Rather, they're a necessity due to the lack of embeddable support in Doctrine. There are alternatives that can push these two attributes outside the pure Domain; however, this approach is simpler, easier, and, as a tradeoff, acceptable. There's another use of surrogate attributes in this book; you can find it in the sub-section *Surrogate Identity* of the section *Identity Operation* of [Chapter 4](#_bookmark101)*, Entities*.

If we wanted to push these two attributes outside of the Domain, this could be achieved through the use of an [Abstract Factory](http://en.wikipedia.org/wiki/Abstract_factory_pattern). First, we need to create a new Entity, DoctrineProduct, in our Infrastructure folder. This Entity will extend from Product Entity. All surrogate fields will be placed in the new class, and methods such as price or setPrice should be reimplemented. We'll map Doctrine to use the new DoctrineProduct as opposed to the Product Entity.

Now we're able to fetch Entities from the database, but what about creating a new Product? At some point, we're required to call new Product, but because we need to deal with DoctrineProduct and we don't want our Application Services to know about Infrastructure details, we'll need to use Factories to create Product Entities. So, in every instance where Entity creation occurs with new, you'll instead call createProduct on ProductFactory.

There could be many additional classes required to avoid placing the surrogate attributes in the original Entity. As such, it's our recommendation to surrogate all the Value Objects to the same Entity, though this admittedly leads to a less pure solution.

### Serialized LOB and Ad Hoc ORM

If the addition of searching capabilities to the Value Objects attributes is not important, there's another pattern that can be considered: the Serialized LOB. This pattern works by serializing the whole Value Object into a string format that can easily be persisted and fetched. The most significant difference between this solution and the embedded alternative is that in the latter option, the persistence footprint requirements are reduced to a single column:

CREATE TABLE ` products` ( id INT NOT NULL,

name VARCHAR( 255) NOT NULL,

price TEXT NOT NULL

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4\_unicode\_ci;

In order to persist the Product Entities using this approach, a change in the DbalProductRepository is required. The Money Value Object must be serialized into a string before persisting the final Entity:

class DbalProductRepository extends DbalRepository implements ProductRepository

{

public function add(Product $aProduct)

{

$sql = 'INSERT INTO products VALUES (, , )';

$stmt = this->connection()->prepare(sql);

$stmt->bindValue(1, aProduct−>id());

$stmt->bindValue(2, aProduct−>name());

$stmt->bindValue(3, $this−>serialize($aProduct->price()));

// ...

}

private function serialize($object)

{

return serialize($object);

}

}

Let's see how our Product is now represented in the database. The table column price is a

TEXT type column that contains a serialization of a Money object representing 9.99 USD:

mysql > select \* from products \G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1.row\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* id : 1

name : Domain-Driven Design in PHP

price : O:22:"Ddd\Domain\Model\Money":2:{s:30:"Ddd\Domain\Model\\ Money amount";i :

999;s:32:"Ddd\Domain\Model\Money currency";O : 25:"Ddd\Domain\Model\\ Currency":1:{\

s:34:" Ddd\Domain\Model\Currency isoCode";s:3:"USD";}}1 row in set(\ 0.00 sec)

This approach does the job. However, it's not recommended due to problems occurring when refactoring classes in your code. Could you imagine the problems if we decided to rename our Money class? Could you imagine the changes that would be required in our database representation when moving the Money class from one namespace to another? Another tradeoff, as explained before, is the lack of querying capabilities. It doesn't matter whether you use Doctrine or not; writing a query to get the products cheaper than, say, 200 USD is almost impossible while using a serialization strategy.

The querying issue can only be solved by using Embedded Values. However, the serialization refactoring problems can be fixed using a specialized library for handling serialization processes.

#### 使用JMS序列化器改进序列化

The serialize/unserialize native PHP strategies have a problem when dealing with class and namespace refactoring. One alternative is to use your own serialization mechanism — for example, concatenating the amount, a one character separator such as |, and the currency ISO code. However, there's another favored approach: using an open source serializer library such as [JMS Serializer](http://jmsyst.com/libs/serializer). Let's see an example of applying it when serializing a Money object:

$myMoney = new Money(999, new Currency('USD'));

$serializer = JMS\Serializer\SerializerBuilder::create()->build();

$jsonData = $serializer−>serialize(myMoney, 'json');

In order to unserialize the object, the process is straightforward:

$serializer = JMS\Serializer\SerializerBuilder::create()->build();

// ...

$myMoney = $serializer−>deserialize(jsonData, 'Ddd', 'json');

With this example, you can refactor your Money class without having to update your database. JMS Serializer can be used in many different scenarios — for example, when working with REST APIs. An important feature is the ability to specify which attributes of an object should be omitted in the serialization process — for example, a password.

Check out the [Mapping Reference](http://jmsyst.com/libs/serializer/master/reference/xml_reference) and the [Cookbook](http://jmsyst.com/libs/serializer/master/cookbook) for more information. JMS Serializer is a must in any Domain-Driven Design project.

### 序列化LOB with Doctrine

In Doctrine, there are different ways of serializing objects in order to eventually persist them.

#### Doctrine Object Mapping Type

Doctrine has support for the Serialize LOB pattern. There are plenty of predefined mapping types you can use in order to match Entity attributes with database columns or even tables. One of those mappings is the object type, which maps an SQL CLOB to a PHP object using serialize() and unserialize().

According to the [Doctrine DBAL 2 Documentation](http://doctrine-orm.readthedocs.io/projects/doctrine-dbal/en/latest/reference/types.html#object), object type:

Maps and converts object data based on PHP serialization. If you need to store an exact representation of your object data, you should consider using this type as it uses serialization to represent an exact copy of your object as string in the database. Values retrieved from the database are always converted to PHP's object type using unserialization or null if no data is present.

This type will always be mapped to the database vendor's text type internally as there is no way of storing a PHP object representation natively in the database. Furthermore this type requires a SQL column comment hint so that it can be reverse engineered from the database. Doctrine cannot correctly map back this type correctly using vendors that do not support column comments, and will instead fall back to the text type instead.

Because the built-in text type of PostgreSQL does not support NULL bytes, the object type will result in unserialization errors. A workaround to this problem is to serialize()/unserialize() and base64\_encode()/base64\_decode() PHP objects and store them into a text field manually.

Let's look at a possible XML mapping for the Product Entity by using the object type:

<xml version="1.0" encoding="utf-8">

<doctrine-mapping

[xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping"](http://doctrine-project.org/schemas/orm/doctrine-mapping) [xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"](http://www.w3.org/2001/XMLSchema-instance) xsi:schemaLocation="

<http://doctrine-project.org/schemas/orm/doctrine-mapping> https://raw.github.com/doctrine/doctrine2/master/doctrine-mapping.xsd">

<entity

name="Product" table="products">

<id

name="id" column="id" type="string" length="255">

<generator strategy="NONE">

</generator>

</id>

<field

name="name" type="string" length="255"

/>

<field

name="price" type="object"

/>

</entity>

</doctrine-mapping>

The key addition is type="object", which tells Doctrine that we're going to be using an object mapping. Let's see how we could create and persist a Product Entity using Doctrine:

// ...

$em−>persist($product);

$em−>flush($product);

Let's check that if we now fetch our Product Entity from the database, it's returned in an expected state:

// ...

$repository = $em->getRepository('Ddd\\Domain\\Model\\Product');

$item = $repository->find(1); var\_dump($item);

/\*

class Ddd\Domain\Model\Product#177 (3) { private $productId => int(1)

private $name => string(41) "Domain-Driven Design in PHP" private $money => class Ddd\Domain\Model\Money#174 (2) {

private $amount => string(3) "100"

private $currency => class Ddd\Domain\Model\Currency#175 (1){ private $isoCode => string(3) "USD"

}

}

}

\* /

Last but not least, the [Doctrine DBAL 2 Documentation](http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/basic-mapping.html#doctrine-mapping-types) states that:

Object types are compared by reference, not by value. Doctrine updates this value if the reference changes and therefore behaves as if these objects are immutable value objects.

This approach suffers from the same refactoring issues as the Ad hoc ORM did. The object mapping type is internally using serialize/unserialize. What about instead using our own serialization?

#### Doctrine Custom Types

Another option is to handle the Value Object persistence using a Doctrine Custom Type. A Custom Type adds a new mapping type to Doctrine — one that describes a custom transformation between an Entity field and the database representation, in order to persist the former.

As the [Doctrine DBAL 2 Documentation](http://doctrine-orm.readthedocs.io/projects/doctrine-dbal/en/latest/reference/types.html#custom-mapping-types) explains:

Just redefining how database types are mapped to all the existing Doctrine types is not at all that useful. You can define your own Doctrine Mapping Types by extending Doctrine\DBAL\Types\Type. You are required to implement 4 different methods to get this working.

With the object type, the serialization step includes information, such as the class, that makes it quite difficult to safely refactor our code.

Let's try to improve on this solution. Think about a custom serialization process that could solve the problem.

One such way could be to persist the Money Value Object as a string in the database encoded in amount|isoCode format:

use Ddd\Domain\Model\Currency; use Ddd\Domain\Model\Money;

use Doctrine\DBAL\Types\TextType;

use Doctrine\DBAL\Platforms\AbstractPlatform;

class MoneyType extends TextType

{

const MONEY = 'money';

public function convertToPHPValue(

$value,

AbstractPlatform $platform

) {

$value = parent::convertToPHPValue($value, $platform);

$value = explode('|', $value); return new Money(

$value[0],

new Currency($value[1])

);

}

public function convertToDatabaseValue(

$value,

AbstractPlatform $platform

) {

return implode( '|',

[

$value->amount(),

$value->currency()->isoCode()

]

);

}

public function getName()

{

return self::MONEY;

}

}

Using Doctrine, you're required to register all Custom Types. It's common to use an

EntityManagerFactory that centralizes this EntityManager creation.

Alternatively, you could perform this step by bootstrapping your application:

use Doctrine\DBAL\Types\Type; use Doctrine\ORM\EntityManager; use Doctrine\ORM\Tools\Setup;

class EntityManagerFactory

{

public function build()

{

Type::addType( 'money',

'Ddd\Infrastructure\Persistence\Doctrine\Type\MoneyType'

);

return EntityManager::create( [

'driver' => 'pdo\_mysql', 'user' => 'root',

'password' => '',

'dbname' => 'ddd',

],

Setup::createXMLMetadataConfiguration( [ DIR .'/config'],

true

)

);

}

}

Now we need to specify in the mapping that we want to use our Custom Type:

<xml version = "1.0" encoding = "utf-8">

<doctrine-mapping>

<entity

name = "Product" table = "product">

<!-- ... -->

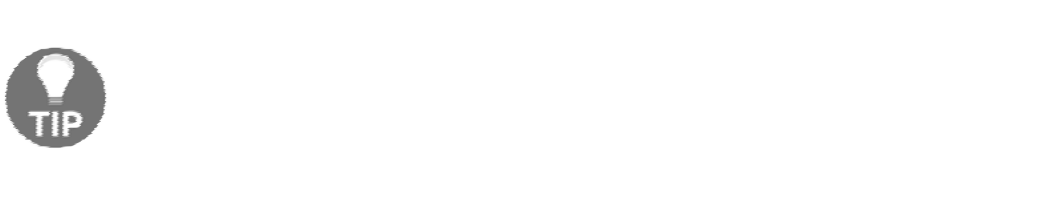
<field

name = "price" type = "money"

/>

</entity>

</doctrine-mapping>



**Why Use XML Mapping?**

Thanks to the XSD schema validation in the headers of the XML mapping file, many **Integrated Development Environment** (**IDEs**) setups provide auto-complete functionality for all the elements and attributes present in the mapping definition. However, in other parts of the book, we use YAML to show a different syntax.

Let's check the database to see how the price was persisted using this approach:

mysql> select \* from products \G

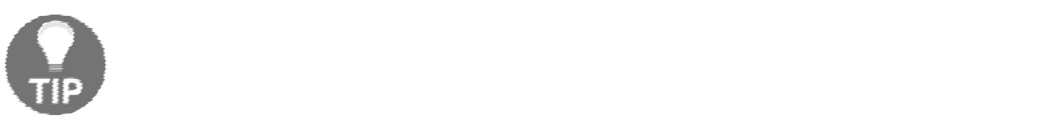
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

name: Domain-Driven Design in PHP price: 999|USD

1 row in set (0.00 sec)

This approach is an improvement on the one before in terms of future refactoring. However, searching capabilities remain limited due to the format of the column. With the Doctrine Custom types, you can improve the situation a little, but it's still not the best option for building your DQL queries. See [Doctrine Custom Mapping Types](http://doctrine-orm.readthedocs.org/en/latest/cookbook/custom-mapping-types.html) for more information.



**Time to Discuss**

Think about and discuss with a peer how would you create a Doctrine Custom Type using JMS to serialize and unserialize a Value Object.

#### Persisting a Collection of Value Objects

Imagine that we'd now like to add a collection of prices to be persisted to our Product Entity. These prices could represent the different prices the product has borne throughout its lifetime or the product price in different currencies. This could be named HistoricalPrice, as shown below:

class HistoricalProduct extends Product

{

/\*\*

\* @var Money[]

\*/

protected $prices;

public function construct(

$aProductId,

$aName,

Money $aPrice, array $somePrices

){

parent:: construct($aProductId, $aName, $aPrice);

$this->setPrices($somePrices);

}

private function setPrices(array $somePrices)

{

$this->prices = $somePrices;

}

public function prices()

{

return $this->prices;

}

}

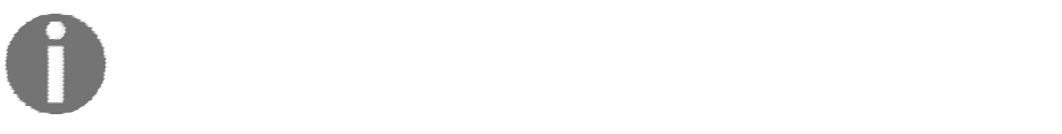
HistoricalProduct extends from Product, so it inherits the same behavior, plus the price collection functionality.

As in the previous sections, serialization is a plausible approach if you don't care about querying capabilities. However, Embedded Values are a possibility if we know exactly how many prices we want to persist. But what happens if we want to persist an undetermined collection of historical prices?

#### Collection Serialized into a Single Column

Serializing a collection of Value Objects into a single column is most likely the easiest solution. Everything that was previously explained in the section about persisting a single Value Object applies in this situation. With Doctrine, you can use an Object or Custom Type

— with some additional considerations to bear in mind: Value Objects should be small in size, but if you wish to persist a large collection, be sure to consider the maximum column length and the maximum row width that your database engine can handle.



**Exercise**

Come up with both Doctrine Object Type and Doctrine Custom Type implementation strategies for persisting a Product with different prices.

#### Collection Backed by a Join Table

In case you want to persist and query an Entity by its related Value Objects, you have the choice to persist the Value Objects as Entities. In terms of the Domain, those objects would still be Value Objects, but we'll need to give them an id and set them up with a one-to- many/one-to-one relation with the owner, a real Entity. To summarize, your ORM handles the collection of Value Objects as Entities, but in your Domain, they're still treated as Value Objects.

The main idea behind the Join Table strategy is to create a table that connects the owner Entity and its Value Objects. Let's see a database representation:

CREATE TABLE ` historical\_products` (

`id` char( 36) COLLATE utf8mb4\_unicode\_ci NOT NULL,

`name` varchar( 255) COLLATE utf8mb4\_unicode\_ci NOT NULL,

`price\_amount` int( 11 ) NOT NULL,

`price\_currency` char( 3) COLLATE utf8mb4\_unicode\_ci NOT NULL, PRIMARY KEY (`id`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4\_unicode\_ci;

The historical\_products table will look the same as products. Remember that HistoricalProduct extends Product Entity in order to easily show how to deal with persisting a collection. A new prices table is now required in order to persist all the different Money Value Objects that a Product Entity can handle:

CREATE TABLE `prices`(

`id` int(11) NOT NULL AUTO\_INCREMENT,

`amount` int(11) NOT NULL,

`currency` char(3) COLLATE utf8mb4\_unicode\_ci NOT NULL, PRIMARY KEY (`id`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4\_unicode\_ci;

Finally, a table that relates products and prices is needed:

CREATE TABLE `products\_prices` (

`product\_id` char( 36) COLLATE utf8mb4\_unicode\_ci NOT NULL,

`price\_id` int( 11 ) NOT NULL,

PRIMARY KEY (`product\_id`, `price\_id`),

UNIQUE KEY `UNIQ\_62F8E673D614C7E7` (`price\_id`),

KEY `IDX\_62F8E6734584665A` (`product\_id`),

CONSTRAINT `FK\_62F8E6734584665A` FOREIGN KEY (`product\_id`)

REFERENCES `historical\_products` (`id`),

CONSTRAINT `FK\_62F8E673D614C7E7` FOREIGN KEY (`price\_id`)

REFERENCES `prices`(`id`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4\_unicode\_ci;

##### Collection Backed by a Join Table with Doctrine

Doctrine requires that all database Entities have a unique identity. Because we want to persist Money Value Objects, we need to then add an artificial identity so Doctrine can handle its persistence. There are two options: including the surrogate identity in the Money Value Object, or placing it in an extended class.

The issue with the first option is that the new identity is only required due to the Database persistence layer. This identity is not part of the Domain.

An issue with the second option is the amount of alterations required in order to avoid this so-called boundary leak. With a class extension, creating new instances of the Money Value Object class from any Domain Object isn't recommended, as it would break the Inversion Principle. The solution is to again create a Money Factory that would need to be passed into Application Services and any other Domain Objects.

In this scenario, we recommend using the first option. Let's review the changes required to implement it in the Money Value Object:

class Money

{

private $amount; private $currency; private $surrogateId;

private $surrogateCurrencyIsoCode;

public function construct($amount, Currency $currency)

{

$this->setAmount($amount);

$this->setCurrency($currency);

}

private function setAmount($amount)

{

$this->amount = $amount;

}

private function setCurrency(Currency $currency)

{

$this->currency = $currency;

$this->surrogateCurrencyIsoCode =

$currency->isoCode();

}

public function currency()

{

if (null === $this->currency) {

$this->currency = new Currency(

$this->surrogateCurrencyIsoCode

);

}

return $this->currency;

}

public function amount()

{

return $this->amount;

}

public function equals(Money $aMoney)

{

return

$this->amount() === $aMoney->amount() &&

$this->currency()->equals($this->currency());

}

}

As seen above, two new attributes have been added. The first one, surrogateId, is not used by our Domain, but it's required for the persistence Infrastructure to persist this Value Object as an Entity in our Database. The second one, surrogateCurrencyIsoCode, holds the ISO code for the currency. Using these new attributes, it's really easy to map our Value Object with Doctrine.

The Money mapping is quite straightforward:

<xml version = "1.0" encoding = "utf-8">

<doctrine-mapping

[xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping"](http://doctrine-project.org/schemas/orm/doctrine-mapping) [xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"](http://www.w3.org/2001/XMLSchema-instance) xsi:schemaLocation="

<http://doctrine-project.org/schemas/orm/doctrine-mapping> https://raw.github.com/doctrine/doctrine2/master/doctrine-mapping.xsd">

<entity

name="Ddd\Domain\Model\Money" table="prices">

<id

name="surrogateId" type="integer"

column="id">

<generator

strategy="AUTO">

</generator>

</id>

<field

name="amount" type="integer" column="amount"

/>

<field

name="surrogateCurrencyIsoCode" type="string"

column="currency"

/>

</entity>

</doctrine-mapping>

Using Doctrine, the HistoricalProduct Entity would have following mapping:

<xml version="1.0" encoding="utf-8">

<doctrine-mapping

[xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping"](http://doctrine-project.org/schemas/orm/doctrine-mapping) [xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"](http://www.w3.org/2001/XMLSchema-instance) xsi:schemaLocation="

<http://doctrine-project.org/schemas/orm/doctrine-mapping> https://raw.github.com/doctrine/doctrine2/master/doctrine-mapping.xsd">

<entity

name="Ddd\Domain\Model\HistoricalProduct" table="historical\_products"

repository-class="

Ddd\Infrastructure\Domain\Model\DoctrineHistoricalProductRepository ">

<many-to-many

field="prices"

target-entity="Ddd\Domain\Model\Money">

<cascade>

<cascade-all/>

</cascade>

<join-table

name="products\_prices">

<join-columns>

<join-column

name="product\_id"

referenced-column-name="id"

/>

</join-columns>

<inverse-join-columns>

<join-column

name="price\_id"

referenced-column-name="id" unique="true"

/>

</inverse-join-columns>

</join-table>

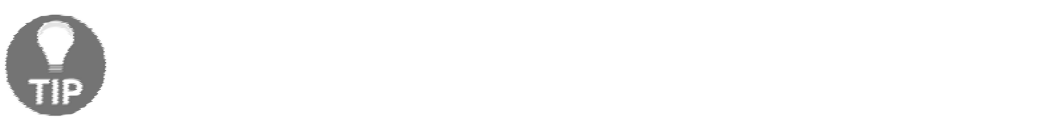
</many-to-many>

</entity>

</doctrine-mapping>

##### Collection Backed by a Join Table with an Ad Hoc ORM

It's possible to do the same with an Ad hoc ORM, where Cascade INSERTS and JOIN queries are required. It's important to be careful about how the removal of Value Objects is handled, in order to not leave orphan the Money Value Objects.

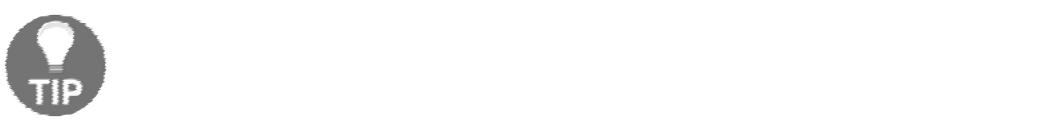


**Exercise**

Think up a solution for DbalHistoricalRepository that would handle the persist method.

#### Collection Backed by a Database Entity

Database Entity is the same solution as Join Table, with the addition of the Value Object that's only managed by the owner Entity. In the current scenario, consider that the Money Value Object is only used by the HistoricalProduct Entity; a Join Table would be overly complex. So the same result could be achieved using a one-to-many database relation.



**Exercise**

Think of the mapping required between HistoricalProduct and Money

if a Database Entity approach is used.

### NoSQL

What about NoSQL mechanisms such as *Redis*, *MongoDB*, or *CouchDB*? Unfortunately, you can't escape these problems. In order to persist an Aggregate using *Redis*, you need to serialize it into a string before setting the value. If you use PHP serialize/unserialize methods, you'll face namespace or class name refactoring issues again. If you choose to serialize in a custom way (JSON, custom string, and so on.), you'll be required to again rebuild the Value Object during *Redis* retrieval.

#### PostgreSQL JSONB and MySQL JSON Type

If our database engine would allow us to not only use the Serialized LOB strategy but also search based on its value, we would have the best of both approaches. Well, good news: now you *can* do this. As of *PostgreSQL version 9.4*, support for [JSONB](http://www.postgresql.org/docs/9.4/static/functions-json.html) has been added. Value Objects can be persisted as JSON serializations and subsequently queried within this JSON serialization.

MySQL has done the same. As of *MySQL 5.7.8*, MySQL supports a native JSON data type that enables efficient access to data in **JSON** (**JavaScript Object Notation**) documents.

According to the [MySQL 5.7 Reference Manual](https://dev.mysql.com/doc/refman/5.7/en/json.html), the JSON data type provides these advantages over storing JSON-format strings in a string column:

Automatic validation of JSON documents stored in JSON columns. Invalid documents produce an error.

Optimized storage format. JSON documents stored in JSON columns are converted to an internal format that permits quick read access to document elements. When the server later must read a JSON value stored in this binary format, the value need not be parsed from a text representation. The binary format is structured to enable the server to look up subobjects or nested values directly by key or array index without reading all values before or after them in the document.

If Relational Databases add support for document and nested document searches with high performance and with all the benefits of an **Atomicity**, **Consistency**, **Isolation**, **Durability**(**ACID**) philosophy, it could save a lot of complexity in many projects.

## Security

Another interesting detail of modeling your Domain concepts using Value Objects is regarding its security benefits. Consider an application within the context of selling flight tickets. If you deal with International Air Transport Association airport codes, also known as [IATA codes](https://en.wikipedia.org/wiki/International_Air_Transport_Association_airport_code), you can decide to use a string or model the concept using a Value Object. If you choose to go with the string, think about all the places where you'll be checking that the string is a valid IATA code. What's the chance you forget somewhere important? On the other hand, think about trying to instantiate new IATA("BCN'; DROP TABLE users;-- "). If you centralize the *guards* in the constructor and then pass an IATA Value Object into your model, avoiding SQL Injections or similar attacks gets easier.

If you want to know more about the security side of Domain-Driven Design, you can follow

[Dan Bergh Johnsson](https://twitter.com/danbjson) or read his [blog](http://dearjunior.blogspot.com.es/search/label/domain%20driven%20security).

## Wrap-Up

Using Value Objects for modeling concepts in your Domain that measure, quantify, or describe is highly recommended. As shown, Value Objects are easy to create, maintain, and test. In order to handle persistence within a Domain-Driven Design application, using an ORM is a must. However, in order to persist Value Objects using Doctrine, the preferred way is using embeddables. In case you're stuck in *version 2.4*, there are two options: adding the Value Object fields directly into your Entity and mapping them (less elegant, but easier), or extending your Entities (far more elegant, but more complex).

# Entities

****

我们已经谈到尝试首先将域中的所有内容建模为价值对象的好处。 但是，在对域进行建模时，可能会出现以下情况：您会发现无处不在的语言中的某些概念需要一个标识线程.

## Introduction

需要身份的对象的清晰示例包括:

一个人。 一个人总是有一个身份，并且在他们的姓名或身份证方面总是一样的.

电子商务系统中的订单。 在这样的背景下，每一个创建的新订单都有自己的标识，并且随着时间的推移也是一样的.

这些概念有一个持久的身份。 无论概念中的数据发生了多少变化，其身份保持不变。 这就是使它们成为实体而不是价值对象的原因。 就PHP的实现而言，它们将是普通的旧类。 例如，考虑一个人的情况如下:

namespace Ddd\Identity\Domain\Model;

class Person

{

private $identificationNumber; private $firstName;

private $lastName;

public function construct(

$anIdentificationNumber, $aFirstName, $aLastName

) {

$this->identificationNumber = $anIdentificationNumber;

$this->firstName = $aFirstName;

$this->lastName = $aLastName;

}

public function identificationNumber()

{

return $this->identificationNumber;

}

public function firstName()

{

return $this->firstName;

}

public function lastName()

{

return $this->lastName;

}

}

或者，在订单的情况下考虑以下内容:

namespace Ddd\Billing\Domain\Model\Order;

class Order

{

private $id; private $amount; private $firstName; private $lastName;

public function construct(

$anId, Amount $amount, $aFirstName, $aLastName

) {

$this->id = $anId;

$this->amount = $amount;

$this->firstName = $aFirstName;

$this->lastName = $aLastName;

}

public function id()

{

return $this->id;

}

public function firstName()

{

return $this->firstName;

}

public function lastName()

{

return $this->lastName;

}

}

## 对象VS. 原始类型

Most of the time, the Identity of an Entity is represented as a primitive type — usually a string or an integer. But using a Value Object to represent it has more advantages:

Value Objects are immutable, so they can't be modified.

Value Objects are complex types that can have custom behaviors, something which primitive types can't have. Take, as an example, **the equality operation**. With Value Objects, equality operations can be modeled and encapsulated in their own classes, making concepts go from implicit to explicit.

Let's see a possible implementation for OrderId, the Order Identity that has evolved into a Value Object:

namespace Ddd\Billing\Domain\Model;

class OrderId

{

private $id;

public function construct($anId)

{

$this->id = $anId;

}

public function id()

{

return $this->id;

}

public function equalsTo(OrderId $anOrderId)

{

return $anOrderId->id === $this->id;

}

}

您可以考虑实施OrderId的不同实现。 上面显示的例子非常简单。 如第3章“值对象”中所述，您可以使构造方法为私有方法，并使用静态工厂方法来创建新实例。 与你的团队交谈，试验并同意。 由于实体身份不是复杂的价值对象，我们的建议是，您不应该担心太多.

Going back to the Order, it's time to update references to OrderId:

class Order

{

private $id; private $amount; private $firstName; private $lastName;

public function construct(

OrderId $anOrderId, Amount $amount, $aFirstName, $aLastName

) {

$this->id = $anOrderId;

$this->amount = $amount;

$this->firstName = $aFirstName;

$this->lastName = $aLastName;

}

public function id()

{

return $this->id;

}

public function firstName()

{

return $this->firstName;

}

public function lastName()

{

return $this->lastName;

}

public function amount()

{

return $this->amount;

}

}

我们的实体具有使用价值对象建模的身份。 让我们考虑创建OrderId的不同方式.

## 身份操作

如前所述，实体的身份是它的定义。 那么，处理它是实体的一个重要方面。 通常有四种方法来定义实体的身份：持久性机制提供了身份，客户端提供了身份，应用程序本身提供了身份，或者另一个有界的上下文提供了身份.

### 持久性机制产生身份

通常，生成身份的最简单方法是将其委托给持久性机制，因为绝大多数持久性机制支持某种类型的身份生成 - 例如MySQL的AUTO\_INCREMENT属性或Postgres和Oracle序列。 这虽然很简单，但有一个主要缺点：在坚持它之前，我们不会拥有实体的身份。 因此，在某种程度上，如果我们使用持久化机制生成的标识进行操作，我们会将Identity操作与基础持久性存储:

CREATE TABLE `orders` (

`id` int(11) NOT NULL auto\_increment,

`amount` decimal (10,5) NOT NULL,

`first\_name` varchar(100) NOT NULL,

`last\_name` varchar(100) NOT NULL, PRIMARY KEY (`id`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4\_unicode\_ci;

And then we might consider this code:

namespace Ddd\Identity\Domain\Model;

class Person

{

private $identificationNumber; private $firstName;

private $lastName;

public function construct(

$anIdentificationNumber, $aFirstName, $aLastName

) {

$this->identificationNumber = $anIdentificationNumber;

$this->firstName = $aFirstName;

$this->lastName = $aLastName;

}

public function identificationNumber()

{

return $this->identificationNumber;

}

public function firstName()

{

return $this->firstName;

}

public function lastName()

{

return $this->lastName;

}

}

如果您曾尝试构建自己的ORM，那么您已经遇到过这种情况。 创建新人的方法是什么？ 如果数据库要生成标识，我们是否必须在构造函数中传递它？ 什么时候以及在哪里使用其身份更新人物？ 如果我们最终不坚持实体会发生什么?

#### 代替身份

有时，当使用ORM将实体映射到持久性存储时，会施加一些约束 - 例如，如果使用IDENTITY生成器策略，则Doctrine需要一个整数字段。 如果它需要另一种身份，这可能与域模型冲突.

处理这种情况的最简单方法是使用图层超类型，其中为持久性存储创建的标识字段被放置:

namespace Ddd\Common\Domain\Model;

abstract class IdentifiableDomainObject

{

private $id;

protected function id()

{

return $this->id;

}

protected function setId($anId)

{

$this->id = $anId;

}

}

namespace Acme\Billing\Domain;

use Acme\Common\Domain\IdentifiableDomainObject; class Order extends IdentifiableDomainObject

{

private $orderId;

public function orderId()

{

if (null === $this->orderId) {

$this->orderId = new OrderId($this->id());

}

return $this->orderId;

}

}

#### Active Record Vs. Data Mapper for Rich Domain Models

Every project always faces the decision of which ORM should be used. There are a lot of good ORMs for PHP out there: Doctrine, Propel, Eloquent, Paris, and many more.

Most of them are [Active Record](http://www.martinfowler.com/eaaCatalog/activeRecord.html) implementations. An Active Record implementation is fine mostly for CRUD applications, but it's not the ideal solution for Rich Domain Models for the following reasons:

The Active Record pattern assumes a one-to-one relation between an Entity and a database table. So it couples the design of the database to the design of the object system**.** And in a Rich Domain Model, sometimes Entities are constructed with information that may come from different data sources.

Advanced things like collections and inheritance are tricky to implement. Most of the implementations force the use, through inheritance, of some sort of constructions that impose several conventions. This can lead to persistence

leakage into the Domain Model by coupling the Domain Model with the ORM.

The only Active Record implementation we've seen that doesn't impose inheriting from a base class is [Castle ActiveRecord](http://docs.castleproject.org/Active%20Record.MainPage.ashx) from [Castle Project](http://www.castleproject.org/), a

.NET framework. While this leads to some degree of separation between persistence and Domain concerns in the produced Entities, it doesn't decouple the low-level persistence details from high-level Domain design.

As mentioned in the previous chapter, currently the best ORM for PHP is [Doctrine](http://doctrine-project.org/) , which is an implementation of the [Data Mapper pattern](http://www.martinfowler.com/eaaCatalog/dataMapper.html). Data Mapper decouples persistence concerns from Domain concerns, leading to persistence-free Entities. This makes the tool the best for someone wanting to build a Rich Domain Model.

### Client Provides Identity

Sometimes, when dealing with certain Domains, the Identities come naturally, with the client consuming the Domain Model. This is likely the ideal case, because the Identity can be modeled easily. Let's take a look at the book-selling market:

namespace Ddd\Catalog\Domain\Model\Book;

class ISBN

{

private $isbn;

private function construct($anIsbn)

{

$this->setIsbn($anIsbn);

}

private function setIsbn($anIsbn)

{

$this->assertIsbnIsValid($anIsbn, 'The ISBN is invalid.');

$this->isbn = $anIsbn;

}

public static function create($anIsbn)

{

return new static($anIsbn);

}

private function assertIsbnIsValid($anIsbn, $string)

{

// ... Validates an ISBN code

}

}

According to [Wikipedia](https://en.wikipedia.org/wiki/International_Standard_Book_Number): The **International Standard Book Number** (**ISBN**) is a unique numeric commercial book identifier. An ISBN is assigned to each edition and variation (except re-printings) of a book. For example, an e-book, a paperback and a hardcover edition of the same book would each have a different ISBN. The ISBN is 13 digits long if assigned on or after 1 January 2007, and 10 digits long if assigned before 2007. The method of assigning an ISBN is nation-based and varies from country to country, often depending on how large the publishing industry is within a country.

The cool thing about the ISBN is that it's already defined in the Domain, it's a valid identifier because it's unique, and it can be easily validated. This is a good example of an Identity provided by the client:

class Book

{

private $isbn; private $title;

public function construct(ISBN $anIsbn, $aTitle)

{

$this->isbn = $anIsbn;

$this->title = $aTitle;

}

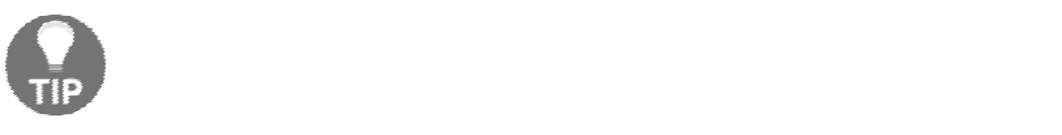
}

Now, it's just a matter of using it:

$book = new Book( ISBN::create('...'),

'Domain-Driven Design in PHP'

);



**Exercise**

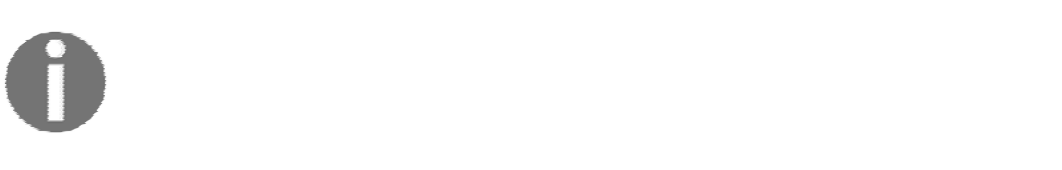
Think about other Domains where Identities are built in and model one.

### Application Generates Identity

If the client can't provide the Identity generally, the preferred way to handle the Identity operation is to let the application generate the Identities, usually through a UUID. This is our recommended approach in the case that you don't have a scenario as shown in the previous section.

According to [Wikipedia](https://en.wikipedia.org/wiki/Universally_unique_identifier):

The intent of UUIDs is to enable distributed systems to uniquely identify information without significant central coordination. In this context the word unique should be taken to mean *practically unique* rather than *guaranteed unique*. Since the identifiers have a finite size, it is possible for two differing items to share the same identifier. This is a form of hash collision. The identifier size and generation process need to be selected so as to make this sufficiently improbable in practice. Anyone can create a UUID and use it to identify something with reasonable confidence that the same identifier will never be unintentionally created by anyone to identify something else. Information labeled with UUIDs can therefore be later combined into a single database without needing to resolve identifier (ID) conflicts.



There are several libraries in PHP that generate UUIDs, and they can be found at Packagist: [https://packagist.org/search/q=uuid](https://packagist.org/search/?q=uuid). The best recommendation is the one developed by Ben Ramsey at the following link: <https://github.com/ramsey/uuid>because it has tons of watchers on GitHub and millions of installations on Packagist.

The preferred place to put the creation of the Identity would be inside a Repository (we'll go deeper into this in the [Chapter 10](#_bookmark315), *Repositories*:

namespace Ddd\Billing\Domain\Model\Order;

interface OrderRepository

{

public function nextIdentity(); public function add(Order $anOrder);

public function remove(Order $anOrder);

}

When using Doctrine, we'll need to create a custom Repository that implements such an interface. It will basically create the new Identity and use the EntityManager in order to persist and delete Entities. A small variation is to put the nextIdentity implementation into the interface that will become an abstract class:

namespace Ddd\Billing\Infrastructure\Domain\Model\Order;

use Ddd\Billing\Domain\Model\Order\Order; use Ddd\Billing\Domain\Model\Order\OrderId;

use Ddd\Billing\Domain\Model\Order\OrderRepository; use Doctrine\ORM\EntityRepository;

class DoctrineOrderRepository

extends EntityRepository implements OrderRepository

{

public function nextIdentity()

{

return OrderId::create();

}

public function add(Order $anOrder)

{

$this->getEntityManager()->persist($anOrder);

}

public function remove(Order $anOrder)

{

$this->getEntityManager()->remove($anOrder);

}

}

Let's quickly review the final implementation of the OrderId Value Object:

namespace Ddd\Billing\Domain\Model\Order; use Ramsey\Uuid\Uuid;

class OrderId

{

private $id;

private function construct($anId = null)

{

$this->id = $id  :Uuid::uuid4()->toString();

}

public static function create($anId = null )

{

return new static($anId);

}

}

The main concern about this approach, as you'll see in the following sections, is how easy it is to persist Entities that contain Value Objects. However, mapping embedded Value Objects that are inside an Entity can be tricky, depending on the ORM.

### Other Bounded Context Generates Identity

This is likely the most complex Identity generation strategy because it forces a local Entity to be dependent not only on local Bounded Context events, but also on external Bounded Contexts events. So in terms of maintenance, the cost would be high.

The other Bounded Context provides an interface to select the Identity from the local Entity. It can take some of the exposed properties as its own.

When synchronization is needed between the Entities of the Bounded Contexts, it can usually be achieved with an Event-Driven Architecture on each of the Bounded Contexts that need to be notified when the original Entity is changed.

## Persisting Entities

Currently, as discussed earlier in the chapter, the best tool for saving Entity state to a persistent store is Doctrine ORM. Doctrine has several ways to specify Entity metadata: by annotations in Entity code, by XML, by YAML, or by plain PHP. In this chapter, we'll discuss in depth why annotations are not the best thing to use when mapping Entities.

### Setting Up Doctrine

First of all, we need to require Doctrine through Composer. At the root folder of the project, the command below has to be executed:

> php composer.phar require "doctrine/orm=^2.5"

Then, these lines will allow you to set up Doctrine:

require\_once '/path/to/vendor/autoload.php';

use Doctrine\ORM\Tools\Setup; use Doctrine\ORM\EntityManager;

$paths = ['/path/to/entity-files'];

$isDevMode = false;

// the connection configuration

$dbParams = [

'driver' => 'pdo\_mysql',

'user' => 'the\_database\_username', 'password' => 'the\_database\_password', 'dbname' => 'the\_database\_name',

];

$config = Setup::createAnnotationMetadataConfiguration($paths, $isDevMode);

$entityManager = EntityManager::create($dbParams, $config);

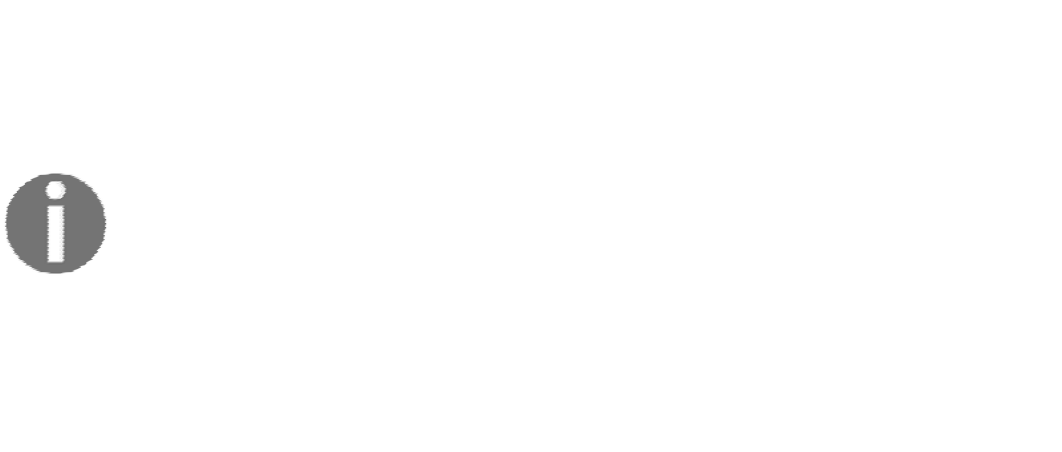
### Mapping Entities

By default, Doctrine's documentation presents the code examples using annotations. So we begin the code example using annotations and discussing why they should be avoided whenever possible.

To do so, we'll bring back the Order class discussed earlier in this chapter.

#### Mapping Entities Using Annotated Code

When Doctrine was released, a catchy way of showing how to map objects in the code examples was by using annotations.



**What's an annotation?**

An annotation is a special form of metadata. In PHP, it's put under source code comments. For example, *PHPDocumentor* makes use of annotations to build API information, and PHPUnit uses some annotations to specify data providers or to provide expectations about exceptions thrown by a piece of code:

class SumTest extends PHPUnit\_Framework\_TestCase

{

/\*\* @dataProvider aMethodName \*/ public function testAddition() {

//...

}

}

In order to map the Order Entity to the persistence store, the source code for the Order

should be modified to add the Doctrine annotations:

use Doctrine\ORM\Mapping\Entity; use Doctrine\ORM\Mapping\Id;

use Doctrine\ORM\Mapping\GeneratedValue; use Doctrine\ORM\Mapping\Column;

/\*\* @Entity \*/ class Order

{

/\*\* @Id @GeneratedValue(strategy="AUTO") \*/ private $id;

/\*\* @Column(type="decimal", precision="10", scale="5") \*/ private $amount;

/\*\* @Column(type="string") \*/ private $firstName;

/\*\* @Column(type="string") \*/ private $lastName;

public function construct( Amount $anAmount,

$aFirstName,

$aLastName

) {

$this->amount = $anAmount;

$this->firstName = $aFirstName;

$this->lastName = $aLastName;

}

public function id()

{

return $this->id;

}

public function firstName()

{

return $this->firstName;

}

public function lastName()

{

return $this->lastName;

}

public function amount()

{

return $this->amount;

}

}

Then, to persist the Entity to the persistent store, it's just as easy to do the following:

$order = new Order(

new Amount(15, Currency::EUR()), 'AFirstName',

'ALastName'

);

$entityManager->persist($order);

$entityManager->flush();

At first glance, this code looks simple, and this can be an easy way to specify mapping information. But it comes at a cost. What's odd about the final code?

First of all, Domain concerns are mixed with Infrastructure concerns. Order is a Domain concept, whereas Table, Column, and so on are infrastructure concerns.

As a result, this Entity is tightly coupled to the mapping information specified by the annotations in the source code. If the Entity were required to be persisted using another Entity manager and with different mapping metadata, this wouldn't be possible.

Annotations tend to lead to side effects and tight coupling, so it would be better to not use them.

So what's the best way to specify mapping information? The best way is the one that allows you to separate the mapping information from the Entity itself. This can be achieved by using XML mapping, YAML mapping, or PHP mapping. In this book, we're going to cover XML mapping.

#### Mapping Entities Using XML

To map the Order Entity using the XML mapping, the setup code of Doctrine should be altered slightly:

require\_once '/path/to/vendor/autoload.php';

use Doctrine\ORM\Tools\Setup; use Doctrine\ORM\EntityManager;

$paths = ['/path/to/mapping-files'];

$isDevMode = false;

// the connection configuration

$dbParams = [

'driver' => 'pdo\_mysql',

'user' => 'the\_database\_username',

'password' => 'the\_database\_password', 'dbname' => 'the\_database\_name',

];

$config = Setup::createXMLMetadataConfiguration($paths, $isDevMode);

$entityManager = EntityManager::create($dbParams, $config);

The mapping file should be created on the path where Doctrine will search for the mapping files, and the mapping files should be named after the fully qualified class name, replacing the backslashes \ with dots. Consider the following:

Acme\Billing\Domain\Model\Order

The preceding illustration would have the mapping file named like this:

Acme.Billing.Domain.Model.Order.dcm.xml

Additionally, it's convenient that all the mapping files use a special XML Schema created specially for specifying mapping information:

https://raw.github.com/doctrine/doctrine2/master/doctrine-mapping.xsd

#### Mapping Entity Identity

Our Identity, OrderId, is a Value Object. As seen in the previous chapter, there are different approaches for mapping a Value Object using Doctrine, embeddables, and custom types. When Value Objects are used as Identities, the best option is custom types.

An interesting new feature in *Doctrine 2.5* is that it's now possible to use Objects as identifiers for Entities, so long as they implement the magic method toString(). So we can add toString to our Identity Value Objects and use them in our mappings:

namespace Ddd\Billing\Domain\Model\Order; use Ramsey\Uuid\Uuid;

class OrderId

{

// ...

public function toString()

{

return $this->id;

}

}

Check the implementation of the Doctrine custom types. They inherit from GuidType, so their internal representation will be a UUID. We need to specify the database native translation. Then we need to register our custom types before we use them. If you need help with these steps, [Custom Mapping Types](http://doctrine-orm.readthedocs.io/projects/doctrine-orm/en/latest/cookbook/custom-mapping-types.html) is a good reference.

use Doctrine\DBAL\Platforms\AbstractPlatform; use Doctrine\DBAL\Types\GuidType;

class DoctrineOrderId extends GuidType

{

public function getName()

{

return 'OrderId';

}

public function convertToDatabaseValue(

$value, AbstractPlatform $platform

) {

return $value->id();

}

public function convertToPHPValue(

$value, AbstractPlatform $platform

) {

return new OrderId($value);

}

}

Lastly, we'll set up the registration of custom types. Again, we have to update our bootstrapping:

require\_once '/path/to/vendor/autoload.php';

// ...

\Doctrine\DBAL\Types\Type::addType( 'OrderId',

'Ddd\Billing\Infrastructure\Domain\Model\DoctrineOrderId'

);

$config = Setup::createXMLMetadataConfiguration($paths, $isDevMode);

$entityManager = EntityManager::create($dbParams, $config);

#### Final Mapping File

With all the changes, we're finally ready, so let's take a look at the final mapping file. The most interesting detail is to check how the id gets mapped with our defined custom type for OrderId:

<xml version="1.0" encoding="UTF-8">

<doctrine-mapping

[xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping"](http://doctrine-project.org/schemas/orm/doctrine-mapping) [xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"](http://www.w3.org/2001/XMLSchema-instance) xsi:schemaLocation="

<http://doctrine-project.org/schemas/orm/doctrine-mapping> https://raw.github.com/doctrine/doctrine2/master/doctrine-mapping.xsd">

<entity

name="Ddd\Billing\Domain\Model\Order" table="orders">

<id name="id" column="id" type="OrderId" />

<field

name="amount" type="decimal" nullable="false" scale="10" precision="5"

/>

<field

name="firstName" type="string" nullable="false"

/>

<field

name="lastName" type="string" nullable="false"

/>

</entity>

</doctrine-mapping>

## Testing Entities

It's relatively easy to test Entities, simply because they're plain old PHP classes with actions derived from the Domain concept they represent. The focus of the test should be the invariants that the Entity protects, because the behavior on the Entities will likely be modeled around those invariants.

For example, and for the sake of simplicity, suppose a Domain Model for a blog is needed. A possible one could be this:

class Post

{

private $title; private $content; private $status; private $createdAt; private $publishedAt;

public function construct($aContent, $title)

{

$this->setContent($aContent);

$this->setTitle($title);

$this->unpublish();

$this->createdAt(new DateTimeImmutable());

}

private function setContent($aContent)

{

$this->assertNotEmpty($aContent);

$this->content = $aContent;

}

private function setTitle($aPostTitle)

{

$this->assertNotEmpty($aPostTitle);

$this->title = $aPostTitle;

}

private function setStatus(Status $aPostStatus)

{

$this->assertIsAValidPostStatus($aPostStatus);

$this->status = $aPostStatus;

}

private function createdAt(DateTimeImmutable $aDate)

{

$this->assertIsAValidDate($aDate);

$this->createdAt = $aDate;

}

private function publishedAt(DateTimeImmutable $aDate)

{

$this->assertIsAValidDate($aDate);

$this->publishedAt = $aDate;

}

public function publish()

{

$this->setStatus(Status::published());

$this->publishedAt(new DateTimeImmutable());

}

public function unpublish()

{

$this->setStatus(Status::draft());

$this->publishedAt = null ;

}

public function isPublished()

{

return $this->status->equalsTo(Status::published());

}

public function publicationDate()

{

return $this->publishedAt;

}

}

class Status

{

const PUBLISHED = 10;

const DRAFT = 20; private $status;

public static function published()

{

return new self(self::PUBLISHED);

}

public static function draft()

{

return new self(self::DRAFT);

}

private function construct($aStatus)

{

$this->status = $aStatus;

}

public function equalsTo(self $aStatus)

{

return $this->status === $aStatus->status;

}

}

In order to test this Domain Model, we must ensure the test covers all the Post invariants:

class PostTest extends PHPUnit\_Framework\_TestCase

{

/\*\* @test \*/

public function aNewPostIsNotPublishedByDefault()

{

$aPost = new Post(

'A Post Content', 'A Post Title'

);

$this->assertFalse(

$aPost->isPublished()

);

$this->assertNull(

$aPost->publicationDate()

);

}

/\*\* @test \*/

public function aPostCanBePublishedWithAPublicationDate()

{

$aPost = new Post(

'A Post Content', 'A Post Title'

);

$aPost->publish();

$this->assertTrue(

$aPost->isPublished()

);

$this->assertInstanceOf( 'DateTimeImmutable',

$aPost->publicationDate()

);

}

}

### DateTimes

Because DateTimes are widely used in Entities, we think it's important to point out specific approaches on unit testing Entities that have fields with date types. Consider that a Post is new if it was created within the last 15 days:

class Post

{

const NEW\_TIME\_INTERVAL\_DAYS = 15;

// ...

private $createdAt;

public function construct($aContent, $title)

{

// ...

$this->createdAt(new DateTimeImmutable());

}

// ...

public function isNew()

{

return

(new DateTimeImmutable())

->diff($this->createdAt)

->days <= self::NEW\_TIME\_INTERVAL\_DAYS;

}

}

The isNew() method needs to compare two DateTimes; it's a comparison between the date when the Post was created and today's date. We compute the difference and check if it's less than the specified amount of days. How do we unit test the isNew() method? As we demonstrated in the implementation, it's difficult to reproduce specific flows in our test suites. What options do we have?

#### 将所有日期作为参数传递

一种可能的选择是在需要时将所有日期作为参数传递:

class Post

{

// ...

public function construct($aContent, $title, $createdAt = null)

{

// ...

$this->createdAt($createdAt : new DateTimeImmutable());

}

// ...

public function isNew($today = null)

{

return

($today  :new DateTimeImmutable())

->diff($this->createdAt)

->days <= self::NEW\_TIME\_INTERVAL\_DAYS;

}

}

This is the easiest approach for unit testing purposes. Just pass different pairs of dates to test all possible scenarios with 100 percent coverage. However, if you consider the client code that's creating and asking for the isNew() method result, things don't look so nice. The resulting code can be a bit weird because of always passing today's DateTime:

$aPost = new Post( 'Hello world!', 'Hi',

new DateTimeImmutable()

);

$aPost->isNew(

new DateTimeImmutable()

);

#### Test Class

Another alternative is to use the Test Class pattern. The idea is to extend the Post class with a new one that we can manipulate to force specific scenarios. This new class is going to be used only for unit testing purposes. The bad news is that we have to modify the original Post class a bit, extracting some methods and changing some fields and methods from private to protected. Some developers may worry about increasing the visibility of class properties just because of testing reasons. However, we think that in most cases, it's worth it:

class Post

{

protected $createdAt;

public function isNew()

{

return

($this->today())

->diff($this->createdAt)

->days <= self::NEW\_TIME\_INTERVAL\_DAYS;

}

protected function today()

{

return new DateTimeImmutable();

}

protected function createdAt(DateTimeImmutable $aDate)

{

$this->assertIsAValidDate($aDate);

$this->createdAt = $aDate;

}

}

As you can see, we've extracted the logic for getting today's date into the today() method. This way, by applying the Template Method pattern, we can change its behavior from a derived class. Something similar happens with the createdAt method and field. Now they're protected, so they can be used and overridden in derived classes:

class PostTestClass extends Post

{

private $today;

protected function today()

{

return $this->today;

}

public function setToday($today)

{

$this->today = $today;

}

}

With these changes, we can now test our original Post class through testing

PostTestClass:

class PostTest extends PHPUnit\_Framework\_TestCase

{

// ...

/\*\* @test \*/

public function aPostIsNewIfIts15DaysOrLess()

{

$aPost = new PostTestClass( 'A Post Content' ,

'A Post Title'

);

$format = 'Y-m-d';

$dateString = '2016-01-01';

$createdAt = DateTimeImmutable::createFromFormat(

$format,

$dateString

);

$aPost->createdAt($createdAt);

$aPost->setToday(

$createdAt->add(

new DateInterval('P15D')

)

);

$this->assertTrue(

$aPost->isNew()

);

$aPost->setToday(

$createdAt->add(

new DateInterval('P16D')

)

);

$this->assertFalse(

$aPost->isNew()

);

}

}

Just one last small detail: with this approach, it's impossible to achieve 100 percent coverage on the Post class, because the today() method is never going to be executed. However, it can be covered by other tests.

#### External Fake

Another option is to wrap calls to the DateTimeImmutable constructor or named constructors using a new class and some static methods. In doing so, we can statically change the result of those methods to behave differently based on specific testing scenarios:

class Post

{

// ...

private $createdAt;

public function construct($aContent, $title)

{

// ...

$this->createdAt(MyCustomDateTimeBuilder::today());

}

// ...

public function isNew()

{

return

(MyCustomDateTimeBuilder::today())

->diff($this->createdAt)

->days <= self::NEW\_TIME\_INTERVAL\_DAYS;

}

}

For getting today's DateTime, we now use a static call to MyCustomDateTimeBuilder::today(). This class also has some setter methods to fake the result to return in the next calls:

class PostTest extends PHPUnit\_Framework\_TestCase

{

// ...

/\*\* @test \*/

public function aPostIsNewIfIts15DaysOrLess()

{

$createdAt = DateTimeImmutable::createFromFormat( 'Y-m-d',

'2016-01-01'

);

MyCustomDateTimeBuilder::setReturnDates( [

$createdAt,

$createdAt->add(

new DateInterval('P15D')

),

$createdAt->add(

new DateInterval('P16D')

)

]

);

$aPost = new Post(

'A Post Content' , 'A Post Title'

);

$this->assertTrue(

$aPost->isNew()

);

$this->assertFalse(

$aPost->isNew()

);

}

}

The main problem with this approach is it's coupled statically with an object. Depending on your use case, it'll also be tricky to create a flexible fake object.

#### Reflection

You can also use reflection techniques for building a new Post class with custom dates. Consider [Mimic](https://github.com/keyvanakbary/mimic), a simple functional library for object prototyping, data hydration, and data exposition:

namespace Domain; use mimic as m;

class ComputerScientist { private $name; private $surname;

public function construct($name, $surname)

{

$this->name = $name;

$this->surname = $surname;

}

public function rocks()

{

return $this->name . ' ' . $this->surname . ' rocks!';

}

}

assert(m\prototype('Domain\ComputerScientist') instanceof Domain\ComputerScientist);

m\hydrate('Domain\ComputerScientist', [ 'name' =>'John' , 'surname'=>'McCarthy'

])->rocks(); //John McCarthy rocks!

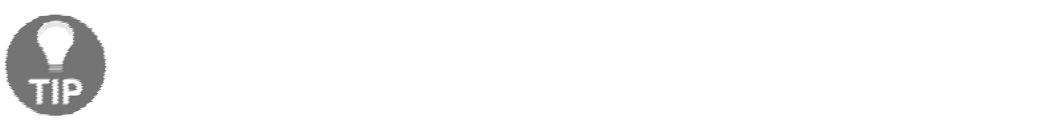
assert(m\expose(

new Domain\ComputerScientist('Grace', 'Hopper')) == [

'name' => 'Grace' , 'surname' => 'Hopper'

]

)



**Share and Discuss**

Discuss with your workmates how to properly unit test your Entities with fixed DateTimes and come up with additional alternatives.

If you want to know more about testing patterns and approaches, take a look at the book

*xUnit Test Patterns: Refactoring Test Code* by Gerard Meszaros.

## Validation

Validation is a highly important process in our Domain Model. It checks not only for the correctness of attributes, but also for that of entire objects and the composition of those objects. Different levels of validation are required in order to keep this Model in a valid state. Just because an object consists of valid attributes (on a per basis) doesn't necessarily mean the object (as a whole) is valid. And the opposite is true: valid objects don't necessarily equal valid compositions.

### Attribute Validation

Some people understand validation as the process whereby a service validates the state of a given object. In this case, the validation conforms to a [Design-by-contract](http://en.wikipedia.org/wiki/Design_by_contract) approach, which consists of preconditions, postconditions, and invariants. One such way to protect a single attribute is by using [Chapter 3](#_bookmark40), *Value Objects*. In order to make our design more flexible for change, we focus only on asserting Domain preconditions that must be met.

Here, we'll be using guards as an easy way of validating the preconditions:

class Username

{

const MIN\_LENGTH = 5; const MAX\_LENGTH = 10;

const FORMAT = '/^[a-zA-Z0-9\_]+$/'; private $username;

public function construct($username)

{

$this->setUsername($username);

}

private setUsername($username)

{

$this->assertNotEmpty($username);

$this->assertNotTooShort($username);

$this->assertNotTooLong($username);

$this->assertValidFormat($username);

$this->username = $username;

}

private function assertNotEmpty($username)

{

if (empty($username)) {

throw new InvalidArgumentException('Empty username');

}

}

private function assertNotTooShort($username)

{

if (strlen($username) < self::MIN\_LENGTH) { throw new InvalidArgumentException(sprintf(

'Username must be %d characters or more', self::MIN\_LENGTH

));

}

}

private function assertNotTooLong($username)

{

if (strlen( $username) > self::MAX\_LENGTH) { throw new InvalidArgumentException(sprintf(

'Username must be %d characters or less', self::MAX\_LENGTH

));

}

}

private function assertValidFormat($username)

{

if (preg\_match(self:: FORMAT, $username) !== 1) { throw new InvalidArgumentException(

'Invalid username format'

);

}

}

}

As you can see in the example above, there are four preconditions that must be satisfied in order to build a Username Value Object. It:

Must not be empty

Must be at least 5 characters Must be less than 10 characters

Must follow a format of alphanumeric characters or underscores

If all the preconditions are met, the attribute will be set and the object will be successfully built. Otherwise, an InvalidArgumentException will be raised, execution will be halted, and the client will be shown an error.

Some developers may consider this kind of validation defensive programming. However, we're not checking that the input is a string or that nulls are not permitted. We can't avoid people using our code incorrectly, but we can control the correctness of our Domain state. As seen in the [Chapter 3](#_bookmark40), *Value Objects*, validation can help us with security too.

[Defensive programming](https://en.wikipedia.org/wiki/Defensive_programming) isn't a bad thing. In general, it makes sense when developing components or libraries that are going to be used as a third party in other projects.

However, when developing your own Bounded Context, those extra paranoid checks (nulls, basic types, type hinting, and so on.) can be avoided to increase development speed by relying on the coverage of your unit test suite.

### Entire Object Validation

There are times when an object composed of valid properties, as a whole, can still be deemed invalid. It can be tempting to add this kind of validation to the object itself, but generally this is an anti-pattern. Higher-level validation changes at a rhythm different than that of the object logic itself. Also, it's good practice to separate these responsibilities.

The validation informs the client about any errors that have been found or collects the results to be reviewed later, as sometimes we don't want to stop the execution at the first sign of trouble.

An abstract and reusable Validator could be something like this:

abstract class Validator

{

private $validationHandler;

public function construct(ValidationHandler $validationHandler)

{

$this->validationHandler = $validationHandler;

}

protected function handleError($error)

{

$this->validationHandler->handleError($error);

}

abstract public function validate();

}

As a concrete example, we want to validate an entire Location object, composed of valid Country, City, and Postcode Value Objects. However, these individual values might be in an invalid state at the time of validation. Maybe the city doesn't form part of the country, or maybe the postcode doesn't follow the city format:

class Location

{

private $country; private $city; private $postcode;

public function construct(

Country $country, City $city, Postcode $postcode

) {

$this->country = $country;

$this->city = $city;

$this->postcode = $postcode;

}

public function country()

{

return $this->country;

}

public function city()

{

return $this->city;

}

public function postcode()

{

return $this->postcode;

}

}

The validator checks the state of the Location object in its entirety, analyzing the meaning of the relationships between properties:

class LocationValidator extends Validator

{

private $location;

public function construct(

Location $location, ValidationHandler $validationHandler

) {

parent:: construct($validationHandler);

$this->location = $location;

}

public function validate()

{

if (!$this->location->country()->hasCity(

$this->location->city()

)) {

}

$this->handleError('City not found');

if (!$this->location->city()->isPostcodeValid(

$this->location->postcode()

)) {

}

}

}

$this->handleError('Invalid postcode');

Once all the properties have been set, we're able to validate the Entity, most likely after some described process. On the surface, it looks as if the Location validates itself. However, this isn't the case. The Location class delegates this validation to a concrete validator instance, splitting these two clear responsibilities:

class Location

{

// ...

public function validate(ValidationHandler $validationHandler)

{

$validator = new LocationValidator($this, $validationHandler);

$validator->validate();

}

}

#### Decoupling Validation Messages

With some minor changes to our existing implementation, we're able to decouple the validation messages from the validator:

class LocationValidationHandler implements ValidationHandler

{

public function handleCityNotFoundInCountry();

public function handleInvalidPostcodeForCity();

}

class LocationValidator

{

private $location;

private $validationHandler;

public function construct( Location $location,

LocationValidationHandler $validationHandler

) {

$this->location = $location;

$this->validationHandler = $validationHandler;

}

public function validate()

{

if (!$this->location->country()->hasCity(

$this->location->city()

)) {

}

$this->validationHandler->handleCityNotFoundInCountry();

if (! $this->location->city()->isPostcodeValid(

$this->location->postcode()

)) {

}

}

}

$this->validationHandler->handleInvalidPostcodeForCity();

We also need to change the signature of the validation method to the following:

class Location

{

// ...

public function validate( LocationValidationHandler $validationHandler

) {

$validator = new LocationValidator($this, $validationHandler);

$validator->validate();

}

}

### Validating Object Compositions

Validating object compositions can be complicated. As such, the preferred way of achieving this is through a Domain Service. The service then communicates with Repositories in order to retrieve the valid Aggregate. Due to the likely complex object graphs that can be created, an Aggregate could be in an intermediate state, requiring other Aggregates to be validated beforehand. We can use Domain Events to notify other parts of the system that a particular element has been validated.

## Entities and Domain Events

We'll explore [Chapter 6](#_bookmark178), *Domain-Events* in future chapters; however, it's important to highlight that operations performed on Entities can fire Domain Events. This approach is used to communicate the Domain change to other parts of the Application, or even to other Applications, as you'll see in [Chapter 12](#_bookmark412), *Integrating Bounded Contexts*:

class Post

{

// ...

public function publish()

{

$this->setStatus( Status::published()

);

$this->publishedAt(new DateTimeImmutable()); DomainEventPublisher::instance()->publish(

new PostPublished($this->id)

);

}

public function unpublish()

{

$this->setStatus( Status::draft()

);

$this-> publishedAt = null; DomainEventPublisher::instance()->publish(

new PostUnpublished($this->id)

);

}

// ...

}

Domain Events can even be fired when a new instance of our Entity is created:

class User

{

// ...

public function construct(UserId $userId, $email, $password)

{

$this->setUserId($userId);

$this->setEmail($email);

$this->setPassword($password);

DomainEventPublisher::instance()->publish( new UserRegistered($this->userId)

);

}

}

## Wrap-Up

Some concepts in the Domain demand Identity — that is, changes to their internal states don't change their own unique identities. We've seen how modeling Identity as a Value Object brings benefits like immutability, in addition to logic for operating the Identity itself. We've also shown several ways of providing Identity, restated in the following pointers:

Persistence mechanism: Easy to implement, but you won't have the Identity before persisting the object, which delays and complicates event propagation. Surrogate ID: Some ORMs require an extra field on your Entity to map the

Identity with the persisting mechanism.

Provided by the client: Sometimes the Identity fits a Domain concept and you can model it inside your Domain.

Generated by the application: You can use a library to generate IDs. Generated by a Bounded Context: Probably the most complex strategy. Other Bounded Contexts provide an interface for generating Identities.

We've seen and discussed Doctrine as a persistence mechanism, we've looked at the drawbacks of using the Active Record pattern, and finally, we've checked different levels of Entity validation:

Attribute validation: Check for specifics inside the object state through preconditions, postconditions, and invariants.

Entire object validation: Look for consistency of an object as a whole. Extracting the validation to an external service is a good practice.

Object compositions: Complex object compositions could be validated through Domain Services. A good way of communicating this to the rest of the application is through Domain Events.

# Services

****

你已经看到了实体和价值对象。 作为基本的构建块，它们应该包含任何应用程序的大部分业务逻辑。 但是，有些情况下实体和值对象不是最佳解决方案。 让我们来看看埃里克埃文斯在他的书“领域驱动设计：解决软件核心的复杂性：

当域中的重要流程或转换不是实体或值对象的自然职责时，请将作为独立接口声明为服务的操作添加到模型中。 根据模型的语言定义界面并确保操作名称是泛在语言的一部分。 使服务无状态.

因此，当需要表示操作时，实体和值对象不是最好的地方，您应该考虑将这些操作建模为服务。 在

域驱动设计，通常会遇到三种不同类型的服务:

**应用程序服务：在标量类型上操作，将它们转换为域类型。 标量类型可以被认为是域模型未知的任何类型。 这包括不属于域的原始类型和类型。 我们将在本章中提供一个概述，但为了深入了解这个主题，请查看第11章，应用程序**.

**域服务：仅在属于域的类型上操作。 它们包含无处不在的语言中可以找到的有意义的概念。 他们持有不适合价值对象或实体的操作**.

**基础设施服务：是否满足基础设施问题的操作，例如发送电子邮件和记录有意义的数据。 就六角形建筑而言，它们居住在域边界之外**.

## 应用服务

应用程序服务是外部世界和域逻辑之间的中间件。 这种机制的目的是将来自外部世界的命令转换成有意义的域指令.

让我们考虑用户注册我们的平台用例。 从外部入手方式开始：从传递机制开始，我们需要为我们的域操作组合输入请求。 使用像Symfony这样的框架作为交付机制，代码看起来像这样:

class SignUpController extends Controller

{

public function signUpAction(Request $request)

{

$signUpService = new SignUpUserService(

$this->get('user\_repository')

);

try {

$response = $signUpService->execute(new SignUpUserRequest(

$request->request->get('email'),

$request->request->get('password')

));

} catch (UserAlreadyExistsException $e) {

return $this->render('error.html.twig', $response);

}

return $this->render('success.html.twig', $response);

}

}

正如你所看到的，我们创建了一个新的Application Services实例，传递了所有需要的依赖关系 - 在本例中是一个UserRepository。 UserRepository是一个可以用任何特定技术实现的接口（例如：MySQL，Redis，Elasticsearch）。 然后，我们为我们的应用服务构建一个请求对象，以便从业务逻辑中抽象出交付机制 - 在本例中是一个Web请求。 最后，我们执行应用程序服务，获取响应，并使用该响应来呈现结果。 在域方面，让我们来检查一下应用服务的一个可能的实现，它协调实现用户注册用例的逻辑:

class SignUpUserService

{

private $userRepository;

public function construct(UserRepository $userRepository)

{

$this->userRepository = $userRepository;

}

public function execute(SignUpUserRequest $request)

{

$user = $this->userRepository->userOfEmail($request->email); if ($user) {

throw new UserAlreadyExistsException();

}

$user = new User(

$this->userRepository->nextIdentity(),

$request->email,

$request->password

);

$this->userRepository->add($user);

return new SignUpUserResponse($user);

}

}

代码中的所有内容都是关于我们想要解决的域问题，而不是我们用来解决它的特定技术。 采用这种方法，我们可以将高级别策略与低级别实施细节分离。 传递机制与域之间的通信由数据结构DTO承载，我们在第2章“建筑风格”:

class SignUpUserRequest

{

public $email; public $password;

public function construct($email, $password)

{

$this->email = $email;

$this->password = $password;

}

}

返回内容有不同的策略，但现在考虑我们不应该返回我们的实体，以便它们不能从我们的应用服务之外修改。

这就是为什么返回另一个DTO信息而不是整个实体的原因。 我们来看一个简单的例子：

class SignUpUserResponse

{

public $id; public $email;

public function construct(User $user)

{

$this->id = $user->id();

$this->email = $user->email();

}

}

为了创建您的响应，您可以使用getter或公共实例变量。 应用程序服务应该注意事务范围和安全性。 但是，您将在第11章“应用程序”中深入了解与应用程序服务相关的这些和其他内容的更多细节.

## Domain Services

在与域专家的对话中，您将遇到无处不在的语言中的概念，这些概念不能完整地表示为实体或值对象，例如:

用户可以自己登录系统

购物车能够自己成为订单

前面的例子是两个具体的概念，它们都不能自然地绑定到实体或值对象。 进一步突出这种古怪，我们可以尝试按如下方式对行为建模:

class User

{

public function signUp($aUsername, $aPassword)

{

// ...

}

}

class Cart

{

public function createOrder()

{

// ...

}

}

在第一个实现的情况下，我们无法知道给定的用户名和密码与被调用的用户实例有关。 显然，这个操作不适合这个实体; 相反，它应该被提取到一个单独的类，使其意图明确。

考虑到这一点，我们可以创建一个域用户身份验证服务:

class SignUp

{

public function execute($aUsername, $aPassword)

{

// ...

}

}

同样，与第二个示例的情况一样，我们可以创建专门从提供的购物车创建订单的域服务:

class CreateOrderFromCart

{

public function execute(Cart $aCart)

{

// ...

}

}

域服务可以被定义为满足域任务的操作，并且自然不适合实体或值对象。 作为代表域中操作的概念，客户端应该使用域服务，而不管其运行历史。

域服务本身不具有任何形式的状态，因此域服务是无状态的操作.

## Domain Services and Infrastructure Services

建模域服务时遇到基础架构依赖性很常见

- 例如，在需要处理密码散列的认证机制的情况下。 在这种情况下，您可以使用分离接口，它允许定义多个哈希机制。 使用这种模式仍然可以为您提供明确的域和基础架构之间的关注点分离:

namespace Ddd\Auth\Domain\Model;

interface SignUp

{

public function execute($aUsername, $aPassword);

}

使用在Domain中找到的上述接口，我们可以在基础结构层中创建一个实现，如下所示:

namespace Ddd\Auth\Infrastructure\Authentication;

class DefaultHashingSignUp implements Ddd\Auth\Domain\Model\SignUp

{

private $userRepository;

public function construct(UserRepository $userRepository)

{

$this->userRepository = $userRepository;

}

public function execute($aUsername, $aPassword)

{

if (!$this->userRepository->has($aUsername)) {

throw UserDoesNotExistException::fromUsername($aUsername);

}

$aUser = $this->userRepository->byUsername($aUsername); if (!$this->isPasswordValidForUser($aUser, $aPassword)) {

throw new BadCredentialsException($aUser, $aPassword);

}

return $aUser;

}

private function isPasswordValidForUser( User $aUser, $anUnencryptedPassword

) {

return password\_verify($anUnencryptedPassword,$aUser->hash());

}

}

这是另一种基于MD5算法的实现:

namespace Ddd\Auth\Infrastructure\Authentication; use Ddd\Auth\Domain\Model\SignUp

class Md5HashingSignUp implements SignUp

{

const SALT = 'S0m3S4lT' ; private $userRepository;

public function construct(UserRepository $userRepository)

{

$this->userRepository = $userRepository;

}

public function execute($aUsername, $aPassword)

{

if (!$this->userRepository->has($aUsername)) { throw new InvalidArgumentException(

sprintf('The user "%s" does not exist.', $aUsername)

);

}

$aUser = $this->userRepository->byUsername($aUsername);

if ($this->isPasswordInvalidFor($aUser, $aPassword)) { throw new BadCredentialsException($aUser, $aPassword);

}

return $aUser;

}

private function salt()

{

return md5(self::SALT);

}

private function isPasswordInvalidFor( User $aUser, $anUnencryptedPassword

) {

$encryptedPassword = md5(

$anUnencryptedPassword . '\_' .$this->salt()

);

return $aUser->hash() !== $encryptedPassword;

}

}

选择此选项可以让我们在基础架构层有多个域服务接口实现。 换句话说，我们最终得到了几个基础架构域服务。 每个基础架构服务将负责处理不同的散列机制。 取决于实现，可以通过依赖注入容器轻松管理该用途 - 例如，通过Symfony的依赖注入组件:

<xml version="1.0">

<container

[xmlns="http://symfony.com/schema/dic/services"](http://symfony.com/schema/dic/services) [xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"](http://www.w3.org/2001/XMLSchema-instance) xsi:schemaLocation="

<http://symfony.com/schema/dic/services> [http://symfony.com/schema/dic/services/services-1.0.xsd">](http://symfony.com/schema/dic/services/services-1.0.xsd)

<services>

<service id="sign\_in" alias="sign\_in.default" />

<service id="sign\_in.default" class="Ddd\Auth\Infrastructure\Authentication

\DefaultHashingSignUp">

<argument type="service" id="user\_repository"/>

</service>

<service id="sign\_in.md5" class="Ddd\Auth\Infrastructure\Authentication

\Md5HashingSignUp">

<argument type="service" id="user\_repository"/>

</service>

</services>

</container>

If, in the future, we wish to handle a new type of hashing, we can simply start by implementing the Domain Service interface. Then it's a matter of declaring the service in the Dependency Injection container and replacing the service alias dependency with the newly created one.

### 代码重用问题

虽然前面描述的实现明确定义了关注点分离，但是每次我们希望实现新的哈希机制时，都需要重复密码验证算法。 解决这个问题的另一种方法是改进代码的重用，将这两项责任分开。 我们可以将密码散列逻辑提取到专门的类中，对所有定义的散列算法使用策略模式。 这样可以将设计保留为可扩展并关闭以进行修改:

namespace Ddd\Auth\Domain\Model;

class SignUp

{

private $userRepository; private $passwordHashing;

public function construct(

UserRepository $userRepository, PasswordHashing $passwordHashing

) {

$this->userRepository = $userRepository;

$this->passwordHashing = $passwordHashing;

}

public function execute($aUsername, $aPassword)

{

if (!$this->userRepository->has($aUsername)) { throw new InvalidArgumentException(

sprintf('The user "%s" does not exist.', $aUsername)

);

}

$aUser = $this->userRepository->byUsername($aUsername);

if ($this->isPasswordInvalidFor($aUser, $aPassword)) { throw new BadCredentialsException($aUser, $aPassword);

}

return $aUser;

}

private function isPasswordInvalidFor(User $aUser, $plainPassword)

{

return !$this->passwordHashing->verify(

$plainPassword,

$aUser->hash()

);

}

}

interface PasswordHashing

{

/\*\*

* @param string $password
* @param string $hash
* @return boolean

\*/

public function verify($plainPassword, hash);

}

Defining different hashing strategies is as easy as implementing the PasswordHashing

interface:

namespace Ddd\Auth\Infrastructure\Authentication;

class BasicPasswordHashing

implements \Ddd\Auth\Domain\Model\PasswordHashing

{

public function verify($plainPassword, $hash)

{

return password\_verify($plainPassword, $hash);

}

}

class Md5PasswordHashing

implements Ddd\Auth\Domain\Model\PasswordHashing

{

const SALT = 'S0m3S4lT' ;

public function verify($plainPassword, $hash)

{

return $hash === $this-> calculateHash($plainPassword);

}

private function calculateHash($plainPassword)

{

return md5($plainPassword . '\_' .$this-> salt());

}

private function salt()

{

return md5(self::SALT);

}

}

## Testing Domain Services

Given the user authentication example from multiple Domain Service implementations, it's extremely beneficial to be able to easily test the service. Typically, however, testing the Template Method implementations can be tricky. As a result, we'll be using a plain password hashing implementation for testing purposes:

class PlainPasswordHashing implements PasswordHashing

{

public function verify($plainPassword, $hash)

{

return $plainPassword === $hash;

}

}

Now we can test all cases in the Domain Service:

class SignUpTest extends PHPUnit\_Framework\_TestCase

{

private $signUp;

private $userRepository;

protected function setUp()

{

$this->userRepository = new InMemoryUserRepository();

$this->signUp = new SignUp(

$this->userRepository,

new PlainPasswordHashing()

);

}

/\*\*

* @test
* @expectedException InvalidArgumentException

\*/

public function itShouldComplainIfTheUserDoesNotExist()

{

$this->signUp->execute('test-username', 'test-password');

}

/\*\*

* @test
* @expectedException BadCredentialsException

\*/

public function itShouldTellIfThePasswordDoesNotMatch()

{

$this->userRepository->add(

new User(

'test-username', 'test-password'

)

);

$this->signUp->execute('test-username', 'no-matching-password')

}

/\*\*

* @test

\*/

public function itShouldTellIfTheUserMatchesProvidedPassword()

{

$this->userRepository->add( new User(

'test-username', 'test-password'

)

);

$this->assertInstanceOf( 'Ddd\Domain\Model\User\User',

$this->signUp->execute('test-username', 'test-password')

);

}

}

## 贫血领域模型与丰富的领域模型

您必须谨慎，不要过度使用系统中的域服务抽象。 遵循此路径可能会导致实体和值对象被剥离所有行为并成为单纯的数据容器。 这与面向对象编程的目标相反，它可以被认为是将数据和行为集中到称为对象的语义单元中，旨在表达真实世界的概念和问题。 域服务过度使用可以被认为是反模式，并被称为贫血域模型.

通常，在开始一个新的项目或功能时，很容易陷入数据建模的陷阱。 这通常包括认为每个数据库表具有直接的一对一对象形式表示。 但是，这种想法可能始终也可能不是确切的情况.

假设我们负责建模订单处理系统。 如果我们首先对数据进行建模，那么我们最终会得到一个像这样的SQL脚本:

CREATE TABLE `orders` (

`ID` INTEGER NOT NULL AUTO\_INCREMENT,

`CUSTOMER\_ID` INTEGER NOT NULL,

`AMOUNT` DECIMAL(17, 2) NOT NULL DEFAULT '0.00',

`STATUS` TINYINT NOT NULL DEFAULT 0,

`CREATED\_AT` DATETIME NOT NULL,

`UPDATED\_AT` DATETIME NOT NULL, PRIMARY KEY (`ID`)

) ENGINE=INNODB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4\_unicode\_ci;

由此，创建Order类表示相对容易。 该表示包括所需的访问方法，这些方法用于设置或从基础订单数据库表中获取数据或从中获取数据：

class Order

{

const STATUS\_CREATED = 10; const STATUS\_ACCEPTED = 20; const STATUS\_PAID = 30; const STATUS\_PROCESSED = 40;

private $id;

private $customerId; private $amount; private $status; private $createdAt; private $updatedAt;

public function construct(

$customerId,

$amount,

$status,

DateTimeInterface $createdAt, DateTimeInterface $updatedAt

) {

$this->customerId = $customerId;

$this->amount = $amount;

$this->status = $status;

$this->createdAt = $createdAt;

$this->updatedAt = $updatedAt;

}

public function setId($id)

{

$this->id = $id;

}

public function getId()

{

return $this->id;

}

public function setCustomerId($customerId)

{

$this->customerId = $customerId;

}

public function getCustomerId()

{

return $this->customerId;

}

public function setAmount($amount)

{

$this->amount = $amount;

}

public function getAmount()

{

return $this->amount;

}

public function setStatus($status)

{

$this->status = $status;

}

public function getStatus()

{

return $this->status;

}

public function setCreatedAt(DateTimeInterface $createdAt)

{

$this->createdAt = $createdAt;

}

public function getCreatedAt()

{

return $this->createdAt;

}

public function setUpdatedAt(DateTimeInterface $updatedAt)

{

$this->updatedAt = $updatedAt;

}

public function getUpdatedAt()

{

return $this->updatedAt;

}

}

此实现的示例用例可能是更新订单状态，如下所示:

// Fetch an order from the database

$anOrder = $orderRepository->find( 1 );

// Update order status

$anOrder->setStatus(Order::STATUS\_ACCEPTED);

// Update updatedAt field

$anOrder->setUpdatedAt(new DateTimeImmutable());

// Save the order to the database

$orderRepository->save($anOrder);

关于代码重用，这个代码有一个类似于初始用户认证解决方案的问题。 为了解决这个问题，这种做法的捍卫者建议使用服务层，从而使操作明确且可重用。 现在可以将此前的实现封装到单独的类中:

class ChangeOrderStatusService

{

private $orderRepository;

public function construct(OrderRepository $orderRepository)

{

$this->orderRepository = $orderRepository;

}

public function execute($anOrderId, $anOrderStatus)

{

// Fetch an order from the database

$anOrder = $this->orderRepository->find($anOrderId);

// Update order status

$anOrder->setStatus($anOrderStatus);

// Update updatedAt field

$anOrder->setUpdatedAt(new DateTimeImmutable());

// Save the order to the database

$this->orderRepository->save($anOrder);

}

}

或者，在更新订单金额的情况下，考虑这一点:

class UpdateOrderAmountService

{

private $orderRepository;

public function construct(OrderRepository $orderRepository)

{

$this->orderRepository = $orderRepository;

}

public function execute( $orderId, $amount)

{

$anOrder = $this->orderRepository->find(1);

$anOrder->setAmount($amount);

$anOrder->setUpdatedAt(new DateTimeImmutable());

$this->orderRepository->save($anOrder);

}

}

在明确的意图操作之后，客户代码将大幅减少:

$updateOrderAmountService = new UpdateOrderAmountService(

$orderRepository

);

$updateOrderAmountService->execute(1, 20.5);

实施这种方法可以产生很大程度的代码重用性。 希望更新订单金额的人只需要检索UpdateOrderAmountService实例并使用适当的参数调用execute方法.

然而，选择这个路径打破了所讨论的面向对象设计原则，并且产生了构建域模型而不利用任何好处的成本.

### 贫血领域模型打破封装

如果我们重新审视用于在我们的服务层中定义服务的代码，我们可以看到，作为使用订单实体的客户，我们需要知道其内部表示的每个细节。 这一发现违背了面向对象编程的基本规则，即将数据与后续行为相结合

### 贫血域模型带来了错误的代码重用意识

假设有一个客户绕过UpdateOrderAmountService的实例，而是取回，更新并直接持续到OrderRepository。 然后，UpdateOrderAmountService服务可能拥有的所有额外业务逻辑将不会执行。 这可能导致订单以不一致的状态存储。 因此，不变量应该被正确保护，最好的办法是让真正的域模型来处理它。 在这个例子中，订单实体将是确保这一点的最佳地点:

class Order

{

// ...

public function changeAmount($amount)

{

$this->amount = $amount;

$this->setUpdatedAt(new DateTimeImmutable());

}

}

请注意，通过将此操作向下推入实体并以无处不在的语言命名，系统可以实现极佳的代码重用。 任何想要更改订单数量的人都必须直接调用Order :: changeAmount方法.

这导致更丰富的类，其中行为是代码重用的目标。 这通常被称为丰富域模型.

**如何避免贫血域模型**

避免陷入贫血域模型的方法是，当开始一个新的项目或功能时，首先考虑行为。 数据库，ORM等仅仅是实现细节，我们应尽可能在开发过程的最后阶段努力推动使用这些工具的决策。 在这样做的时候，我们可以关注一个重要的真实属性：行为.

就像实体一样，域服务也可以触发第6章域名事件。 但是，当事件主要由域服务而非实体触发时，这又是您可能正在创建贫血域模型的一个指标。

## 包起来

正如我们所看到的，服务代表我们系统内的操作，我们可以区分它们的三个版本:

**应用程序服务：帮助协调来自外部世界的请求到域中。 这些服务不应包含域逻辑。 事务在应用程序级别进行处理; 在跨国装饰器中包装你的服务将使你的代码事务不可知**.

**域服务：仅使用域概念进行操作，这是由无处不在的语言表达的。 请记住推迟实施细节并首先考虑行为，因为滥用域服务将导致无效域模型和不良面向对象设计**.

**基础架构服务：在基础架构上运行，执行诸如发送电子邮件或登录信息等**.

我们最重要的建议是，在决定创建域服务之前，您应该考虑所有选项。 首先尝试将您的业务逻辑移入实体或值中。 与一些同事核对。 再次查看。 如果使用不同的方法，最好的选择是创建一个域服务，那就去做吧.

# Domain-Events

****

软件事件是系统中发生的事情，其他组件可能有兴趣知道。 通常PHP开发人员不习惯使用事件，这不是该语言的一项功能。 然而，更常见的是看到新的框架和库如何接受它们以提供解耦，重用和加速代码的新方法。

域事件是与域更改相关的事件。 域名事件是域名专家关心的域名.

在领域驱动设计中，域事件是帮助的基础构建块:

与其他有界上下文沟通.

提高性能和可伸缩性，推动最终的一致性.

作为历史检查点.

域事件代表了异步通信的本质。 有关此主题的更多信息，我们推荐您使用Gregor Hohpe和Bobby Woolf撰写的企业集成模式：设计，构建和部署消息传递解决方案.

## 介绍

想想一个JavaScript 2D平台游戏。 屏幕上有很多不同的组件可以在同一时间在屏幕上互动。 有一个组件表示剩余的生命数，另一个表示所有得分点，另一个表示剩余时间以完成当前水平。 每次你的角色在敌人身上跳跃时，点数都会增加。 当你的得分高于一定数量时，你会获得额外的生命。 当你的角色拿起钥匙时，通常会打开一扇门。 但是，所有这些组件如何相互影响？ 这种情况下的最佳体系结构是什么？

可能有两个主要选项：第一个选项是将每个组件与其连接的组件相连接。 但是，在上面的例子中，这意味着很多组件都被耦合在一起，每个新的添加都要求开发者修改代码。 但是你还记得开放封闭原则（OCP）吗？ 添加一个新组件不应该使它成为第一个组件必须更新; 这将是太多的工作来维护.

第二种更好的方法是将所有组件连接到一个处理游戏中所有重要事件的单个对象。 它从每个组件接收事件并将它们转发给特定组件。 例如，评分组件会对EnemyKilled Event感兴趣，而LifeCaptured Event对玩家实体和剩余生命组件非常有用。 通过这种方式，所有组件都可以连接到管理所有通知的单个组件。 采用这种方法，添加或删除组件不会影响现有组件.

在开发单个应用程序时，事件可用于解耦组件。 当以分布式方式开发整个域时，事件对于解耦每个在域中扮演角色的服务或应用程序非常有用。 关键点是相同的，但规模不同.

## 定义

域事件是用于通知域更改的本地或远程有界上下文的一种特定类型的事件.

Vaughn Vernon将域名事件定义为:

发生在域中的事件.

Eric Evans将域名事件定义为:

领域模型的一个完整部分，表示在域中发生的事情。 忽略不相关的域活动，同时明确域专家想要跟踪或通知的事件，或者与其他模型对象中的状态更改关联的事件.

马丁福勒定义了一个域名事件:

捕获影响域的一些有趣内容.

Web应用程序中的域事件示例为UserRegistered，OrderPlaced，UserRelocated和ProductAdded.

### 短篇故事

在售票代理商中，内容管理员决定增加U2演出的价格。 使用她的后台，她编辑了节目。 ShowPriceChanged域事件将在同一事务中以新的展示价格发布并保存到数据库中.

批处理采用域事件并将其排入RabbitMQ。 域事件分布在两个队列中：一个用于同一本地有界上下文，另一个用于商业智能目的。

在第一个队列中，工作人员使用Event中的ID获取相应的Show并将其推送到Elasticsearch服务器，以便用户在搜索时可以看到新的价格。 它也可以更新不同数据库表中的新价格.

在第二个队列中，工作人员将信息插入到日志服务器或数据湖中，在该日志服务器或数据湖中可以运行报告或数据挖掘过程.

无法使用域事件集成的外部应用程序可以访问所有

ShowPriceChanged使用本地有界上下文提供的REST API的事件。

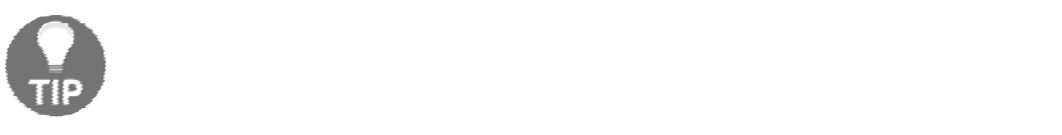
正如您所看到的，域事件对于处理最终一致性和集成不同的有界上下文非常有用。 聚合创建事件并发布它们。 用户可以存储事件，然后将它们转发给远程用户.

### 隐喻

我们周二去巴布尔吃顿饭，并用信用卡付款。 这可能会被模拟为事件类型为PurchasePlaced，我的信用卡主题和周二发生日期的事件。 如果Babur's使用旧的手动系统，直到星期五才传输交易，交易将在星期五生效.

事情发生。 并非所有这些都很有趣，有些可能值得记录，但不会引起反应。 然而，最有趣的事情会引起反应。 许多系统需要对有趣的事件做出反应。 通常你需要知道系统为什么会这样做.

通过将系统输入汇集到域事件流中，您可以记录系统的所有输入。 这可以帮助您组织处理逻辑，并且还可以让您保留对系统输入的审核日志.



**Exercise**

尝试查找当前域中潜在域名事件的示例.

### 真实的例子

在详细介绍域名事件之前，我们先看一个域名事件的真实例子，以及他们如何在我们的应用程序和我们的整个域中帮助我们.

让我们考虑一个简单的应用服务，它将注册一个新用户 - 例如，在电子商务环境中。 应用程序服务将在另一章中解释，所以不要太担心接口。 相反，只关注执行方法:

class SignUpUserService implements ApplicationService

{

private $userRepository; private $userFactory; private $userTransformer;

public function construct( UserRepository $userRepository, UserFactory $userFactory, UserTransformer $userTransformer

) {

$this->userRepository = $userRepository;

$this->userFactory = $userFactory;

$this->userTransformer = $userTransformer;

}

/\*\*

* @param SignUpUserRequest $request
* @return User
* @throws UserAlreadyExistsException

\*/

public function execute(SignUpUserRequest $request)

{

$email = $request->email();

$password = $request->password();

$user = $this->userRepository->userOfEmail($email ); if ($user) {

throw new UserAlreadyExistsException();

}

$user = $this->userFactory->build(

$this->userRepository->nextIdentity(),

$email,

$password

);

$this->userRepository->add($user);

$this->userTransformer->write($user);

}

}

如图所示，应用程序服务部分检查用户是否已经存在。 如果没有，它会创建一个新的用户并将其添加到UserRepository中.

现在考虑一个额外的要求：注册时必须通过电子邮件通知新用户。 在不考虑它的情况下，想到的第一种方法是更新我们的应用服务，以包含一段可以完成这项工作的代码 - 可能是在添加方法之后运行的某种EmailSender。 但是，让我们考虑另一种方法.

如何解决UserRegistered事件，以便另一个组件侦听可以作出反应并发送该电子邮件？ 这种新方法有一些很酷的好处。 首先，每当新用户注册时必须执行新的操作，我们不需要更新我们的应用服务的代码.

其次，它更容易测试。 应用程序服务保持简单，每次开发新的操作时，我们都会为操作编写测试。

后来在同一个电子商务项目中，我们被告知要集成一个非PHP编写的开源游戏化平台。 用户每次在我们的电子商务有界上下文中购买或查看产品时，他们都可以获取可以在其电子商务用户个人资料页面上显示的徽章或通过电子邮件通知。 我们如何模拟问题?

遵循第一种方法，我们将更新应用服务，以与前面的电子邮件确认方法类似的方式与新平台集成。 通过域事件方法，我们可以为UserRegistered事件创建另一个侦听器，它将通过REST或SOA直接连接到游戏化平台。 更好的是，它可以将事件传递给像RabbitMQ这样的消息传递系统，以便游戏化有界上下文可以订阅并自动获得通知。 我们的电子商务有限上下文根本不需要了解游戏化有界上下文.

## 特点

域名事件通常是不可变的，因为它们是过去某种事物的记录。 除了对事件的描述之外，域事件通常包含事件发生时间的时间戳以及事件中涉及的实体的身份。 另外，域事件通常有一个单独的时间戳，指示事件何时进入系统，以及输入该事件的人的身份。 如果有用，域事件的标识可以基于这些属性的一些集合。 因此，例如，如果同一事件的两个实例到达节点，则它们可以被识别为相同.

域事件的本质是您使用它来捕获可能触发您正在开发的应用程序状态或您的域中对这些更改感兴趣的其他应用程序状态的更改的事物。 然后处理这些事件对象以更改系统并存储以提供审计日志.

### 命名约定

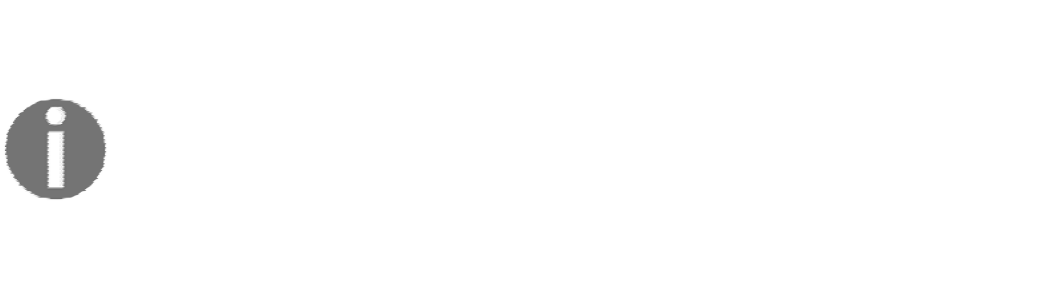
所有事件都应该用过去式来表示，因为它们是过去完成的事情 - 例如，CustomerRelocated，CargoShipped或InventoryLossageRecorded。 在英语中有一些有趣的例子，可能会试图用名词而不是过去式的动词; 这方面的一个例子就是将地震或Capsize作为对自然灾害感兴趣的国会议员的相关活动。 我们建议避免使用域名事件等名称的诱惑，并在过去时刻坚持使用动词.

### 域名事件和无处不在的语言

当我们讨论迁移客户的副作用时，请考虑泛在语言的差异。 事件使得这个概念是明确的，而之前，发生在一个聚集内或多个聚合之间的变化是作为一个需要探索和定义的隐含概念。 例如，在大多数系统中，当像Hibernate或实体框架这样的库发生副作用时，它不会影响域。 这些事件从客户的角度来看是隐含的和透明的。 事件的引入使得这个概念变得明确，并且是无处不在的语言的一部分。 搬迁客户不仅仅是改变一些东西， 而是产生一个在语言中明确定义的CustomerRelocatedEvent.

### 不变性

正如我们已经提到的，域名事件会讨论过去，并描述您域中已经发生的变化。 根据定义，改变过去是不可能的，除非你是Marty McFly并且拥有DeLorean，可能并非如此。 所以请记住，域名事件是不可改变的.



**Symfony事件调度程序**

一些PHP框架支持事件。 但是，不要将这些事件与域事件混淆; 他们在特征和目标上有所不同。 例如，Symfony具有Event Dispatcher组件，如果需要为状态机实现Event系统，则可以依赖它。 在Symfony中，从请求到响应的转换也由事件处理。 但是，Symfony事件是可变的，每个侦听器都能够修改，添加或更新事件中的信息.

## 建模事件

为了准确地描述您的业务领域，您必须与领域专家密切合作并定义无处不在的语言。 这需要使用域事件，实体，值对象等来构建域概念。 在为事件建模时，根据无处不在的语言在它们起源的有界上下文中命名它们及其属性。 如果事件是对Aggregate执行命令操作的结果，则该名称通常来自执行的命令。 事件名称反映事件的过去性质很重要.

让我们考虑我们的用户注册功能; 域事件需要表示它。 以下代码显示了基本域事件的最小界面：

interface DomainEvent

{

/\*\*

* @return DateTimeImmutable

\*/

public function occurredOn();

}

正如您所看到的，所需的最少信息是DateTimeImmutable，这是知道Event何时发生的必要信息。

现在让我们使用下面的代码模拟新的用户注册事件。 正如我们上面提到的，名字应该是过去式的动词，所以UserRegistered可能是一个不错的选择:

class UserRegistered implements DomainEvent

{

private $userId;

public function construct(UserId $userId)

{

$this->userId = $userId;

$this->occurredOn = new \DateTimeImmutable();

}

public function userId()

{

return $this->userId;

}

public function occurredOn()

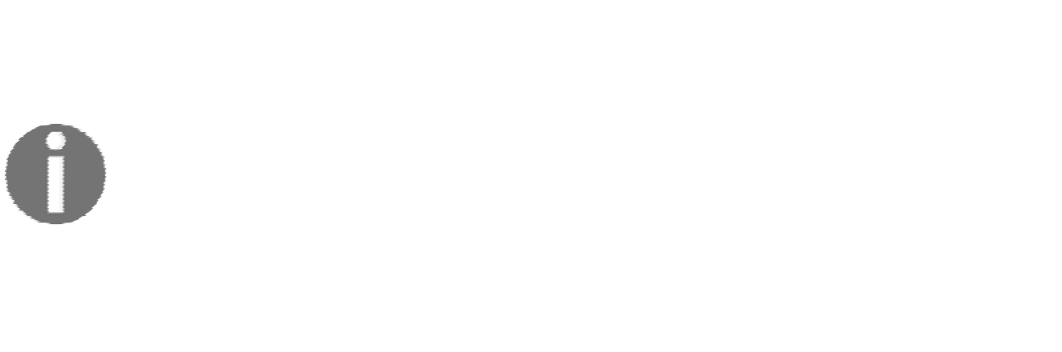
{

return $this->occurredOn;

}

}

用户通知用户创建新用户所需的最少信息量是UserId。 有了这些信息，任何进程，命令或应用程序服务（来自相同的有界上下文或不同的进程）都可能对此事件做出反应.



**As a Rule of Thumb**

域事件通常被设计为不可变的

构造函数将初始化域事件的完整状态.

域事件将让获取者访问其属性包括执行操作的聚合的标识包括与第一个聚合相关的其他聚合标识包含导致该事件的参数（如果有用的话）

但是如果来自相同有界上下文或不同域的专家需要更多信息会发生什么？ 让我们看看以更多信息为模型的相同域事件 - 例如，电子邮件地址:

class UserRegistered implements DomainEvent

{

private $userId; private $userEmail ;

public function construct(UserId $userId, $userEmail)

{

$this-> userId = $userId;

$this->userEmail = $userEmail ;

$this->occurredOn = new DateTimeImmutable();

}

public function userId()

{

return $this->userId;

}

public function userEmail ()

{

return $this->userEmail ;

}

public function occurredOn()

{

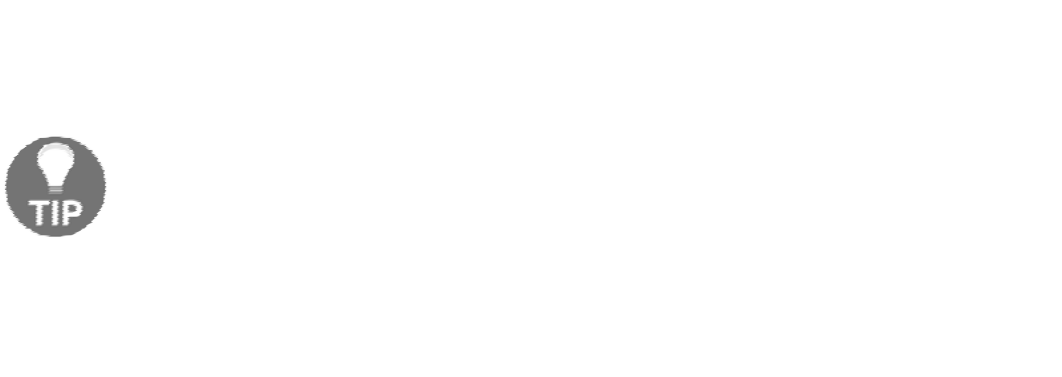
return $this->occurredOn;

}

}

上面，我们添加了电子邮件地址。 将更多信息添加到域事件可以帮助提高性能或简化不同有界上下文之间的集成。 从另一个有界上下文的角度思考可能有助于为事件建模。 当在上游有界上下文中创建新用户时，下游用户必须创建自己的用户。 在下游需要的情况下，添加用户电子邮件可能会将同步请求保存到上游受限上下文。

你还记得游戏化的例子吗？ 为了创建游戏化平台的用户，可能称为Player，来自电子商务有界上下文的UserId可能就足够了。 但是，如果游戏平台必须通过电子邮件通知用户关于奖励的情况会发生什么？ 在这种情况下，电子邮件地址也是强制性的。 所以如果电子邮件地址包含在原始域名事件中，我们就完成了。 如果情况并非如此，则游戏化有界上下文需要通过REST或SOA集成请求来自电子商务有界上下文的信息.



**为什么不是整个用户实体？?**

想知道是否应该在域事件中包含来自有界上下文中的整个用户实体？ 我们的建议是，你没有。 域事件可能用于在内部将消息传递给给定的有界上下文或从外部传递给其他有界上下文。 换句话说，C2C电子商务产品目录中的卖主可以是产品评论的作者，也可以是产品评论的作者。 两者都可以共享相同的ID或电子邮件，但Seller和Author是代表来自不同有界上下文的不同实体的不同概念。 所以来自一个有界上下文的实体在另一个有界上下文中没有意义或完全不同.

## Doctrine Events

域事件不仅仅用于执行批量作业，如发送电子邮件或与其他有界上下文进行通信; 他们对性能和可伸缩性的改进也很有意思。 我们来看一个例子.

考虑以下情况。 你有一个电子商务应用程序。 您的主要持久性机制是MySQL，但为了浏览和过滤目录，您正在使用更好的方法，例如Elasticsearch或Solr。 在Elasticsearch上，您将获得存储在完整数据库中的信息的子集。 你如何保持数据同步？ 内容团队从后台工具更新目录时会发生什么情况?

有人不时重新编制整个目录。 这非常昂贵和缓慢。 更聪明的方法可能是更新与已更新的产品相关的一个或一些文档。 我们怎么做到这一点？ 使用域名事件.

但是，如果您一直在使用Doctrine，那么这对您来说可能并不新鲜。 根据Doctrine 2 ORM 2文档:

Doctrine 2具有一个轻量级事件系统，它是Common包的一部分。 学说使用它来分派系统事件，主要是生命周期事件。 您也可以将其用于您自己的自定义事件.

此外，它指出:

生命周期回调在实体类上定义。 它们允许您在该实体类的实例遇到相关生命周期事件时触发回调。 可以为每个生命周期事件定义多个回调。 生命周期回调最适合特定实体类生命周期的简单操作.

我们来看看Doctrine Events文档中的一个例子:

/\*\* @Entity @HasLifecycleCallbacks \*/ class User

{

// ...

/\*\*

\* @Column(type="string", length=255)

\*/

public $value;

/\*\* @Column(name="created\_at", type="string", length=255) \*/ private $createdAt;

/\*\* @PrePersist \*/

public function doStuffOnPrePersist()

{

$this->createdAt = date('Y-m-d H:i:s');

}

/\*\* @PrePersist \*/

public function doOtherStuffOnPrePersist()

{

$this-> value = 'changed from prePersist callback!';

}

/\*\* @PostPersist \*/

public function doStuffOnPostPersist()

{

$this->value = 'changed from postPersist callback!';

}

/\*\* @PostLoad \*/

public function doStuffOnPostLoad()

{

$this->value = 'changed from postLoad callback!';

}

/\*\* @PreUpdate \*/

public function doStuffOnPreUpdate()

{

$this->value = 'changed from preUpdate callback!';

}

}

您可以在Doctrine Entity生命周期的每个不同重要时刻挂钩特定任务。 例如，在PostPersist上，您可以生成实体的JSON文档并将其放入Elasticsearch中。 这样，在不同的持久性机制之间保持数据一致性很容易。

学习活动是活动围绕您的实体获益的一个很好的例子。 但是你可能想知道他们的问题是什么。 这是因为它们耦合到一个框架，它们是同步的，它们在应用程序级别上运行，但不是用于通信目的。 所以这就是为什么域名事件，尽管实施和处理更难以更有趣。

## 坚持域名事件

坚持事件永远是一个好主意。 你们中的一些人可能会想知道为什么你不应该直接将域事件发布到消息传递或日志记录系统。 这是因为坚持它们具有有趣的好处:

您可以通过REST界面将您的域名事件公开给其他有界上下文.

在将它们推送到RabbitMQ之前，您可以在相同的数据库事务中持久化域事件和聚合更改。 （你不想发送关于没有发生的事情的通知，就像你不想错过关于发生的事情的通知一样。）

商业智能可以使用这些数据进行分析，预测或趋势。

您可以审核您的实体更改.

对于事件采购，您可以重构来自域事件的聚合.

### Event Store

我们在哪里坚持域名事件？ 在活动商店。 事件存储是一个域事件存储库，作为抽象（接口或抽象类）存在于我们的域空间中。 其职责是追加域事件并查询它们。 一个可能的基本接口可能如下:

interface EventStore

{

public function append(DomainEvent $aDomainEvent); public function allStoredEventsSince($anEventId);

}

不过，根据您的域名事件的使用情况，以前的界面可以有更多的方法来查询您的活动.

在实现方面，你可以决定使用一个Doctrine Repository，一个DBAL或一个普通的PDO。 因为域事件是不可变的，所以使用Doctrine Repository会增加不必要的性能损失，尽管对于中小型应用程序来说，Doctrine可能是好的。 让我们看看Doctrine的一个可能的实现:

class DoctrineEventStore extends EntityRepository implements EventStore

{

private $serializer;

public function append(DomainEvent $aDomainEvent)

{

$storedEvent = new StoredEvent( get\_class($aDomainEvent),

$aDomainEvent->occurredOn(),

$this->serializer()->serialize($aDomainEvent, 'json')

);

$this->getEntityManager()->persist($storedEvent);

}

public function allStoredEventsSince($anEventId)

{

$query = $this->createQueryBuilder('e'); if ($anEventId) {

$query->where('e.eventId > :eventId');

$query->setParameters(['eventId' => $anEventId]);

}

$query->orderBy('e.eventId');

return $query->getQuery()->getResult();

}

private function serializer()

{

if (null === $this->serializer) {

/\*\* \JMS\Serializer\Serializer\SerializerBuilder \*/

$this->serializer = SerializerBuilder::create()->build();

}

return $this->serializer;

}

}

StoredEvent是映射到数据库所需的Doctrine实体。 正如您可能已经看到的那样，在持续存储时追加和保存之后，没有同步通话。 如果此操作位于Doctrine事务中，则不需要。 所以，让我们在没有电话的情况下离开它，并且在谈论应用程序服务时我们会详细讨论。

现在让我们看看StoredEvent实现:

class StoredEvent implements DomainEvent

{

private $eventId; private $eventBody; private $occurredOn; private $typeName;

/\*\*

* @param string $aTypeName
* @param \DateTimeImmutable $anOccurredOn
* @param string $anEventBody

\*/

public function construct(

$aTypeName, \DateTimeImmutable $anOccurredOn, $anEventBody

) {

$this->eventBody = $anEventBody;

$this->typeName = $aTypeName;

$this->occurredOn = $anOccurredOn;

}

public function eventBody()

{

return $this->eventBody;

}

public function eventId()

{

return $this->eventId;

}

public function typeName()

{

return $this->typeName;

}

public function occurredOn()

{

return $this->occurredOn;

}

}

And here is its mapping:

Ddd\Domain\Event\StoredEvent: type: entity

table: event repositoryClass:

Ddd\Infrastructure\Application\Notification\DoctrineEventStore

id:

eventId:

type: integer column: event\_id generator: strategy: AUTO

fields:

eventBody:

column: event\_body type: text

typeName:

column: type\_name type: string length: 255

occurredOn:

column: occurred\_on type: datetime

为了将Domain Events保留在不同的字段中，我们必须将这些字段作为序列化字符串加入。 typeName标识域范围域事件。 实体或值对象在有界上下文中有意义，但域事件定义了有界上下文之间的通信协议.

在分布式系统中，狗屎会发生。 您必须处理未发布的域事件，丢失链中的某个位置或发布多次。 这就是为什么持续使用ID标识域名事件很重要，以便跟踪哪些域名事件已发布以及哪些域名事件缺失.

## 从域模型发布事件

域名事件应该在它们代表的事实发生时发布。 例如，当新用户注册时，应该发布新的用户注册事件.

报纸隐喻之后：

**建模域事件就像写一篇新闻文章**

**发布域事件就像在论文中打印文章一样**

**传播域名事件就像传递报纸一样，每个人都可以阅读文章**

发布域事件的推荐方法是使用简单的侦听器观察者模式来实现DomainEventPublisher.

### 从实体发布域事件

继续以我们的应用程序注册的新用户为例，我们来看看如何发布相应的域名事件:

class User

{

protected $userId; protected $email ; protected $password;

public function construct(UserId $userId, $email, $password)

{

$this->setUserId($userId);

$this->setEmail($email);

$this->setPassword($password);

DomainEventPublisher::instance()->publish( new UserRegistered($this->userId)

);

}

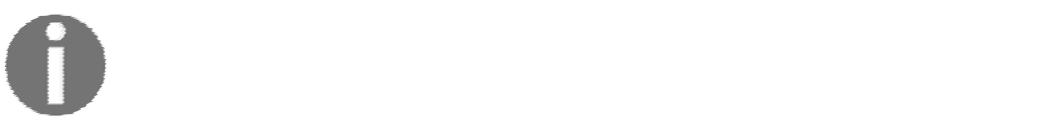
// ...

}

如示例所示，创建用户时，会发布新的用户注册事件。 它在Entity构造函数中完成，而不是在外部，因为使用这种方法，保持域的一致性更容易; 任何创建新用户的客户都将发布其相应的事件。 另一方面，这使得使用需要在不使用其构造函数的情况下创建用户实体的基础结构变得更加复杂。 例如，Doctrine使用serialize和unserialize技术来重新创建一个对象而不调用它的构造函数。 但是，如果你必须自己创造，这不会像在教义中那么容易。

通常，从诸如数组等普通数据构造对象称为水合。 让我们看一个简单的方法来构建从数据库中获取的新用户。 首先，我们通过应用Factory Method模式将Domain Event发布提取到它自己的方法。

根据维基百科:



模板方法模式是一种行为设计模式，它定义了操作中算法的程序框架，将一些步骤推迟到子类中:

class User

{

protected $userId; protected $email; protected $password;

public function construct(UserId $userId, $email, $password)

{

$this->setUserId($userId);

$this->setEmail($email);

$this->setPassword($password);

$this->publishEvent();

}

protected function publishEvent()

{

DomainEventPublisher::instance()->publish( new UserRegistered($this->userId)

);

}

// ...

}

现在，让我们用一个新的基础设施实体扩展我们的当前用户，这个实体将为我们完成这项工作。 这里的诀窍是使publishEvent不做任何事情，以免发布域事件:

class CustomOrmUser extends User

{

protected function publishEvent()

{

}

public static function fromRawData($data)

{

return new self(

new UserId($data['user\_id']),

$data['email'],

$data['password']

);

}

}

记住要小心这种方法; 您可能会从持久性机制中获取无效对象，因为域规则始终在更改。 不使用父构造函数的另一种方法可能如下:

class CustomOrmUser extends User

{

public function construct()

{

}

public static function fromRawData($data)

{

$user = new self();

$user->userId = new UserId($data['user\_id']);

$user->email = $data['email'];

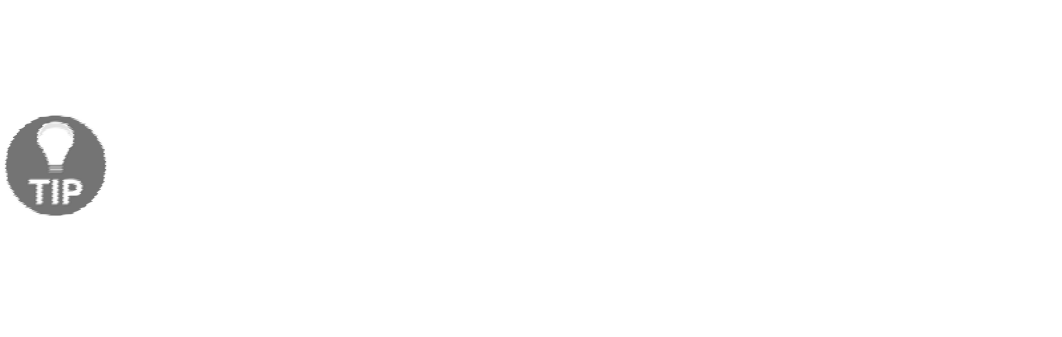
$user->password = $data['password'];

return $user;

}

}

通过这种方法，父构造函数不会被调用，并且用户属性必须受到保护。 其他的选择是反射，在构造函数中传递标志，使用Proxy-Manager之类的代理库，或者使用像Doctrine这样的ORM.



**Other Strategy for Publishing Domain Events**

As you can see in the previous example, we're using a static class for publishing our Domain Events. Other people, as an alternative, and especially when using [Event Sourcing](http://martinfowler.com/eaaDev/EventSourcing.html), will suggest that Entities hold all the fired Events internally within a field. In order to access all the Events, a getter is used in the Aggregate. This is also a valid approach. However, sometimes it's a bit difficult to keep track of which Entities have fired an Event. It can also be difficult to fire Events from places that aren't just Entities, example: Domain Services. On the plus side, testing if an Entity has fired an Event is much easier.

### 从域或应用程序服务发布您的域名事件

您应该努力从链中的更深层发布域名事件。 实体或价值对象越接近内部越好。 正如我们在前一节中看到的，有时这并不容易，但最终结果对于客户来说更简单。 我们已经看到开发者从应用程序服务或域服务发布域事件。 这看起来更容易做到，但最终会导致贫血域模型。 这与在域服务中推送业务逻辑而不是将其放入实体中时不同.

### 域名事件发布者如何工作

您应该努力从链中的更深层发布域名事件。 实体或价值对象越接近内部越好。 正如我们在前一节中看到的，有时这并不容易，但最终结果对于客户来说更简单。 我们已经看到开发者从应用程序服务或域服务发布域事件。 这看起来更容易做到，但最终会导致贫血域模型。 这与在域服务中推送业务逻辑而不是将其放入实体中时不同...:

class DomainEventPublisher

{

private $subscribers;

private static $instance = null;

public static function instance()

{

if (null === static::$instance) { static::$instance = new static();

}

return static::$instance;

}

private function construct()

{

$this->subscribers = [];

}

public function clone()

{

throw new BadMethodCallException('Clone is not supported');

}

public function subscribe(

DomainEventSubscriber $aDomainEventSubscriber

) {

$this->subscribers[] = $aDomainEventSubscriber;

}

public function publish(DomainEvent $anEvent)

{

foreach ($this->subscribers as $aSubscriber) { if ($aSubscriber->isSubscribedTo($anEvent)) {

$aSubscriber->handle($anEvent);

}

}

}

}

发布方法遍历所有可能的订阅者，检查他们是否对发布的域事件感兴趣。 如果是这种情况，则调用订户的句柄方法.

订阅方法添加了一个新的DomainEventSubscriber，它将监听特定的域事件类型：

interface DomainEventSubscriber

{

/\*\*

* @param DomainEvent $aDomainEvent

\*/

public function handle($aDomainEvent);

/\*\*

* @param DomainEvent $aDomainEvent
* @return bool

\*/

public function isSubscribedTo($aDomainEvent);

}

正如我们已经讨论过的，坚持所有的域名事件是一个好主意。 通过使用特定的用户，我们可以轻松地坚持在我们的应用中发布的所有域名事件。 让我们创建一个DomainEventSubscriber，它将监听所有的域事件，无论是哪种类型，并使用我们的EventStore持久化它们:

class PersistDomainEventSubscriber implements DomainEventSubscriber

{

private $eventStore;

public function construct(EventStore $anEventStore)

{

$this->eventStore = $anEventStore;

}

public function handle($aDomainEvent)

{

$this->eventStore->append($aDomainEvent);

}

public function isSubscribedTo($aDomainEvent)

{

return true;

}

}

$ eventStore可以是一个自定义的Doctrine Repository，正如已经看到的，或者任何其他能够将DomainEvents保存到数据库中的对象.

### 设置DomainEventListeners

哪里是建立DomainEventPublisher订户的最佳地点？ 这取决于。 对于可能影响整个请求周期的全局用户，最好的地方可能是DomainEventPublisher初始化本身。 对于受特定应用服务影响的用户，服务实例化可能是一个更好的地方。 我们来看一个使用Silex的例子.

在Silex中，注册域名事件发布者的最佳方式是保留所有域名事件，即使用应用程序中间件。 根据Silex 2.0文档:

应用程序中间件之前允许您在执行控制器之前调整请求.

这是订阅负责持久化数据库的监听器的正确位置那些事件将在稍后发送给RabbitMQ:

// ...

$app['em'] = $app-> share(function () {

return (new EntityManagerFactory())->build();

});

$app['event\_repository'] = $app->share(function ($app) { return $app['em']->getRepository(

'Ddd\Domain\Model\Event\StoredEvent'

);

});

$app['event\_publisher'] = $app->share(function($app) { return DomainEventPublisher::instance();

});

$app->before(

function(Symfony\Component\HttpFoundation\Request $request) use($app) {

$app['event\_publisher']->subscribe( new PersistDomainEventSubscriber(

$app['event\_repository']

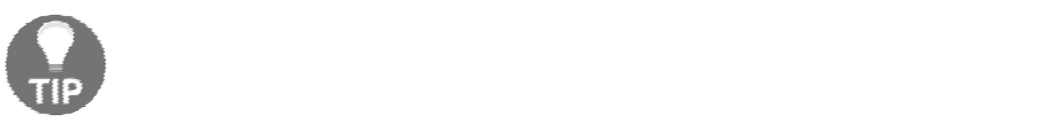
)

);

}

);

通过这种设置，每当Aggregate发布域事件时，它都会持久保存到数据库中。 任务完成.



**Exercise**

如果您正在使用Symfony，Laravel或其他PHP框架，请找到一种方法来订阅全球特定的订户，以执行围绕您的域名事件的任务.

如果您希望在请求即将完成时对所有域事件执行任何操作，则可以创建一个将所有已发布的域事件存储在内存中的侦听器。 如果您向该侦听器添加getter以返回所有域事件，则可以决定要执行什么操作。 如果您不想或如果您无法像前面所建议的那样在同一个事务中坚持事件，这可能很有用

### Testing Domain Events

You already know how to publish Domain Events, but how can you unit test this and ensure that UserRegistered is really fired? The easiest way we suggest is to use a specific EventListener that will work as a [Spy](http://www.martinfowler.com/bliki/TestDouble.html) to record whether or not the Domain Event was published. Let's see an example of the User Entity unit test:

use Ddd\Domain\DomainEventPublisher; use Ddd\Domain\DomainEventSubscriber;

class UserTest extends \PHPUnit\_Framework\_TestCase

{

// ...

/\*\*

* @test

\*/

public function itShouldPublishUserRegisteredEvent()

{

$subscriber = new SpySubscriber();

$id = DomainEventPublisher::instance()->subscribe($subscriber);

$userId = new UserId();

new User($userId, 'valid@email.com', 'password'); DomainEventPublisher::instance()->unsubscribe($id);

$this->assertUserRegisteredEventPublished($subscriber,$userId);

}

private function assertUserRegisteredEventPublished(

$subscriber, $userId

) {

$this->assertInstanceOf(

'UserRegistered', $subscriber->domainEvent

);

$this->assertTrue(

$subscriber->domainEvent->serId()->equals($userId)

);

}

}

class SpySubscriber implements DomainEventSubscriber

{

public $domainEvent;

public function handle($aDomainEvent)

{

$this->domainEvent = $aDomainEvent;

}

public function isSubscribedTo($aDomainEvent)

{

return true;

}

}

有一些替代上述。 您可以使用DomainEventPublisher的静态设置器或某个反射框架来检测该调用。 但是，我们认为我们分享的方法更自然。 最后但并非最不重要的一点是，请记住清理Spy订阅，以免影响其余单元测试的执行.

## 将消息传播到远程有界上下文

为了将一组域事件传递给本地或远程有界上下文，有两种主要策略：消息传递和REST API。 第一个计划使用诸如RabbitMQ之类的消息系统来传输域事件。 第二个计划是创建一个REST API来访问特定有界上下文的域事件.

### Messaging

随着所有的域名事件被保存到数据库中，传播新闻的唯一信息就是将他们推送到我们最喜欢的邮件系统。 我们更喜欢RabbitMQ，但任何其他系统（如ActiveMQ或ZeroMQ）都可以完成这项工作。 为了与使用PHP的RabbitMQ集成，没有太多选项，但是php-amqplib将完成这项工作.

首先，我们需要一种能够将持久性域事件发送到RabbitMQ的服务。 您可能想要查询所有事件的EventStore并发送每个事件，这不是一个坏主意。 但是，我们可以多次推送相同的域名事件，并且一般而言，我们需要最小化重新发布的域名事件的数量。 如果重新发布的域名事件数量为0，那就更好了。 为了不重新发布域名事件，我们需要某种组件来跟踪哪些域名事件已被推送，哪些域名剩余。 最后但并非最不重要的是，一旦我们知道我们必须推送哪些域名事件，我们会发送它们并跟踪发布到我们的邮件系统中的最后一条。 让我们看看这个服务的一个可能的实现:

class NotificationService

{

private $serializer; private $eventStore;

private $publishedMessageTracker; private $messageProducer;

public function construct( EventStore $anEventStore,

PublishedMessageTracker $aPublishedMessageTracker, MessageProducer $aMessageProducer,

Serializer $aSerializer

) {

$this->eventStore = $anEventStore;

$this->publishedMessageTracker = $aPublishedMessageTracker;

$this->messageProducer = $aMessageProducer;

$this->serializer = $aSerializer;

}

/\*\*

* @return int

\*/

public function publishNotifications($exchangeName)

{

$publishedMessageTracker = $this->publishedMessageTracker();

$notifications = $this->listUnpublishedNotifications(

$publishedMessageTracker

->mostRecentPublishedMessageId($exchangeName)

);

if (!$notifications) { return 0;

}

$messageProducer = $this->messageProducer();

$messageProducer->open($exchangeName); try {

$publishedMessages = 0;

$lastPublishedNotification = null;

foreach ($notifications as $notification) {

$lastPublishedNotification = $this->publish(

$exchangeName,

$notification,

$messageProducer

);

$publishedMessages++;

}

} catch (\Exception $e) {

// Log your error (trigger\_error, Monolog, etc.)

}

$this->trackMostRecentPublishedMessage(

$publishedMessageTracker,

$exchangeName,

$lastPublishedNotification

);

$messageProducer->close($exchangeName);

return $publishedMessages;

}

protected function publishedMessageTracker()

{

return $this->publishedMessageTracker;

}

/\*\*

* @return StoredEvent[]

\*/

private function listUnpublishedNotifications(

$mostRecentPublishedMessageId

) {

return $this

->eventStore()

->allStoredEventsSince($mostRecentPublishedMessageId);

}

protected function eventStore()

{

return $this->eventStore;

}

private function messageProducer()

{

return $this->messageProducer;

}

private function publish(

$exchangeName,

StoredEvent $notification, MessageProducer $messageProducer

) {

$messageProducer->send(

$exchangeName,

$this->serializer()->serialize($notification, 'json'),

$notification->typeName(),

$notification->eventId(),

$notification->occurredOn()

);

return $notification;

}

private function serializer()

{

return $this->serializer;

}

private function trackMostRecentPublishedMessage( PublishedMessageTracker $publishedMessageTracker,

$exchangeName,

$notification

) {

$publishedMessageTracker->trackMostRecentPublishedMessage(

$exchangeName, $notification

);

}

}

NotificationService depends on three interfaces. We've already seen EventStore, which is responsible for appending and querying Domain Events. The second one is PublishedMessageTracker, which is responsible for keeping track of pushed messages. The third one is MessageProducer, an interface representing our messaging system:

interface PublishedMessageTracker

{

/\*\*

* @param string $exchangeName
* @return int

\*/

public function mostRecentPublishedMessageId($exchangeName);

/\*\*

* @param string $exchangeName
* @param StoredEvent $notification

\*/

public function trackMostRecentPublishedMessage(

$exchangeName, $notification

);

}

The mostRecentPublishedMessageId method returns the ID of last PublishedMessage, so that the process can start from the next one. trackMostRecentPublishedMessage is responsible for tracking which message was sent last, in order to be able to republish messages in case you need to. $exchangeName represents the communication channel we're going to use to send out our Domain Events. Let's see a Doctrine implementation of PublishedMessageTracker:

class DoctrinePublishedMessageTracker extends EntityRepository\ implements PublishedMessageTracker

{

/\*\*

* @param $exchangeName
* @return int

\*/

public function mostRecentPublishedMessageId($exchangeName)

{

$messageTracked = $this->findOneByExchangeName($exchangeName); if (!$messageTracked) {

return null ;

}

return $messageTracked->mostRecentPublishedMessageId();

}

/\*\*

\*@param $exchangeName

* @param StoredEvent $notification

\*/

public function trackMostRecentPublishedMessage(

$exchangeName, $notification

) {

if(!$notification) { return;

}

$maxId = $notification->eventId();

$publishedMessage= $this->findOneByExchangeName($exchangeName);

if(null === $publishedMessage){

$publishedMessage = new PublishedMessage(

$exchangeName,

$maxId

);

}

$publishedMessage->updateMostRecentPublishedMessageId($maxId);

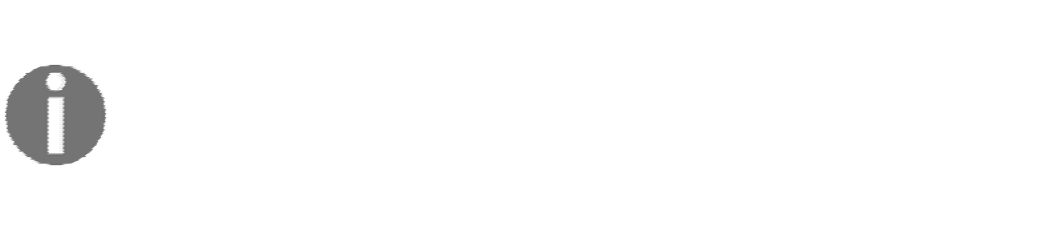
$this->getEntityManager()->persist($publishedMessage);

$this->getEntityManager()->flush($publishedMessage);

}

}

This code is quite straightforward. The only edge case we have to consider is when no Domain Event has already been published.



**Why an Exchange Name?**

We'll see this in more detail in the [Chpater 12](#_bookmark0), *Integrating Bounded Contexts*. However, when a system is running and a new Bounded Context comes into play, you might be interested in resending all the Domain Events to the new Bounded Context. So keeping track of the last Domain Event published and the channel where it was sent might come in handy later.

In order to keep track of published Domain Events, we need an exchange name and a notification ID. Here's a possible implementation:

class PublishedMessage

{

private $mostRecentPublishedMessageId; private $trackerId;

private $exchangeName;

/\*\*

* @param string $exchangeName
* @param int $aMostRecentPublishedMessageId

\*/

public function construct(

$exchangeName, $aMostRecentPublishedMessageId

) {

$this->mostRecentPublishedMessageId =

$aMostRecentPublishedMessageId;

$this->exchangeName = $exchangeName;

}

public function mostRecentPublishedMessageId()

{

return $this->mostRecentPublishedMessageId;

}

public function updateMostRecentPublishedMessageId($maxId)

{

$this->mostRecentPublishedMessageId = $maxId;

}

public function trackerId()

{

return $this->trackerId;

}

}

And here is its corresponding mapping:

Ddd\Domain\Event\PublishedMessage: type: entity

table: event\_published\_message\_tracker repositoryClass:

Ddd\Infrastructure\Application\Notification\ DoctrinePublished\MessageTracker

id:

trackerId:

column: tracker\_id type: integer generator: strategy: AUTO

fields:

mostRecentPublishedMessageId:

column: most\_recent\_published\_message\_id type: bigint

exchangeName:

type: string

column: exchange\_name

Now let's see what the MessageProducer interface is used for, along with its implementation details:

interface MessageProducer

{

public function open($exchangeName);

/\*\*

* @param $exchangeName
* @param string $notificationMessage
* @param string $notificationType
* @param int $notificationId
* @param \DateTimeImmutable $notificationOccurredOn
* @return

\*/

public function send(

$exchangeName,

$notificationMessage,

$notificationType,

$notificationId,

\DateTimeImmutable $notificationOccurredOn

);

public function close($exchangeName);

}

Easy. The open and close methods open and close a connection with the messaging system. send takes a message body — message name and message ID — and sends them to our messaging engine, whatever it is. Because we've chosen RabbitMQ, we need to implement the connection and sending process:

abstract class RabbitMqMessaging

{

protected $connection; protected $channel ;

public function construct(AMQPConnection $aConnection)

{

$this->connection =$aConnection;

$this->channel = null ;

}

private function connect($exchangeName)

{

if (null !== $this->channel ) { return;

}

$channel = $this->connection->channel();

$channel->exchange\_declare(

$exchangeName, 'fanout', false, true, false

);

$channel->queue\_declare(

$exchangeName, false, true, false, false

);

$channel->queue\_bind($exchangeName, $exchangeName);

$this->channel = $channel ;

}

public function open($exchangeName)

{

}

protected function channel ($exchangeName)

{

$this->connect($exchangeName);

return $this->channel;

}

public function close($exchangeName)

{

$this->channel->close();

$this->connection->close();

}

}

class RabbitMqMessageProducer extends RabbitMqMessaging implements MessageProducer

{

/\*\*

* @param $exchangeName
* @param string $notificationMessage
* @param string $notificationType
* @param int $notificationId
* @param \DateTimeImmutable $notificationOccurredOn

\*/

public function send(

$exchangeName,

$notificationMessage,

$notificationType,

$notificationId,

\DateTimeImmutable $notificationOccurredOn

) {

$this->channel ($exchangeName)->basic\_publish( new AMQPMessage(

$notificationMessage, [

'type'=>$notificationType, 'timestamp'=>$notificationOccurredOn->getTimestamp(), 'message\_id'=>$notificationId

]

),

$exchangeName

);

}

}

Now that we have a DomainService for pushing Domain Events into a messaging system like RabbitMQ, it's time to execute them. We need to choose a delivery mechanism to run the service. We personally suggest creating a [Symfony Console](http://symfony.com/doc/current/components/console/introduction.html) Command:

class PushNotificationsCommand extends Command

{

protected function configure()

{

$this

->setName('domain:events:spread')

->setDescription('Notify all domain events via messaging')

->addArgument( 'exchange-name',

InputArgument::OPTIONAL,

'Exchange name to publish events to', 'my-bc-app'

);

}

protected function execute(

InputInterface $input, OutputInterface $output

) {

$app = $this->getApplication()->getContainer();

$numberOfNotifications =

$app['notification\_service']

->publishNotifications(

$input->getArgument('exchange-name')

);

$output->writeln( sprintf(

'<comment>%d</comment>' . '<info>notification(s) sent!</info>',

$numberOfNotifications

)

);

}

}

Following the Silex example, let's see the definition of the

$app['notification\_service'] defined in the [Silex Pimple Service Container](http://silex.sensiolabs.org/doc/services.html#id1):

// ...

$app['event\_store']=$app->share( function ($app) {

return $app['em']->getRepository('Ddd\Domain\Event\StoredEvent');

});

$app['message\_tracker'] = $app->share(function($app) { return $app['em']

->getRepository('Ddd\Domain\Event\Published\Message');

});

$app['message\_producer'] = $app->share(function () { return new RabbitMqMessageProducer(

new AMQPStreamConnection('localhost', 5672, 'guest', 'guest')

);

});

$app['message\_serializer'] = $app->share(function () { return SerializerBuilder::create()->build();

});

$app['notification\_service'] = $app->share(function ($app) { return new NotificationService(

$app['event\_store'],

$app['message\_tracker'],

$app['message\_producer'],

$app['message\_serializer']

);

});

//...

### Syncing Domain Services with REST

With the EventStore already implemented in the messaging system, it should be easy to add some pagination capabilities, query for Domain Events, and render a JSON or XML representation publishing a REST API. Why is that interesting? Well, distributed systems using messaging have to face many different problems, such as messages that don't arrive, messages that arrive duplicated, or messages that arrive in an unexpected order. That's why it's nice to provide an API to publish your Domain Events so that other Bounded Contexts can ask for some missing information. Just as an example, consider that you make an HTTP request to an /events endpoint. A possible result would be the following:

[

{

"id": 1,

"version": 1,

"typeName": "Lw\\Domain\\Model\\User\\UserRegistered", "eventBody": {

"user\_id": {

"id": "459a4ffc-cd57-4cf0-b3a2-0f2ccbc48234"

}

},

"occurredOn": {

"date": "2016-05-26 06:06:07.000000",

"timezone\_type": 3, "timezone": "UTC"

}

},

{

"id": 2,

"version": 2,

"typeName": "Lw\\Domain\\Model\\Wish\\WishWasMade", "eventBody": {

"wish\_id": {

"id": "9e90435a-395c-46b0-b4c4-d4b769cbf201"

},

"user\_id": {

"id": "459a4ffc-cd57-4cf0-b3a2-0f2ccbc48234"

},

"address": ["john@example.com",](mailto:john@example.com) "content": "This is my new wish!"

},

"occurredOn": {

"date": "2016-05-26 06:06:27.000000",

"timezone\_type": 3, "timezone": "UTC"

},

"timeTaken": "650"

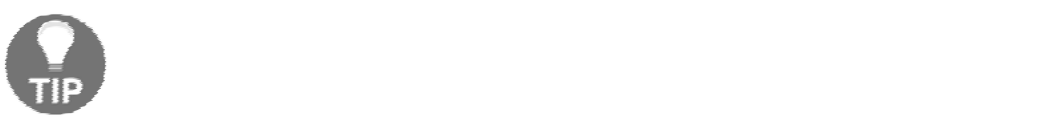
},

//...

]

As you can see in the previous example, we're exposing a set of Domain Events in a JSON REST API. In the output example, you can see a JSON representation of each of the Domain Events. There are some interesting points. First, the version field. Sometimes your Domain Events will evolve: they'll include more fields, they'll change the behavior of some existing fields, or they'll remove some existing fields. That's why it's important to add a version field in your Domain Events. If other Bounded Contexts are listening to such Events, they can use the version field to parse the Domain Event in different ways. You may have faced the same problem when versioning REST APIs.

Another point is the name. If you want to use the classname of the Domain Event, it may work in most cases. The problem is when a team decides to change the name of the class because of a refactoring. In this case, all Bounded Contexts listening to that name would stop working. This problem only occurs if you publish different Domain Events in the same queue. If you publish each Domain Event type in a different queue, it's not a real problem, but if you choose this approach, you'll face a different set of problems, such as receiving unordered events. Like in many other instances, there's a tradeoff involved. We strongly recommend you read *Enterprise Integration Patterns:* [Designing, Building, and Deploying](http://www.amazon.com/Enterprise-Integration-Patterns-Designing-Addison-Wesley-ebook/dp/B007MQLL4E) [Messaging Solutions](http://www.amazon.com/Enterprise-Integration-Patterns-Designing-Addison-Wesley-ebook/dp/B007MQLL4E). In this book, you'll learn different patterns for integrating multiple applications using asynchronous methods. Because Domain Events are messages sent in an integration channel, all messaging patterns also apply to them.



**Exercise**

Think about the pros and cons of having a REST API for Domain Events. Consider Bounded Context coupling. You can also try to implement a REST API for your current application.

## Wrap-Up

We've seen the tricks to model a proper DomainEvent with a base interface, we've seen where to publish the DomainEvent (the nearer to the Entities the better), and we've seen the strategies for spreading those DomainEvents to local and remote Bounded Contexts. Now, the only thing remaining is listening for a notification in the messaging system, reading it, and executing the corresponding Application Service or Command. We'll see how to do this in [Chapter 12](#_bookmark412), *Integrating Bounded Contexts* and [Chapter 5](#_bookmark156), *Services*.

# Modules

****

*当您将一些课程放在一个模块中时，您会告诉下一位研究您设计的开发人员，一起思考它们。 如果你的模型讲述一个故事，模块是章节.*

[Domain-Driven Design:Tackling Complexity in the Heart of Software](https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215)

*-Eric Evans*

构建应用程序遵循域驱动设计时，常见的问题是放置代码的位置。 特别是如果您使用的是PHP框架，了解推荐的代码放置方式，放置基础架构代码的位置以及应该如何构建模型中的不同概念.

在领域驱动设计中，这有一个战术模式：模块。 如今，每个人都在模块中构建代码。 所有语言都有某种工具将类和语言定义分组在一起。 Java有包。 Ruby有模块。 PHP有命名空间.

领域驱动设计更进一步将您的类打包和分组在一起，并为这些构建块提供语义含义。 事实上，它将模块视为模型的一部分。 作为模型的一部分，重要的是要找到最佳命名方式，将彼此靠近的域对象组合在一起，并保持不相关的域对象解耦。 不应将模块视为分离代码的方式，而应将其视为分离模型中有意义的概念的一种方式.

## 总体概述

如第1章“域驱动设计入门”中所述，我们的域在内部组织到子域中。 每个子域理想地由一个有界上下文建模和实现，但有时需要多个子域。 如果设计良好，每个有界上下文都是一个由团队开发和管理的独立系统。 我们的建议是使用整个应用程序来实现每个有界上下文。 这意味着两个有界上下文不会存在于相同的代码库中。 因此，它们可以独立部署，具有不同的开发周期，甚至可以使用不同的语言开发。 在有界上下文中，您将使用模块来将域对象分组在一起，这些对象之间保持强有力的关系。

## 在PHP中利用模块

在PHP 5.3之前，模块还没有完全支持。 但自PHP 5.3推出以来，我们可以使用PHP名称空间来实现模块模式。 由于历史原因，我们将介绍如何在PHP 5.3之前使用名称空间，但是您应该努力使用支持PHP名称空间的PHP版本。 最好的选择总是成为PHP的最新稳定版本.

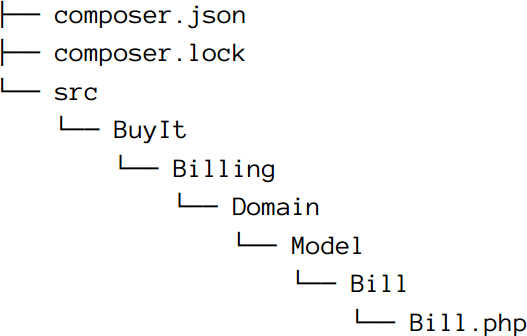
### 一级命名空间

常用的方法是使用标识贵公司的第一级名称空间。 这将有助于避免与第三方库的冲突。 如果您使用的是PSR-0，则会为命名空间提供一个真实文件夹; 如果你使用PSR-4，你不需要它。 我们会尽快进一步深入。 但首先，我们来看看PHP命名空间的约定.

### PEAR样式命名空间

在PHP 5.3之前，由于缺乏命名空间构造，所以使用了PEAR风格的命名空间。 PEAR是PHP扩展和应用程序库的首字母缩写，在过去的美好时代，它是一个可重用组件的存储库。 它仍然很活跃，但它不是很方便，也没有多少人使用它 - 特别是在Composer和Packagist被引入后。 作为可重用组件的来源，PEAR需要一种避免类名冲突的方法，因此贡献者开始在名称空间前面添加类名。 还有一些项目使用这种形式的命名空间（PHPUnit和Zend Framework 1来命名一对）。 PEAR样式的命名空间的一个例子:

以下是PEAR风格命名空间的一个例子:



使用PEAR样式名称空间的Bill实体的类名称将变为BuyIt\_Billing\_Domain\_Model\_Bill\_Bill。 然而，这有点难看，它并没有遵循主要的领域驱动设计经典之一：每一个类的名字都应该以无处不在的语言来命名。 出于这个原因，我们强烈不鼓励它的使用.

#### PSR-0和PSR-4命名空间

命名空间在引入PHP 5.3时进入场景，以及其他重要功能。 这是一个重大转变，一群最重要的框架协作者使用PHP-FIG（PHP Framework Interop Group的首字母缩略词）出现，试图标准化和统一框架和库创建的常见方面。 该组织发布的第一个PHP标准建议（PSR）是一个自动加载标准，简而言之，它使用名称空间提出了一个类和一个PHP文件之间的一对一关系。 今天，PSR-4（简化PSR-0，仍然保持类和物理PHP文件之间的关系）是构建代码的首选和推荐方式。 我们认为这应该是用于在项目中实现模块的.

回顾前一节中所示的相同文件夹结构，我们来看看PSR-0有哪些变化。 使用名称空间和PSR-0的账单实体的类名将简单地变成比尔，而完全合格的类名称将是BuyIt \ Billing \ Domain \ Model \ Bill \ Bill.

正如您所看到的，这使我们能够根据泛在语言来命名Domain对象，并且这是构建和组织代码的首选方式。 如果您使用的是Composer，就像您应该做的那样，您需要在composer.json文件中设置一些自动加载配置:

...

"autoload": {

"psr-0": {

"BuyIt\\": "src/BuyIt/"

}

},

"autoload-dev": {

"psr-0": {

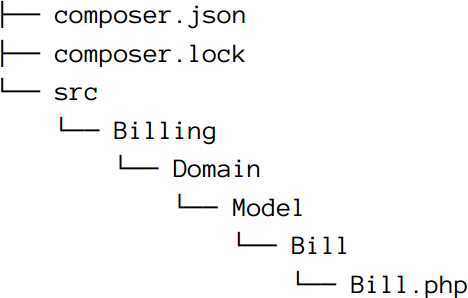
"BuyIt": "tests/BuyIt/"

}

},

...

If you're not using PSR-4 or you haven't migrated from PSR-0 yet, we strongly recommend doing so. You can get rid of the first-level namespace folder, and your code structure will better match the Ubiquitous Language:



However, in order to avoid the collision with third-party libraries, it's still recommended to add the first-level namespace in your composer.json file:

...

"autoload": {

"psr-4": {

"BuyIt\\": "src/"

}

},

"autoload-dev": {

"psr-4": {

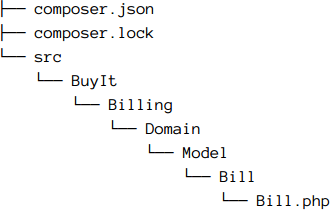
"BuyIt\\": "tests/"

}

},

...

如果您更喜欢拥有第一级名称空间但使用PSR-4，则需要做一些小的更改:



...

"autoload": {

"psr-4": {

"BuyIt\\": "src/BuyIt/"

}

},

"autoload-dev": {

"psr-4": {

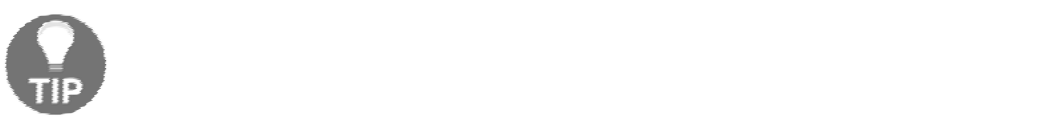
"BuyIt\\": "tests/BuyIt/"

}

},

...

正如您在示例中可能已经注意到的那样，我们分割src并测试文件夹。 这样做是为了优化由Composer生成的自动加载文件，并且它会减少存储类图所需的内存。 它还将帮助您在生成单元测试代码覆盖率报告时设置白名单和黑名单选项。 如果您想更多地了解Composer的自动加载配置，请查看文档.



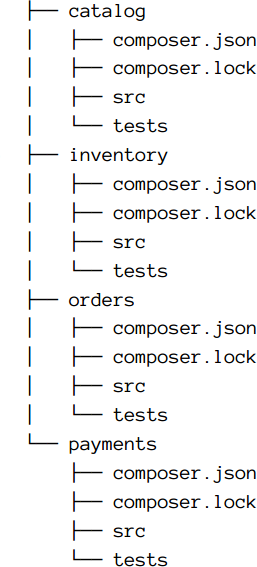
**What about PHAR files?**

They could also be used, however, we don't recommend this. As an exercise, make a list of pros and cons for using PHAR files to model modules.

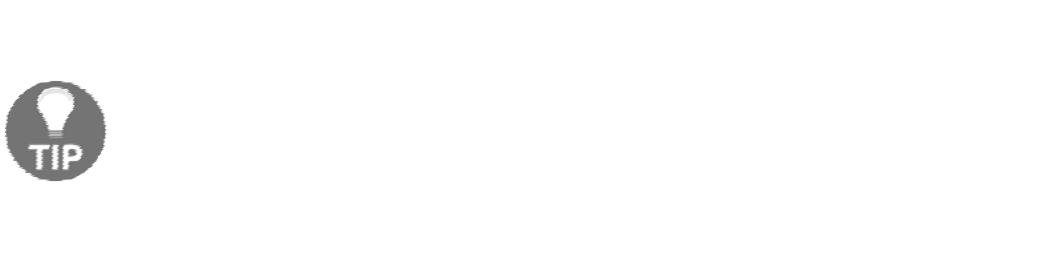
## 有界的上下文和应用程序

如果我们以一个名为BuyIt的虚构公司为例来讨论一个电子商务域，那么为每个解决特定领域区域的不同有界上下文创建一个不同的应用程序可能是有意义的.

如果某些不同的有界上下文是订单管理，支付管理，目录管理和库存管理，我们建议为每一个应用程序:



每个应用程序都公开了所需的任何一组传递机制。 随着微服务趋势，越来越多的人构建了有界上下文，最终将REST API暴露给外部世界。 但是，有界上下文不仅仅是一个API。 请记住，API只是众多交付机制之一; 一个有界的上下文可以提供一个web界面来进行交互.



**两个有界的上下文可以在同一个应用程序中吗？ 另一种方式呢?**

最好的选择是一个子域，一个有界上下文和一个应用程序。 如果我们用两个应用程序实现了一个有界上下文，维护和部署会有点棘手。 对于实现两个有界上下文的应用程序，部署过程，运行测试的时间以及合并问题可能会降低开发速度.

请注意，每个有界的上下文名称代表了我们电子商务领域中的一个有意义的概念，并且以无处不在的语言:

**目录以保存与产品描述，产品组合等相关的所有代码**.

**存货以保存与产品库存管理有关的所有代码**.

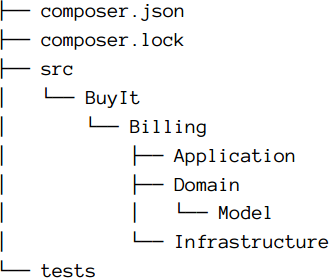
**订单保存所有与订单处理系统相关的代码。 它将包含负责处理订单的有限状态机**.

**支付所有与付款，账单和运单相关的代码**.

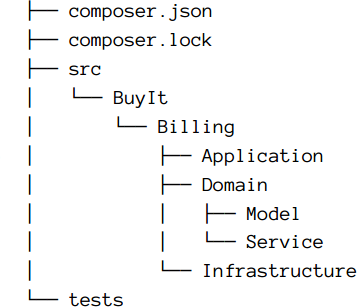
## 在模块中构造代码

让我们进一步深入一个有界的上下文。 举例来说，订单上下文并检查结构细节。 顾名思义，这个有界上下文负责代表订单流经的所有流程 - 从创建到交付给购买它的客户。 此外，它是一个独立的应用程序，因此它包含一个源代码文件夹和一个测试文件夹。 源代码文件夹包含此有界上下文所需的所有代码：域代码，基础架构代码和应用程序层。

下图应该说明组织:



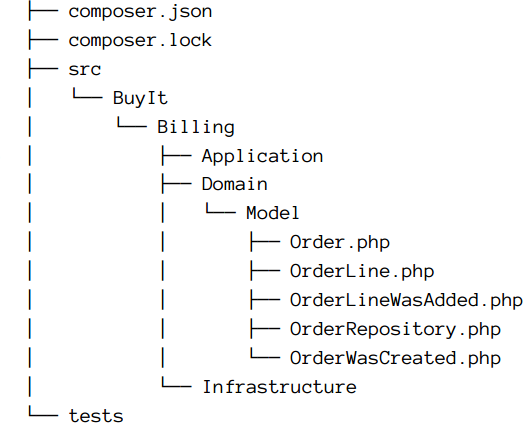
所有代码的前缀都是以组织名称（本例中为BuyIt）命名的供应商命名空间，并且包含两个子文件夹：域保存所有域代码，基础架构保存基础架构层，从而将所有域逻辑与 基础设施层的细节。 遵循这种结构，我们清楚地表明，我们将使用六边形架构作为基础架构。 以下是可以使用的替代结构的示例:



上述结构样式使用额外的子文件夹来存储域模型中定义的服务。 虽然这个组织可能有意义，但我们的偏好是不使用它，因为这种分离代码的方式往往更侧重于架构元素，而不是模型中的相关概念。 我们相信这种风格很容易导致在域模型之上的某种服务层，这不一定是坏事。 请记住，域服务用于描述域中不属于实体或值对象的操作。 所以从现在开始，我们将坚持以前的代码组织.

可以直接在Domain / Model子文件夹中放置代码。 例如，习惯上可以在其中放置通用接口和服务，例如DomainEventPublisher或DomainEventSubscriber.

如果我们必须建立一个订单管理上下文的模型，我们可能会有一个带有其存储库和所有状态信息的订单实体。 所以我们的第一次尝试是将所有这些元素直接放到域/模型子文件夹中。 乍一看，这看起来可能是最简单的方法：



### 设计指南

考虑在实现模块时要注意的一些基本规则和典型问题：

命名空间应该以无处不在的语言来命名.

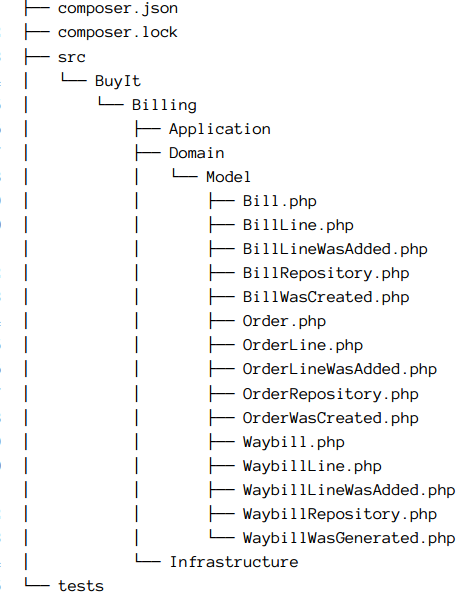
不要根据模式或构建块（值对象，服务，实体等）命名您的名称空间。

创建名称空间，以便内部内容尽可能松散地与其他名称空间耦合.

重构命名空间与您的代码一样。 移动它们，重命名它们，对它们进行分组，提取它们，等等.

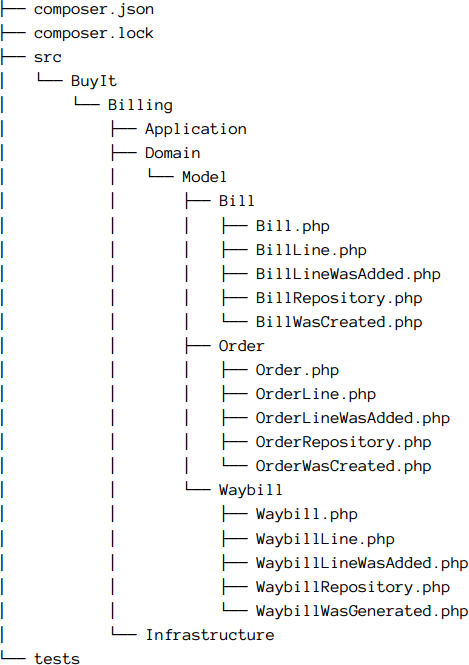
请勿使用商业产品名称，因为它们可能会更改。 坚持无所不在的语言.

我们已将Order和OrderLine实体，OrderLineWasAdded和OrderWasCreated事件以及OrderRepository放入相同的子文件夹域/模型中。 这种结构可能很好，但那是因为我们仍然有一个简单的模型。 条例实体及其存储库如何？ 或Waybill实体及其各自的存储库？ 让我们添加所有这些元素，看看它们如何适合实际的代码结构:

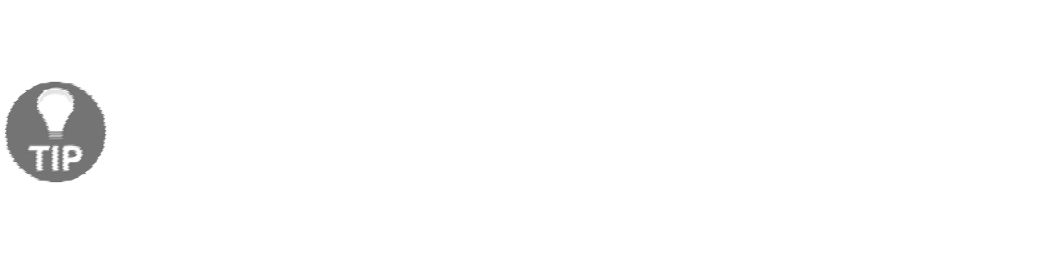


虽然这种代码组织形式可以很好，但从长远来看，它可能变得不实际并且不可维护。 每次迭代和添加新功能时，模型都会变得更大，并且子文件夹将会消耗更多的代码。 我们需要以一种让我们对模型一目了然的方式来分割代码。 没有技术问题，只是领域问题。 为了实现这一点，我们可以使用无处不在的语言来分割模型，找到有意义的概念，帮助我们按照域来逻辑分组.

To do this, we could try the following approach:



这样，从概念上讲，代码更有条理。 正如Eric Evans在蓝皮书中指出的那样，模块是沟通的一种方式，因为它们为我们提供关于域模型如何在内部工作的见解，并帮助我们增强内聚力并减少概念之间的耦合。 如果我们看一下前面的例子，我们可以看到Order和OrderLine的概念是强相关的，所以它们生活在同一个模块中。 另一方面，Order和Waybill虽然共享相同的上下文，但它们是不同的概念，所以它们生活在不同的模块中。 模块不仅是将模型中的相关概念分组的一种方式，而且也是表达模型设计的一部分的一种方式.



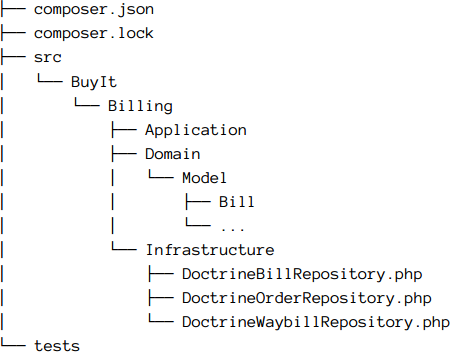
**Should We Place Repositories, Factories, Domain Events, and Services in Their Own Subfolders?**

Effectively, they could be placed into their own subfolders, but it's strongly discouraged. In doing so, we would be mixing technical concerns and Domain concerns — remember that the module's main interest is to group related concepts from the Domain model and decouple them from non-related concepts. Modules don't separate code but instead separate meaningful concepts.

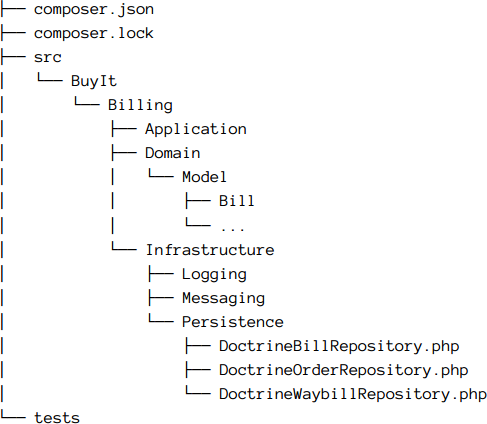
### 基础设施层中的模块

到目前为止，我们一直在讨论如何在Domain层中构建和组织代码，但我们几乎没有提到基础架构层。 由于我们使用六边形体系结构来反转域层和基础设施层之间的依赖关系，因此我们需要一个位置，让我们可以将所有在Domain层中定义的接口实现。 回到账单上下文的例子，我们需要一个地方来实现BillRepository，OrderRepository和WaybillRepository。

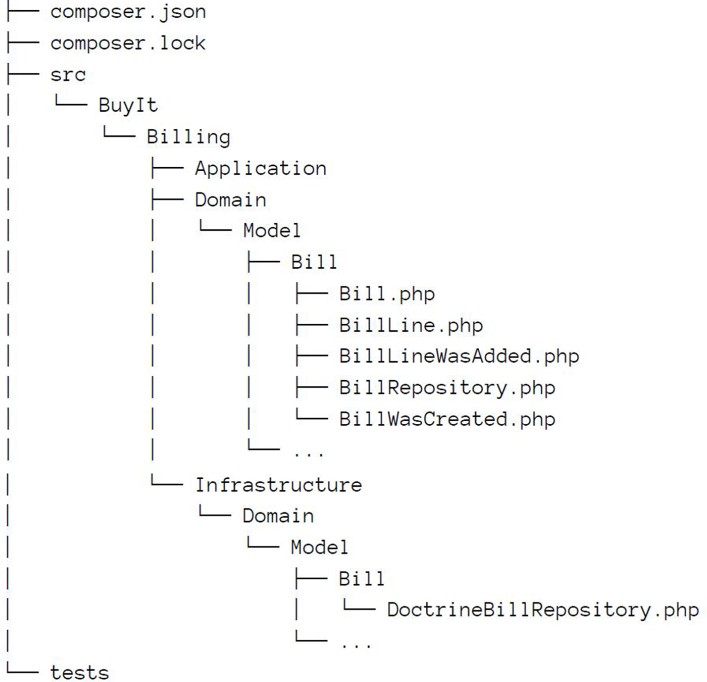
很明显，他们应该被放置到基础设施文件夹中，但是在哪里？ 假设我们决定使用Doctrine ORM来实现持久层。 我们如何将我们的存储库的Doctrine实现放入基础架构文件夹？ 让我们直接看看它的外观:



我们可以保持这种状态，但正如我们在领域层看到的那样，这种结构和组织会在几个模型迭代中快速腐烂并变得混乱。 每当模型增长时，它可能需要更多的基础架构，最后我们会混合不同的技术问题，如持久性，消息传递，日志记录等等。 我们第一次尝试避免混乱的基础架构实现是为有界上下文中的每个技术问题定义一个模块:



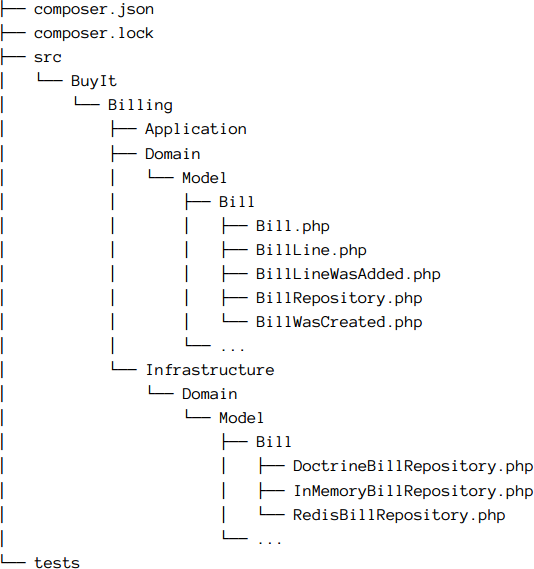
这看起来好多了，而且在长期内比我们的第一次尝试更容易维护。 然而，我们的命名空间与泛在语言缺乏某种关系。 我们来考虑一个变体:



好多了。 它与我们的域模型组织相匹配，但在基础设施层内

- 加上一切似乎更容易找到。 如果您事先知道您将始终拥有单一的持久性机制，那么您可以坚持这种结构和组织。 这是相当简单和容易维护.

但是当你需要使用几种持久机制时呢？ 现在，拥有关系持久性机制和某种类似Redis或Riak的共享内存中持久性，或者具有某种本地内存中实现来测试代码的情况很常见。 让我们看看这是如何符合实际的方法：

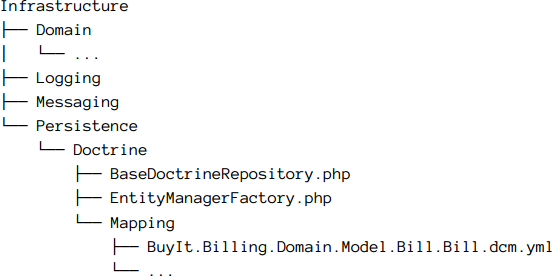


我们推荐以上。 但是，所有Repository实现都位于同一个模块中。 当拥有这么多不同的技术时，这看起来有点奇怪。 如果您觉得它很有趣，可以创建一个附加模块，以便通过其基础技术对相关实现进行分组:

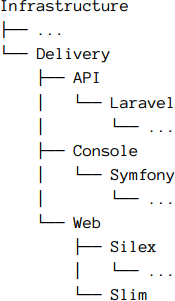


这种方法与单元测试组织相似。 但是，有类，配置，模板等。 无法与域模型匹配。 这就是为什么您可能在基础架构内部有与特定技术相关的附加模块.

Where should you place Doctrine mapping files or Twig templates?



正如你所看到的，为了使Doctrine工作，我们需要一个EntityManagerFactory和所有的映射文件。 我们还可能包含任何其他基础类所需的基础结构对象。 因为它们与我们的域模型没有直接关系，所以最好将这些资源放在不同的模块中。 交付机制（API，Web，控制台命令等）也会发生同样的情况。 实际上，您可以为每个传递机制使用不同的PHP框架或库:



In the previous example, we were using the Laravel Framework for serving the API, the Symfony Console Component as the entry point for the command line, and Silex and Slim for the web delivery mechanism. Regarding the User Interface, you should place it inside each delivery mechanism. However, if there's any chance to share the UI between different delivery mechanisms, you can create a module called UI at the same level as Persistence or Delivery. In general, our suggestion is struggling with how the frameworks tell you to organize your code. Frameworks should obey you, and not the other way around.

### Mixing Different Technologies

In large business-critical applications, it's quite common to have a mix of several technologies. For example, in read-intensive web applications, you usually have some sort of denormalized data source (Solr, Elasticsearch, Sphinx, and so on.) that provides all the reads of the application, while a traditional RDBMS like MySQL or Postgres is mainly responsible for handling all the writes. When this occurs, one of the concerns that normally arises is whether we can have read operations go with the search engine and write operations go with the traditional RDBMS data source. Our general advice here is that these kind of situations are a smell for CQRS, since we need to scale the reads and the writes of the application independently. So if you can go with CQRS, that's likely the best choice.

But if for any reason you can't go with CQRS, an alternative approach is needed. In this situation, the use of the Proxy pattern from the *Gang of Four* comes in handy. We can define an implementation of a Repository in terms of the Proxy pattern:

namespace BuyIt\Billing\Infrastructure\FullTextSearching\Elastica;

use BuyIt\Billing\Domain\Model\Order\OrderRepository;

use BuyIt\Billing\Infrastructure\Domain\Model\Order\Doctrine\ DoctrineOrderRepository;

class ElasticaOrderRepository implements OrderRepository

{

private $client;

private $baseOrderRepository;

public function construct( Client $client,

DoctrineOrderRepository $baseOrderRepository

) {

$this->client = $client;

$this->baseOrderRepository = $baseOrderRepository;

}

public function find($id)

{

return $this->baseOrderRepository->find($id);

}

public function findBy(array $criteria)

{

$search = new \Elastica\Search($this->client);

// ...

return $this->toOrder($search->search());

}

public function add($anOrder)

{

// First we attempt to add it to the Elastic index

$ordersIndex = $this->client->getIndex('orders');

$orderType = $ordersIndex->getType('order');

$orderType->addDocument( new \ElasticaDocument(

$anOrder->id(),

$this->toArray($anOrder)

)

);

$ordersIndex->refresh();

// When it is done, we attempt to add it to the RDBMS store

$this->baseOrderRepository->add($anOrder);

}

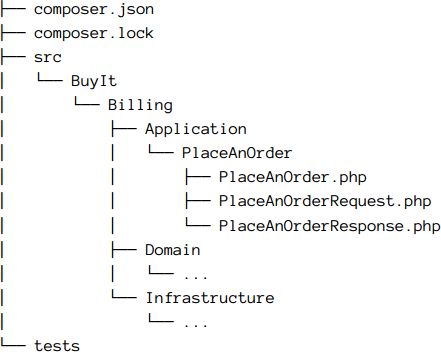
}

此示例使用DoctrineOrderRepository和Elastica客户端（一个与Elasticsearch服务器交互的客户端）提供了一个天真的实现。 请注意，对于某些操作，我们使用RDBMS数据源，而对于其他操作，我们使用的是Elastica客户端。 另请注意，添加操作由两部分组成。 第一个尝试将Order存储到Elasticsearch索引，第二个尝试将Order存储到关系数据库中，并将操作委托给Doctrine实现。 考虑到这只是一个例子和一个方法。 它可能可以改进 - 例如，现在整个添加操作是同步的。 例如，我们可以将操作排队到某种将订单存储在Elasticsearch中的消息中间件。 根据您的需求，有很多可能性和改进.

### 应用层中的模块

We've seen Domain and Infrastructure modules, so now let's take a look at the Application layer. In Domain-Driven Design, we suggest using Application Services as a way of decoupling the client from both the Domain Model and the necessary knowledge on how to interact with it. As you'll see in [Chapter 11](#_bookmark358), *Application*, an Application Service is built with its dependencies, is executed with a DTO request, and returns a DTO response.

It can also use an output dependency to return the result:



Our suggestion is to create modules around Application Services. Each module will hold its request and response. If you're using the Data Transformer as an output dependency, follow the Infrastructure approach as you would with the UI.

## Wrap-Up

Modules are a way of grouping and separating concepts in our application. Modules should be named following the Ubiquitous Language. We shouldn't forget that modules are a way to communicate high-level concepts, which aids us in keeping coupling low and cohesion high. We've seen that we could create meaningful modules even in old versions of PHP by using prefixes. Nowadays, it's easy to build our modules following the PSR-0 and PSR-4 namespacing conventions.

# Aggregates

****

Aggregates are probably the most difficult building blocks of Domain-Driven Design. They're hard to understand, and they're even harder to properly design. But don't worry; we're here to help you. However, before jumping into Aggregates, there are some key concepts we need to go through first: transactions and concurrency strategies.

## Introduction

If you've worked with e-commerce applications, it's likely you've faced bugs related to data inconsistencies in your database. For example, consider a shopping order with a total amount of $99.99, which doesn't match with the sum of the amounts of each line in the order, $89.99. Where did that extra $10 come from?

Or, consider a website that sells tickets for the cinema. There's a movie theater with 100 available seats, and after a successful movie promotion, everyone is on the website waiting for the tickets to become available for purchase. Once you open the sales, everything happens fast and you somehow end up selling 102 tickets. You may have specified that there are only 100 seats, but for some reason you exceeded that threshold.

You might even have experience with tracking systems such as JIRA or Redmine. Think about a team of Developers, QAs, and a Product Owner. What could happen if everyone sorts and moves around user stories during a planning meeting and then saves? The final backlog or sprint prioritization would be the one from the team member who saved last.

In general, data inconsistencies occur when we deal with our persistence mechanism in a non-atomic way. An example of this is when you send three queries to a database and some of them work and some don't. The final state of the database is inconsistent. Sometimes, you want these three queries to succeed or fail all together, and that can be fixed with transactions. However, be careful, because as you will see in this chapter, not all inconsistencies are fixed with transactions. In fact, sometimes other data inconsistencies need locking or concurrency strategies. These kinds of tools might come up against your application performance, so be aware that there's a tradeoff involved.

You may think that these kinds of data inconsistencies only occur in databases, but that's not true. For example, if we use a document-oriented database such as Elasticsearch, we can have data inconsistency between two documents. Furthermore, most of the NoSQL persistence storage systems don't support ACID transactions. This means you can't persist or update more than one document in a single operation. So, if we make different requests to Elasticsearch, one may fail, leaving the data persisted in Elasticsearch inconsistent.

Keeping data consistent is a challenge. Not leaking infrastructure issues into the Domain is a bigger challenge. Aggregates aim to help you with both of these things.

## Key Concepts

Persistence engines — and databases in particular — have some features for fighting data inconsistencies: ACID, constraints, referential integrity, locking, concurrency controls, and transactions. Let's review these concepts before working with Aggregates.

Most of these concepts are on the Internet and available to the public. We want to thank the people at Oracle, PostgreSQL, and Doctrine for doing amazing work with their documentation. They have carefully defined and explained these important terms, and rather than reinvent the wheel, we've compiled some of these official explanations to share with you.

### ACID

As discussed in a previous section, **ACID** stands for **atomicity**, **consistency**, **isolation**, and

**durability**. According to the [MySQL Glossary](http://dev.mysql.com/doc/refman/5.7/en/glossary.html#glos_acid):

These properties are all desirable in a database system, and are all closely tied to the notion of a transaction. For example, the transactional features of MySQL InnoDB engine adhere to the ACID principles.

Transactions are *atomic* units of work that can be committed or rolled back. When a transaction makes multiple changes to the database, either all the changes succeed when the transaction is committed, or all the changes are undone when the transaction is rolled back.

The database remains in a *consistent* state at all times, after each commit or rollback, and while transactions are in progress. If related data is being updated across multiple tables, queries see either all old values or all new values, not a mix of old and new values.

Transactions are protected *isolated* from each other while they are in progress. They cannot interfere with each other or see each other's uncommitted data. This isolation is achieved through the locking mechanism. Experienced users can adjust the isolation level, trading off less protection in favor of increased performance and concurrency, when they can be sure that the transactions really do not interfere with each other.

The results of transactions are *durable*: once a commit operation succeeds, the changes made by that transaction are safe from power failures, system crashes, race conditions, or other potential dangers that many non-database applications are vulnerable to. Durability typically involves writing to disk storage, with a certain amount of redundancy to protect against power failures or software crashes during write operations.

### Transactions

According to the [PostgreSQL 8.2.23 Documentation](https://www.postgresql.org/docs/8.2/static/tutorial-transactions.html):

Transactions are a fundamental concept of all database systems. The essential point of a transaction is that it bundles multiple steps into a single, all-or-nothing operation. The intermediate states between the steps are not visible to other concurrent transactions, and if some failure occurs that prevents the transaction from completing, then none of the steps affect the database at all.

For example, consider a bank database that contains balances for various customer accounts, as well as total deposit balances for branches. Suppose that we want to record a payment of $100.00 from Alice's account to Bob's account. Simplifying outrageously, the SQL commands for this might look like:

UPDATE accounts

SET balance = balance - 100.00 WHERE name = 'Alice';

UPDATE branches

SET balance = balance - 100.00

WHERE name = (SELECT branch\_name FROM accounts WHERE name ='Alice');

UPDATE accounts

SET balance = balance + 100.00 WHERE name = 'Bob';

UPDATE branches

SET balance = balance + 100.00

WHERE name = (SELECT branch\_name FROM accounts WHERE name ='Bob');

The details of these commands are not important here. The important point is that there are several separate updates involved to accomplish this rather simple operation. Our bank's officers will want to be assured that either all these updates happen, or none of them happen. It would certainly not do for a system failure to result in Bob receiving $100.00 that was not debited from Alice. Nor would Alice long remain a happy customer if she was debited without Bob being credited. We need a guarantee that if something goes wrong partway through the operation, none of the steps executed so far will take effect. Grouping the updates into a transaction gives us this guarantee. A transaction is said to be atomic: from the point of view of other transactions, it either happens completely or not at all.

We also want a guarantee that once a transaction is completed and acknowledged by the database system, it has indeed been permanently recorded and won't be lost even if a crash ensues shortly thereafter. For example, if we are recording a cash withdrawal by Bob, we do not want any chance that the debit to his account will disappear in a crash just after he walks out the bank door. A transactional database guarantees that all the updates made by a transaction are logged in permanent storage (That is: on disk) before the transaction is reported complete.

Another important property of transactional databases is closely related to the notion of atomic updates: when multiple transactions are running concurrently, each one should not be able to see the incomplete changes made by others. For example, if one transaction is busy totalling all the branch balances, it would not do for it to include the debit from Alice's branch but not the credit to Bob's branch, nor vice versa. So transactions must be all-or-nothing not only in terms of their permanent effect on the database, but also in terms of their visibility as they happen. The updates made so far by an open transaction are invisible to other transactions until the transaction completes, whereupon all the updates become visible simultaneously.

In PostgreSQL, for example, a transaction is set up by surrounding the SQL commands of the transaction with BEGIN and COMMIT commands. So our banking transaction would actually look like:

BEGIN;

UPDATE accounts

SET balance = balance - 100.00 WHERE name = 'Alice';

-- etc etc COMMIT;

If, partway through the transaction, we decide we do not want to commit (perhaps we just noticed that Alice's balance went negative), we can issue the command ROLLBACK instead of COMMIT, and all our updates so far will be canceled.

PostgreSQL actually treats every SQL statement as being executed within a transaction. If you do not issue a BEGIN command, then each individual statement has an implicit BEGIN and (if successful) COMMIT wrapped around it. A group of statements surrounded by BEGIN and COMMIT is sometimes called a transaction block.

All this is happening within the transaction block, so none of it is visible to other database sessions. When and if you commit the transaction block, the committed actions become visible as a unit to other sessions, while the rolled-back actions never become visible at all.

### Isolation Levels

According to the [MySQL Glossary](http://dev.mysql.com/doc/refman/5.7/en/glossary.html#glos_isolation_level), transaction isolation is:

One of the foundations of database processing. Isolation is the "I" in the acronym ACID. The isolation level is the setting that fine-tunes the balance between performance and reliability, consistency, and reproducibility of results when multiple transactions are making changes and performing queries at the same time.

From highest amount of consistency and protection to the least, the isolation levels supported by InnoDB, for example, are: SERIALIZABLE, REPEATABLE READ, READ COMMITTED, and READ UNCOMMITTED.

With InnoDB tables, many users can keep the default isolation level REPEATABLE READ for all operations. Expert users might choose the read committed level as they push the boundaries of scalability with OLTP processing, or during data warehousing operations where minor inconsistencies do not affect the aggregate results of large amounts of data. The levels on the edges (SERIALIZABLE and READ UNCOMMITTED) change the processing behavior to such an extent that they are rarely used.

### Referential Integrity

According to the [MySQL Glossary](http://dev.mysql.com/doc/refman/5.7/en/glossary.html#glos_referential_integrity), referential integrity is:

The technique of maintaining data always in a consistent format, part of the ACID philosophy. In particular, data in different tables is kept consistent through the use of foreign key constraints, which can prevent changes from happening or automatically propagate those changes to all related tables. Related mechanisms include the unique constraint, which prevents duplicate values from being inserted by mistake, and the NOT NULL constraint, which prevents blank values from being inserted by mistake.

### Locking

According to the [MySQL Glossary](http://dev.mysql.com/doc/refman/5.7/en/glossary.html#glos_locking), locking is:

The system of protecting a transaction from seeing or changing data that is being queried or changed by other transactions. The locking strategy must balance reliability and consistency of database operations (the principles of the ACID philosophy) against the performance needed for good concurrency. Fine-tuning the locking strategy often involves choosing an isolation level and ensuring all your database operations are safe and reliable for that isolation level.

### Concurrency

According to the [MySQL Glossary](http://dev.mysql.com/doc/refman/5.7/en/glossary.html#glos_concurrency), concurrency is:

The ability of multiple operations (in database terminology, transactions) to run simultaneously, without interfering with each other. Concurrency is also involved with performance, because ideally the protection for multiple simultaneous transactions works with a minimum of performance overhead, using efficient mechanisms for locking.

#### Pessimistic Concurrency Control (PCC)

The book [Elasticsearch: The Definitive Guide](https://github.com/elastic/elasticsearch-definitive-guide/blob/master/030_Data/40_Version_control.asciidoc) by Clinton Gormley and Zachary Tong discusses PCC, saying that:

Widely used by relational databases, this approach assumes that conflicting changes are likely to happen and so blocks access to a resource in order to prevent conflicts. A typical example is locking a row before reading its data, ensuring that only the thread that placed the lock is able to make changes to the data in that row.

##### With Doctrine

According to the [Doctrine 2 ORM Documentation](http://doctrine-orm.readthedocs.io/projects/doctrine-orm/en/latest/reference/transactions-and-concurrency.html#locking-support) on locking support:

Doctrine 2 offers support for Pessimistic- and Optimistic-locking strategies natively. This allows to take very fine-grained control over what kind of locking is required for your Entities in your application.

According to the [Doctrine 2 ORM Documentation](http://doctrine-orm.readthedocs.io/projects/doctrine-orm/en/latest/reference/transactions-and-concurrency.html#pessimistic-locking) on pessimistic locking:

Doctrine 2 supports Pessimistic Locking at the database level. No attempt is being made to implement pessimistic locking inside Doctrine, rather vendor-specific and ANSI-SQL commands are used to acquire row-level locks. Every Doctrine Entity can be part of a pessimistic lock, there is no special metadata required to use this feature.

However for Pessimistic Locking to work you have to disable the Auto-Commit Mode of your Database and start a transaction around your pessimistic lock use- case using the *Explicit Transaction Demarcation*. Doctrine 2 will throw an Exception if you attempt to acquire an pessimistic lock and no transaction is running.

Doctrine 2 currently supports two pessimistic lock modes:

Pessimistic Write Doctrine\DBAL\LockMode::PESSIMISTIC\_WRITE, locks the underlying database rows for concurrent Read and Write Operations.

Pessimistic Read Doctrine\DBAL\LockMode::PESSIMISTIC\_READ, locks other concurrent requests that attempt to update or lock rows in write mode.

You can use pessimistic locks in three different scenarios: Using EntityManager#find($className, $id,

\Doctrine\DBAL\LockMode::PESSIMISTIC\_WRITE) or

EntityManager#find($className, $id,

\Doctrine\DBAL\LockMode::PESSIMISTIC\_READ)

Using EntityManager#lock($entity,

\Doctrine\DBAL\LockMode::PESSIMISTIC\_WRITE) or

EntityManager#lock($entity,

\Doctrine\DBAL\LockMode::PESSIMISTIC\_READ)

Using Query#setLockMode(\Doctrine\DBAL\LockMode::PESSIMISTIC\_WRITE) or Query#setLockMode(\Doctrine\DBAL\LockMode::PESSIMISTIC\_READ)

#### Optimistic Concurrency Control

According to [Wikipedia](https://en.wikipedia.org/wiki/Optimistic_concurrency_control):

**Optimistic concurrency control** (**OCC**) is a concurrency control method applied to transactional systems such as relational database management systems and software transactional memory. OCC assumes that multiple transactions can frequently complete without interfering with each other. While running, transactions use data resources without acquiring locks on those resources. Before committing, each transaction verifies that no other transaction has modified the data it has read. If the check reveals conflicting modifications, the committing transaction rolls back and can be restarted. Optimistic concurrency control was first proposed by H.T. Kung.

OCC is generally used in environments with low data contention. When conflicts are rare, transactions can complete without the expense of managing locks and without having transactions wait for other transactions' locks to clear, leading to higher throughput than other concurrency control methods. However, if contention for data resources is frequent, the cost of repeatedly restarting transactions hurts performance significantly; it is commonly thought that other concurrency control methods have better performance under these conditions.

However, locking-based "pessimistic" methods also can deliver poor performance because locking can drastically limit effective concurrency even when deadlocks are avoided.

##### With Elasticsearch

According to [Elasticsearch: The Definitive Guide](https://github.com/elastic/elasticsearch-definitive-guide/blob/master/030_Data/40_Version_control.asciidoc), when OCC is used by Elasticsearch:

This approach assumes that conflicts are unlikely to happen and doesn't block operations from being attempted. However, if the underlying data has been modified between reading and writing, the update will fail. It is then up to the application to decide how it should resolve the conflict. For instance, it could reattempt the update, using the fresh data, or it could report the situation to the user.

Elasticsearch is distributed. When documents are created, updated, or deleted, the new version of the document has to be replicated to other nodes in the cluster.

Elasticsearch is also asynchronous and concurrent, meaning that these replication requests are sent in parallel, and may arrive at their destination out of sequence. Elasticsearch needs a way of ensuring that an older version of a document never overwrites a newer version.

Every document has a \_version number that is incremented whenever a document is changed. Elasticsearch uses this \_version number to ensure that changes are applied in the correct order. If an older version of a document arrives after a new version, it can simply be ignored.

We can take advantage of the \_version number to ensure that conflicting changes made by our application do not result in data loss. We do this by specifying the version number of the document that we wish to change. If that version is no longer current, our request fails.

Let's create a new blog post:

PUT /website/blog/1/\_create

{

"title": "My first blog entry", "text": "Just trying this out..."

}

The response body tells us that this newly created document has \_version number

1. Now imagine that we want to edit the document: we load its data into a web form, make our changes, and then save the new version.

First we retrieve the document:

GET /website/blog/1

The response body includes the same \_version number of 1:

{

"index": "website",

"type": "blog",

"id": "1",

"version": 1,

"found": true,

"\_source": {

"title": "My first blog entry", "text": "Just trying this out..."

}

}

Now, when we try to save our changes by reindexing the document, we specify the version to which our changes should be applied. We want this update to succeed only if the current \_version of this document in our index is version 1:

PUT /website/blog/1version=1

{

"title": "My first blog entry",

"text": "Starting to get the hang of this..."

}

This request succeeds, and the response body tells us that the \_version has been incremented to 2:

{

"index": "website",

"type": "blog",

"id": "1",

"version": 2, "created": false

}

However, if we were to rerun the same index request, still specifying version=1, Elasticsearch would respond with a 409 Conflict HTTP response code, and a body like the following:

{

"error": {

"root\_cause": [{

"type": "version\_conflict\_engine\_exception", "reason":

"[blog][1]: version conflict,current[2],provided

[1]",

}],

"index": "website",

"shard": "3"

"type": "version\_conflict\_engine\_exception" , "reason":

"[blog][1]:version conflict,current [2],provided[1]", "index": "website",

"shard": "3"

},

"status": 409

}

This tells us that the current \_version number of the document in Elasticsearch is 2, but that we specified that we were updating version 1.

What we do now depends on our application requirements. We could tell the user that somebody else has already made changes to the document, and to review the changes before trying to save them again. Alternatively, as in the case of the widget stock\_count previously, we could retrieve the latest document and try to reapply the change.

All APIs that update or delete a document accept a version parameter, which allows you to apply optimistic concurrency control to just the parts of your code where it makes sense.

##### With Doctrine

According to the [Doctrine 2 ORM Documentation](http://doctrine-orm.readthedocs.io/projects/doctrine-orm/en/latest/reference/transactions-and-concurrency.html#optimistic-locking) on optimistic locking:

Database transactions are fine for concurrency control during a single request. However, a database transaction should not span across requests, the so-called "user think time". Therefore a long-running "business transaction" that spans multiple requests needs to involve several database transactions. Thus, database transactions alone can no longer control concurrency during such a long-running business transaction. Concurrency control becomes the partial responsibility of the application itself.

Doctrine has integrated support for automatic optimistic locking via a version field. In this approach any entity that should be protected against concurrent modifications during long-running business transactions gets a version field that is either a simple number (mapping type: integer) or a timestamp (mapping type: datetime). When changes to such an entity are persisted at the end of a long-running conversation the version of the entity is compared to the version in the database and if they don't match, an OptimisticLockException is thrown, indicating that the entity has been modified by someone else already.

You designate a version field in an entity as follows. In this example we'll use an integer:

class User

{

// ...

/\*\* @Version @Column(type="integer") \*/ private $version;

// ...

}

When a version conflict is encountered during EntityManager#flush(), an OptimisticLockException is thrown and the active transaction rolled back (or marked for rollback). This exception can be caught and handled. Potential responses to an OptimisticLockException are to present the conflict to the user or to refresh or reload objects in a new transaction and then retrying the transaction.

With PHP promoting a share-nothing architecture, the time between showing an update form and actually modifying the entity can in the worst scenario be as long as your applications session timeout. If changes happen to the entity in that time frame you want to know directly when retrieving the entity that you will hit an optimistic locking exception:

You can always verify the version of an entity during a request either when calling

EntityManager#find():

use Doctrine\DBAL\LockMode;

use Doctrine\ORM\OptimisticLockException;

$theEntityId = 1;

$expectedVersion = 184; try{

$entity = $em->find( 'User',

$theEntityId, LockMode::OPTIMISTIC,

$expectedVersion

);

// do the work

$em->flush();

} catch (OptimisticLockException $e){ echo

'Sorry, someone has already changed this entity.' . 'Please apply the changes again!';

}

Or you can use EntityManager#lock() to find out:

use DoctrineDBALLockMode;

use DoctrineORMOptimisticLockException;

$theEntityId = 1;

$expectedVersion = 184;

$entity = $em->find('User', $theEntityId); try {

// assert version em−>lock(entity, LockMode::OPTIMISTIC,

$expectedVersion);

} catch (OptimisticLockException $e){ echo

'Sorry, someone has already changed this entity.' . 'Please apply the changes again!';

}

According to [Doctrine 2 ORM Documentation's](http://doctrine-orm.readthedocs.io/projects/doctrine-orm/en/latest/reference/transactions-and-concurrency.html#important-implementation-notes) important implementation notes:

You can easily get the optimistic locking workflow wrong if you compare the wrong versions. Say you have Alice and Bob editing a hypothetical blog post:

Alice reads the headline of the blog post being "Foo", at optimistic lock version 1 (GET Request)

Bob reads the headline of the blog post being "Foo", at optimistic lock version 1 (GET Request)

Bob updates the headline to "Bar", upgrading the optimistic lock version to 2 (POST Request of a Form)

Alice updates the headline to "Baz", ... (POST Request of a Form)

Now at the last stage of this scenario the blog post has to be read again from the database before Alice's headline can be applied. At this point you will want to check if the blog post is still at version 1 (which it is not in this scenario).

Using optimistic locking correctly, you have to add the version as an additional hidden field (or into the SESSION for more safety). Otherwise you cannot verify the version is still the one being originally read from the database when Alice performed her GET request for the blog post. If this happens you might see lost updates you wanted to prevent with Optimistic Locking.

See the example code, The form (GET Request):

$post = $em->find('BlogPost', 123456);

echo '<input type="hidden" name="id" value="' .

$post->getId() . '"/>';

echo '<input type="hidden" name="version" value="' .

$post->getCurrentVersion() . '" />';

And the change headline action (POST Request):

$postId = (int) $\_GET['id'];

$postVersion = (int) $\_GET['version'];

$post = $em->find( 'BlogPost',

$postId, DoctrineDBALLockMode::OPTIMISTIC,

$postVersion

);

Wow — that was a lot of information to take in. However, don't worry if you don't completely understand everything. The more you work with Aggregates and Domain- Driven Design, the more you'll encounter moments when transactionality has to be considered in designing your Application.

To summarize, if you want to keep your data consistent, use transactions. However, be careful about overusing transactions or locking strategies because these can slow your Application down or make it unusable. If you want to have a really fast Application, optimistic concurrency can help you. Last but not least, some data can eventually be consistent. This means that we allow our data to not be consistent for a particular window of time. During that time, some inconsistencies are acceptable. Eventually, an asynchronous process will perform the final task to remove such inconsistencies.

## What Is an Aggregate?

聚合是保存其他实体和值对象的实体，有助于保持数据的一致性。 来自Vaughn Vernon的实施领域驱动设计:

聚合是精心制作的一致性边界，用于聚集实体和值对象.

另一本绝对值得购买和阅读的书是NoSQL Distilled：Pramod J. Sadalage和Martin Fowler撰写的“Polyglot持久性新兴世界简介”。 这本书说：

在域驱动设计中，聚合是我们希望作为一个单元处理的相关对象的集合。 特别是它是数据操纵和一致性管理的单位。 通常，我们喜欢使用原子操作更新聚合，并根据聚合与我们的数据存储进行通信.

### 马丁福勒说什么...

From <http://martinfowler.com/bliki/DDD_Aggregate.html>:

聚合是域驱动设计中的一种模式。 DDD聚合是可以视为单个单元的域对象群集。 一个示例可能是一个订单及其行项目，这些将是单独的对象，但将订单（连同其行项目）作为单个集合处理.

聚合将其组件对象之一作为聚合根。 任何来自聚合外部的引用只应该转到聚合根目录。 因此根可以确保总体的完整性.

聚合是您请求加载或保存整个聚合的数据存储传输的基本元素。 事务不应该跨越聚合边界。

DDD聚合有时会与集合类（列表，地图等）相混淆。 DDD聚合是领域概念（顺序，诊所访问，播放列表），而集合是通用的。 聚合通常包含多个集合以及简单的字段。 聚合这个术语是一个常用的术语，用于各种不同的上下文中（例如：UML），在这种情况下，它并不是指与DDD聚合相同的概念。

### 维基百科说什么...

From [https://en.wikipedia.org/wiki/Domain-driven\_design#Building\_blocks\_of\_D](https://en.wikipedia.org/wiki/Domain-driven_design#Building_blocks_of_DDD) [DD](https://en.wikipedia.org/wiki/Domain-driven_design#Building_blocks_of_DDD):

聚合：由根实体绑定在一起的对象的集合，也称为聚合根。 聚合根通过禁止外部对象保持对其成员的引用来保证聚合内所做更改的一致性.

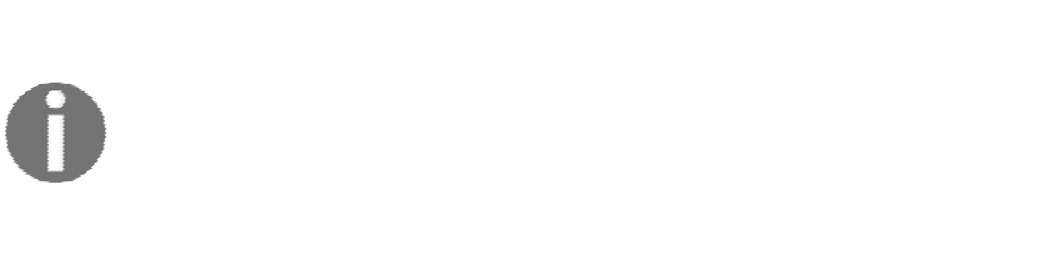
例如：驾驶汽车时，不必担心向前移动车轮，使发动机燃烧火花和燃料等。 你只是在开车。 在这种情况下，汽车是其他几个对象的集合，并作为所有其他系统的聚合根。

## 为什么要聚合?

狂热的读者可能会想知道这与聚合和聚合设计有什么关系。 实际上，这是一个很好的问题。 有一个直接的关系，所以让我们来探索它。 关系模型使用表来存储数据。 这些表由行组成，其中每行通常代表应用程序感兴趣概念的实例。 此外，每行可以指向同一数据库的其他表上的其他行，并且通过使用参照完整性可以保持此关系之间的一致性。 这个模型很好， 但是，它缺少一个非常基本的词语：客体词.

事实上，当我们谈论关系模型时，我们正在谈论表格，行和行之间的关系。 当我们谈论面向对象的模型时，我们主要讨论的是对象的组合。 所以每次我们取数据 - 一组行

- 从关系数据库中，我们运行一个翻译流程，负责构建我们可以使用的内存表示。 这同样适用于相反的方向。 每次我们需要在数据库中存储一个对象时，我们都应该运行另一个转换过程来将该对象转换为给定的一组行或表。 这种从对象到行或表的翻译意味着您可以对数据库运行不同的查询。 因此，如果没有使用任何特定的工具（如交易），就不可能保证数据一直保持不变。 这个问题就是所谓的阻抗不匹配.



**Impedance Mismatch**

The object-relational impedance mismatch is a set of conceptual and technical difficulties that are often encountered when a **relational database management system** (**RDBMS**) is being used by a program written in an object-oriented programming language or style, particularly when objects or class definitions are mapped in a straightforward way to database tables or relational schemata.

Extracted from [Wikipedia](https://en.wikipedia.org/wiki/Object-relational_impedance_mismatch)

The impedance mismatch [is not an easy problem to solve](http://martinfowler.com/bliki/OrmHate.html), so we highly discourage trying to solve it on your own. It would be a huge undertaking, and it's simply not worth the effort. Luckily, there are some libraries out there that take care of this translation process. They're commonly known as Object-Relational Mappers (which we've discussed in earlier chapters) and their primary concern is to ease the process of translating from the Relational Model to the Object-Oriented Model, and vice versa.

This is an issue that also affects NoSQL persistence engines and not just databases. Most NoSQL engines use documents. An Entity is translated into a document representation such as JSON, XML, binary, and so on. and then persisted. The main difference with RDBMS databases is that if a main Entity (such as Order) has other related Entities (such as OrderLines), you can more easily design a single JSON document that will contain all the information. With this approach, with a single request to your NoSQL engine, you don't need transactions.

Nevertheless, if you're using NoSQL or RDBMS for fetching and persisting your Entities, you'll need one or more queries. In order to ensure data consistency, those queries or requests need to be executed as a single operation. Running as a single operation can guarantee that data will be consistent.

What does consistent mean? It means that all data persisted into our database must be compliant with all business rules, also known as invariants. An example of a business invariant could be how on GitHub, a user is able to have unlimited public repositories but no private repositories. However, if this user pays $12 per month, then they're able to have up to 10 private repositories.

Relational databases provide three main tools for helping us with data consistency: \* **Referential** integrity: Foreign keys, nullable checks, and so on. \* **Transactions**: Run multiple queries as a single operation. The problem with transactions is the same as that of branches and merges in your code repository. Keeping a branch has a performance cost (memory, CPU, storage, indexing, and so on.). If too many people (concurrency) are touching the same data, conflicts will occur and transaction commits will fail. \* **Locking**: Block rows or tables. Other queries around the same tables and rows must wait for the block to be removed. Locking has a negative impact on the performance of your application.

Suppose we have an e-commerce application we want to expand to other countries and regions, and suppose the release goes fairly well and sales increase. A pretty evident side effect of the release is that the database should be able to handle the additional load increase. As seen earlier, there are two scaling methods: up or out.

Scaling up means we improve the hardware infrastructure we have (For example: better CPU, more memory, better hard disks). Scaling out means adding more machines that will organize in a cluster for doing specific work. In this case, we could have a cluster of databases.

But relational databases aren't designed to scale horizontally, since we can't configure them to save one set of rows to a given machine and another set of rows to a different one.

Relational databases are easy to scale up, but **the Relational Model doesn't scale horizontally**.

In the NoSQL world, data consistency is a bit more difficult: transactions and referential integrity aren't generally supported, while locking is supported but generally not encouraged.

NoSQL databases aren't affected as drastically by the impedance mismatch. They match perfectly with Aggregate Design because they enable us to easily store and retrieve single units atomically. For example, when using a key-value store such as Redis, an Aggregate could be serialized and stored on a specific key. On a document-oriented store such as Elasticsearch, an Aggregate would be serialized into a JSON and persisted as a document. As mentioned before, the problem comes when multiple documents must be updated at once.

For that reason, when persisting any object with a single representation (one document, so no multiple queries needed), it's easy to distribute those single units across several machines, called nodes, which make up a cluster of NoSQL databases. It's common knowledge that these databases are easy to distribute, which means that the style of databases is easy to scale horizontally.

## A Bit of History

Around the beginning of the 21st century, companies such as Amazon and Google grew massively. In order to consolidate their growth, they used clustering techniques: not only did they have better servers, but they also relied on many more of them working together.

In a scenario such as this, deciding how to store your data is key. If you take an Entity and spread its information throughout multiple servers, in multiple nodes of a cluster, the effort needed to control transactions is high. The same thing applies if you want to fetch an Entity. So if you can design your Entity in a way that is persisted in the node of a cluster, it makes things much easier. That's one of the reasons why Aggregate Design is so important.

If you want to know more about the history of Aggregate Design outside of Domain-Driven Design, take a look at [NoSQL Distilled: A Brief Guide to the Emerging World of](https://www.amazon.com/NoSQL-Distilled-Emerging-Polyglot-Persistence/dp/0321826620) [Polyglot Persistence](https://www.amazon.com/NoSQL-Distilled-Emerging-Polyglot-Persistence/dp/0321826620).

## 骨料解剖

聚合是一个可能包含其他实体和值对象的实体。 父实体被称为根实体.

没有任何子实体或值对象的单个实体本身也是一个聚合。 这就是为什么在一些书中，使用术语聚合体而不是术语实体。 当我们在这里使用它们时，Entity和Aggregate意味着同样的事情.

聚合的主要目标是保持您的域模型一致。 聚合集中了大部分业务规则。 聚合在您的持久性机制中以原子方式持久化。 无论有多少个实体和值对象存在于根实体内部，它们都将作为一个单元以原子方式持久化。 我们来看一个例子.

考虑电子商务应用程序，网站等。 用户可以下订单，该订单有多行，用于定义购买什么产品，价格，数量和生产线总金额。 订单也有总金额，即所有行金额的总和。

如果您更新线路金额而不是订单金额，会发生什么情况？ 数据不一致。 为了解决这个问题，通过根实体对聚合内的任何实体或值对象进行修改。 我们合作过的大多数PHP开发人员都比较乐于构建对象，然后从客户端代码处理他们的关系，而不是在实体内部推动业务逻辑:

$order = ...

$orderLine = new OrderLine(

'Domain-Driven Design in PHP', 24.99

);

$order->addOrderLine($orderLine);

正如前面的代码示例所示，新手甚至普通开发人员通常先构建子对象，然后使用setter将它们与父对象关联起来。 考虑以下方法:

$order = ...

$orderLine = $order->addOrderLine( 'Domain-Driven Design in PHP', 24.99

);

Or, consider this approach:

$order = ...

$order->addOrderLine(

'Domain-Driven Design in PHP', 24.99

);

这些方法非常有趣，因为它们遵循两个软件设计原则：告诉 - 不要问及德米特法则.

According to [Martin Fowler](http://martinfowler.com/bliki/TellDontAsk.html):

Tell-Don't-Ask是一个原则，可以帮助人们记住面向对象是关于将数据与对数据进行操作的函数捆绑在一起的。 它提醒我们，与其向对象提供数据并根据这些数据采取行动，我们应该告诉对象要做什么。 这鼓励将行为转化为与数据一起使用的对象.

According to [Wikipedia](https://en.wikipedia.org/wiki/Law_of_Demeter):

The **Law of Demeter** (**LoD**) or principle of least knowledge is a design guideline for developing software, particularly object-oriented programs. In its general form, the LoD is a specific case of loose coupling...and can be succinctly summarized in each of the following ways:

Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.

Each unit should only talk to its friends; don't talk to strangers. Only talk to your immediate friends.

The fundamental notion is that a given object should assume as little as possible about the structure or properties of anything else (including its subcomponents), in accordance with the principle of "information hiding".

Let's continue with the order example. You've already learned how to run operations through the root Entity. Now let's update a product quantity of a line in an order. This increases the quantity, the line total amount, and the order amount. Great! Now it's time to persist the order with the changes.

If you're using MySQL, you can imagine that we'll need two UPDATE statements: one for the orders table, and another one for the order\_lines table. What could happen if these two queries aren't performed inside a transaction?

Let's assume that the UPDATE statement that updates the line order works properly. However, the UPDATE on the order total amount fails due to network connectivity issues. In such a scenario, you would end up with a data inconsistency in your Domain Model.

Transactions help you keep this consistency.

If you're using Elasticsearch, the situation is a bit different. You can map the order with a JSON document that holds order lines internally, so just a single request is needed.

However, if you decide to map the order with one JSON document and each of its order lines with another JSON document, you're in trouble, as Elasticsearch doesn't support transactions. Ouch!

An Aggregate is fetched and persisted using its own [Chapter 10](#_bookmark315), *Repositories*. If two Entities don't belong to the same Aggregate, both will have their own Repository. If a true business invariant exists and two Entities belong to the same Aggregate, you'll only have one Repository. This Repository will be the one for the root Entity.

What are the cons of Aggregates? The problem when dealing with transactions is the possibility of performance issues and operation errors. We'll explore this in depth soon.

## Aggregate Design Rules

在设计聚合时，要遵循一些规则和注意事项才能获得所有好处并使负面影响最小化。 如果你现在什么都不了解，别担心， 作为例子，我们将向您展示一个小应用程序，我们将引用您介绍的规则.

### 基于业务真不变量的设计聚合

首先，什么是不变量？ 不变式是在代码执行过程中必须是真实和一致的规则。 例如，堆栈是LIFO（Last In，First Out）数据结构，我们可以将项目推入并弹出项目。 我们也可以询问堆栈中有多少物品; 这就是所谓的堆栈大小。 考虑一个纯粹的PHP实现，而不使用任何特定的PHP数组函数，如array\_pop:

class Stack

{

private $data;

public function construct()

{

$this->data = [];

}

public function push($value)

{

$this->data[] = $value;

}

public function size()

{

$size = 0;

for ($i = 0; $i < count($this->data); $i++) {

$size++;

}

return $size;

}

/\*\*

\* @return mixed

\*/

public function pop()

{

$topIndex = $this->size() - 1;

$top = $this->data[$topIndex]; unset($this->data[$topIndex]); return $top;

}

}

考虑以前的大小方法实现。 这远非完美，但它有效。 但是，由于它是在上面的代码中实现的，因此这是一个CPU密集型和高成本的调用。 幸运的是，可以通过引入一个私有属性来跟踪内部数组中元素的数量，从而改进此方法:

class Stack

{

private $data; private $size;

public function construct()

{

$this->data = [];

$this->size = 0;

}

public function push($value)

{

$this->data[] = $value;

$this->size++;

}

public function size()

{

return $this->size;

}

/\*\*

\* @return mixed

\*/

public function pop()

{

$topIndex = $this->size--;

$top = $this->data[$topIndex]; unset($this->data[$topIndex]);

return $top;

}

}

通过这些更改，size方法现在是一个快速操作，因为它只是返回size字段的值。 为了达到这个目的，我们引入了一个名为size的新整数属性。 当创建一个新的堆栈时，大小的值为0，堆栈中没有元素。 当我们使用push方法将新元素添加到堆栈中时，我们也增加了size字段的值。 同样，当我们使用pop方法从堆栈中删除一个值时，我们减小了大小的值.

通过递增和减小大小的值，我们保持它与堆栈内的元素的实数一致。 在调用Stack类中的任何公共方法之前和之后，大小值是一致的。 结果，大小值总是等于堆栈中元素的数量。 这是一个不变的！ 我们可以把它写成$ this-> size === count（$ this-> data）.

一个真正的业务不变是一个业务规则，它必须始终是真实的，并且在一个总合中是一致的。 通过事务一致，我们的意思是更新聚合必须是原子操作。 包含在一个聚合内的所有数据必须以原子方式持久化。 如果我们不遵循这个规则，我们可以坚持代表非有效聚合的数据.

According to [Vaughn Vernon](https://www.amazon.com/Implementing-Domain-Driven-Design-Vaughn-Vernon/dp/0321834577):

设计合理的聚合是一种可以按照业务所需的任何方式修改的聚合，其中不变量在单个事务中完全一致。 在所有情况下，正确设计的Bounded Context只会修改每个事务的一个Aggregate实例。 更重要的是，我们不能在没有应用事务分析的情况下正确推理总体设计.

正如引言中所讨论的，在电子商务应用程序中，订单金额必须与该订单中每行的金额总和相匹配。 这是一个不变的或商业规则。 我们必须在相同的事务中将订单和订单行保留到数据库中。 这迫使我们使Order和OrderLine成为同一个Aggregate的一部分，其中Order是Aggregate Root。 由于Order是根，所有与OrderLines相关的操作都必须经过Order。 因此，不需要在Order之外实例化OrderLine对象，然后使用setter方法将OrderLines添加到Order中。 相反，我们必须在订单上使用工厂方法。

通过这种方法，我们有一个入口点来对Aggregate执行操作：订单。 这意味着没有机会调用某种方法来打破这样的规则。 每次通过订单添加或更新订单行时，订单金额都会在内部重新计算。 让所有操作都通过根来帮助我们保持聚合一致。 通过这种方式，打破任何不变是更困难的。

### Small Aggregates Vs. Big Aggregates

对于我们工作过的大多数网站和项目，几乎95％的聚合由单个根实体和一些价值对象组成。 没有其他实体需要在同一总计中。 所以在大多数情况下，没有真正的业务不变来保持一致.

小心一对多关系，这些关系不一定使两个实体成为一个聚合体，其中一个是根。 正如我们将看到的，关系可以通过引用实体身份来处理。

正如介绍中所解释的，聚合是一个事务边界。 边界越小，提交多个并发事务时出现冲突的机会就越少。 在设计聚合体时，你应该努力将它们设计得很小。 如果没有真正的保护范围，那意味着所有单个实体都自己形成一个聚合体。 这很好，因为这是实现最佳性能的最佳场景。 为什么？ 因为锁定问题和失败的事务问题被最小化.

如果您决定采用大型聚合，保持数据一致可能会更容易，但可能不切实际。 当具有大型聚合的应用在生产环境中运行时，当多个用户执行操作时，他们开始遇到问题。 使用乐观并发时，主要问题是事务性失败。 使用锁定时，问题是速度缓慢和超时.

我们来考虑一些激进的例子。 使用乐观并发时，想象整个域都是版本化的，任何实体上的每个操作都会为整个域创建一个新版本。 在这种情况下，如果两个用户在不同的实体上执行不同的操作，而这些操作根本无法关联，则由于版本不同，第二个请求会遇到事务故障。 另一方面，当使用悲观并发时，想象一下我们在每个操作上锁定数据库的场景。 这将阻止所有用户，直到锁定被释放。 这意味着许多请求将会等待，并且在某个时候可能会超时。 这两个示例都保持数据一致，但应用程序不能由多个用户使用。

最后但并非最不重要的一点是，在设计大型聚合时，由于它们可能拥有实体集合，因此考虑将这些集合加载到内存中会带来性能影响，这一点很重要。 即使使用可延迟加载集合的ORM（如Doctrine）（只在需要时才加载集合），如果集合太大，则无法放入内存。

### 通过身份引用其他实体

当两个实体不构成一个聚合但是相关时，使实体相互引用的最佳选择是使用恒等。 身份已在第4章Entitie中进行了解释.

考虑一个用户及其订单，并假设我们没有找到任何真正的不变量。 用户和订单不属于同一个汇总。 如果您需要知道哪个用户拥有特定的订单，则可以询问订单的UserId是什么。 UserId是一个保存用户身份的值对象。 我们通过使用其存储库UserRepository来获取整个用户。 此代码通常位于应用程序服务中。.

作为一般性解释，每个Aggregate都有自己的Repository。 如果您已经获取了特定的Aggregate，并且您需要获取另一个相关的Aggregate，那么您可以在Application Services或Domain Services中执行此操作。 应用程序服务将依靠存储库来获取所需的聚合.

从一个集合跳转到另一个集合通常称为遍历或导航您的域。 使用ORM，通过映射实体之间的所有关系很容易。 但是，这也是非常危险的，因为您可以轻松地结束在特定功能中运行的无数查询。 一般来说，你不应该这样做。 不要映射你的实体之间的所有关系，因为你可以。 相反，如果两个实体形成一个聚合，则只映射ORM中聚合内的实体之间的关系。 如果情况并非如此，您将使用存储库来获取引用的聚合.

### 修改每个事务和请求的一个聚合

Consider the following scenario: you make a request, it gets into your controller, and it intends to update two different Aggregates. Each Aggregate keeps the data consistent within that Aggregate. However, what would happen if the request goes well over the first Aggregate update but suddenly stops (server restarted, reloaded, out of memory, and so on.) and the second Aggregate isn't updated? Is that a data consistency issue? It may be.

Let's consider some solutions.

From Vaughn Vernon's [Implementing Domain-Driven Design](https://www.amazon.com/Implementing-Domain-Driven-Design-Vaughn-Vernon/dp/0321834577):

In a properly designed Bounded Context modifies only one Aggregate instance per transaction in all cases. What is more, we cannot correctly reason on Aggregate design without applying transactional analysis. Limiting modification to one Aggregate instance per transaction may sound overly strict. However, it is a rule of thumb and should be the goal in most cases. It addresses the very reason to use Aggregates.

If, in a single request, you need to update two Aggregates, it may just be that those two Aggregates are a single one and they need to both be updated in the same transaction. If not, you can wrap the entire request in a transaction, but we wouldn't recommend this as the main option because of the performance issues and the transaction errors involved.

If both updates on different Aggregates don't need to be wrapped into a transaction, this means we can assume some delay between one update and the other. In such a scenario, a more Domain-Driven Design approach is to use Domain Events. When doing so, the first Aggregate update will fire a Domain Event. That event will be persisted in the same transaction as the Aggregate update and then published into our message queue. Later, a worker will take the event from the queue and perform the second Aggregate update. Such an approach pushes for Eventual Consistency, reduces the size of the transaction boundaries, improves performance, and reduces transaction errors.

## 示例应用程序服务：用户和愿望

Now you know the basic rules for Aggregate Design.

The best way to learn about Aggregates is by seeing code. So let's consider the scenario of a web application where users can make wishes to be granted if something happens to them, similar to a will. For example, I would like to send an email to my wife explaining what to do with my GitHub account if I die in a horrible accident, or maybe I want to send an email telling her how much I loved her. The way to check that I'm still alive is to answer emails the platform sends to me. (If you want to know more about [this application](https://github.com/dddinphp/last-wishes), you can visit our [GitHub](https://github.com/dddinphp) account. So we have users and their wishes. Let's consider only one use case: "As a User, I want to make a Wish." How could we model this? Using good practices when designing Aggregates, let's try to push for small Aggregates. In this case, that means using two different Aggregates of one Entity each, User and Wish. For the relationship between them, we should use an identifier, such as UserId.

### 没有不变，两个聚合体

我们将在下面的章节中讨论应用程序服务，但现在，我们来检查一下创建一个愿望的不同方法。 第一种方法，特别是新手，可能与此类似:

class MakeWishService

{

private $wishRepository;

public function construct(WishRepository $wishRepository)

{

$this->wishRepository = $wishRepository;

}

public function execute(MakeWishRequest $request)

{

$userId = $request->userId();

$address = $request->address();

$content = $request->content();

$wish = new Wish(

$this->wishRepository->nextIdentity(), new UserId($userId),

$address,

$content

);

$this->wishRepository->add($wish);

}

}

这段代码可能允许尽可能达到最佳性能。 你几乎可以在幕后看到INSERT语句; 这种用例的最少操作次数是1次，这很好。 通过目前的实施，我们可以根据业务需求创建尽可能多的愿望，这也是很好的.

但是，可能存在一个潜在问题：我们可以为域中不存在的用户创建愿望。 无论我们用于持续聚合的技术如何，这都是一个问题。 即使我们正在使用内存中的实现，我们也可以在没有相应的用户的情况下创建一个Wish.

这是破坏的业务逻辑。 当然，这可以使用数据库中的外键修复，从wish（user\_id）到user（id），但是如果我们不使用外键数据库会发生什么？ 如果它是NoSQL数据库（如Redis或Elasticsearch）会发生什么情况?

如果我们想解决这个问题，以便相同的代码可以在不同的基础设施中正常工作，我们需要检查用户是否存在。 最简单的方法可能在同一个应用服务中:

class MakeWishService

{

// ...

public function execute(MakeWishRequest $request)

{

$userId = $request->userId();

$address = $request->address();

$content = $request->content();

$user = $this->userRepository->ofId(new UserId($userId)); if (null === $user) {

throw new UserDoesNotExistException();

}

$wish = new Wish(

$this->wishRepository->nextIdentity(),

$user->id(),

$address,

$content

);

$this->wishRepository->add($wish);

}

}

这可以工作，但在应用服务中执行检查存在问题：该检查在代理链中很高。 如果任何不是此应用服务的其他代码片段（例如域服务或另一个实体）想要为非现有用户创建愿望，则可以这样做。 看看下面的代码：

// Somewhere in a Domain Service or Entity

$nonExistingUserId = new UserId('non-existing-user-id');

$wish = new Wish(

$this->wishRepository->nextIdentity(),

$nonExistingUserId,

$address,

$content

);

如果您已经阅读过第9章“工厂”，那么您有解决方案。 工厂帮助我们保持业务不变，这正是我们在这里所需要的.

有一种隐含的不变说，如果没有一个有效的用户，我们不允许这样做。 让我们看看工厂如何帮助我们：

abstract class WishService

{

protected $userRepository; protected $wishRepository;

public function construct( UserRepository $userRepository, WishRepository $wishRepository

) {

$this->userRepository = $userRepository;

$this->wishRepository = $wishRepository;

}

protected function findUserOrFail($userId)

{

$user = $this->userRepository->ofId(new UserId($userId)); if (null === $user) {

throw new UserDoesNotExistException();

}

return $user;

}

protected function findWishOrFail($wishId)

{

$wish = $this->wishRepository->ofId(new WishId($wishId)); if (!$wish) {

throw new WishDoesNotExistException();

}

return $wish;

}

protected function checkIfUserOwnsWish(User $user, Wish $wish)

{

if (!$wish->userId()->equals($user->id())) { throw new \InvalidArgumentException(

'User is not authorized to update this wish'

);

}

}

}

class MakeWishService extends WishService

{

public function execute(MakeWishRequest $request)

{

$userId = $request->userId();

$address = $request->address();

$content = $request->content();

$user = $this->findUserOrFail($userId);

$wish = $user->makeWish(

$this->wishRepository->nextIdentity(),

$address,

$content

);

$this->wishRepository->add($wish);

}

}

正如你所看到的，用户做出愿望，我们的代码也是如此。 makeWish是建立愿望的工厂方法。 该方法返回来自所有者的用UserId构建的新Wish：

class User

{

// ...

/\*\*

\* @return Wish

\*/

public function makeWish(WishId $wishId, $address, $content)

{

return new Wish(

$wishId,

$this->id(),

$address,

$content

);

}

// ...

}

为什么我们要返回一个愿望，而不是像我们对教义所做的那样将新的愿望添加到内部集合中？ 总而言之，在这种情况下，用户和愿望不符合总计，因为没有真正的业务保持不变。 用户可以根据需要添加和删除尽可能多的愿望。 愿望和他们的用户可以在不同的交易中在数据库中独立更新，如果需要的话。

遵循之前解释过的关于总体设计的规则，我们应该瞄准小聚合，这就是结果。 每个实体都有自己的存储库。 在这种情况下，希望引用其拥有User的用户使用Identities - UserId。 获取用户的所有愿望可以通过WishRepository中的查找器完成，并且可以轻松分页，而不会有任何性能问题：

interface WishRepository

{

/\*\*

* @param WishId $wishId

\*

* @return Wish

\*/

public function ofId(WishId $wishId);

/\*\*

* @param UserId $userId

\*

* @return Wish[]

\*/

public function ofUserId(UserId $userId);

/\*\*

* @param Wish $wish

\*/

public function add(Wish $wish);

/\*\*

* @param Wish $wish

\*/

public function remove(Wish $wish);

/\*\*

* @return WishId

\*/

public function nextIdentity();

}

这种方法的一个有趣的方面是，我们不必在我们最喜欢的ORM中映射User和Wish之间的关系。 因为我们使用UserId从Wish中引用User，所以我们只需要Repositories。 让我们考虑一下如何使用Doctrine来映射这些实体：

Lw\Domain\Model\User\User: type: entity

id:

userId:

column: id type: UserId

table: user repositoryClass:

Lw\Infrastructure\Domain\Model\User\DoctrineUser\Repository fields:

email:

type: string password:

type: string

Lw\Domain\Model\Wish\Wish: type: entity

table: wish repositoryClass:

Lw\Infrastructure\Domain\Model\Wish\DoctrineWish\Repository

id:

wishId:

column: id type: WishId

fields:

address:

type: string content:

type: text userId:

type: UserId column: user\_id

No relation is defined. After making a new wish, let's write some code for updating an existing one:

class UpdateWishService extends WishService

{

public function execute(UpdateWishRequest $request)

{

$userId = $request->userId();

$wishId = $request->wishId();

$email = $request->email();

$content = $request->content();

$user = $this->findUserOrFail($userId);

$wish = $this->findWishOrFail($wishId);

$this->checkIfUserOwnsWish($user, $wish);

$wish->changeContent($content);

$wish->changeAddress($email);

}

}

由于用户和愿望不构成聚合，为了更新愿望，我们需要首先使用WishRepository检索它。 一些额外的检查确保只有拥有者可以更新愿望。 正如您可能已经看到的那样，$ wish已经是我们域中的现有实体，因此不需要使用Repository再次将其添加回来。 但是，为了使更改持久化，我们的ORM必须知道更新的信息，并在完成工作后清除对数据库的任何剩余更改。 别担心， 我们将在第11章“应用程序”中仔细看看。 为了完成这个例子，我们来看看如何删除一个愿望:

class RemoveWishService extends WishService

{

public function execute(RemoveWishRequest $request)

{

$userId = $request->userId();

$wishId = $request->wishId();

$user = $this->findUserOrFail($userId);

$wish = $this->findWishOrFail($wishId);

$this->checkIfUserOwnsWish($user, $wish);

$this->wishRepository->remove($wish);

}

}

正如您可能已经看到的那样，您可以重构代码的某些部分，例如构造函数和所有权检查，以便在两个应用程序服务中重用它们。 随意考虑你将如何做到这一点。 最后但并非最不重要的是，我们如何获得特定用户的所有愿望：

class ViewWishesService extends WishService

{

/\*\*

* @return Wish[]

\*/

public function execute(ViewWishesRequest $request)

{

$userId = $request->userId();

$wishId = $request->wishId();

$user = $this->findUserOrFail($userId);

$wish = $this->findWishOrFail($wishId);

$this->checkIfUserOwnsWish($user, $wish);

return $this->wishRepository->ofUserId($user->id());

}

}

This is quite straightforward. However, we'll go deeper into how to render and return information from Application Services in the corresponding chapter. For now, returning a collection of Wishes will do the job.

Let's sum up this non-Aggregate approach. We couldn't find any true business invariant to consider User and Wish as an Aggregate, which is why each of them is an Aggregate. User has its own Repository, UserRepository. Wish has its own Repository too, WishRepository. Each Wish holds a UserId reference to owner, User. Even so, we didn't require a transaction. That's the best scenario in terms of performance and scalability issues. However, life is not always so wonderful. Consider what could happen with a true business invariant.

### 每个用户不超过三个愿望

我们的应用程序是一个巨大的成功，现在是时候从中获得一些收益。 我们希望新用户有最多三个愿望可用。 作为一个用户，如果你想有更多的愿望，你将来可能不得不支付一个高级账户。 让我们看看我们如何改变我们的代码，以遵循关于最大愿望数的新业务规则（在这种情况下，不要考虑高级用户）.

暂时考虑下面的代码。 除了前面的部分关于将逻辑推入实体的内容之外，下面的代码可以工作吗？:

class MakeWishService

{

// ...

public function execute(MakeWishRequest $request)

{

$userId = $request->userId();

$address = $request->email();

$content = $request->content();

$count = $this->wishRepository->numberOfWishesByUserId( new UserId($userId)

);

if ($count >= 3) {

throw new MaxNumberOfWishesExceededException();

}

$wish = new Wish(

$this->wishRepository->nextIdentity(), new UserId($userId),

$address,

$content

);

$this->wishRepository->add($wish);

}

}

它看起来可以。 这很简单 - 可能太简单了。 在这里我们遇到了不同的问题。 首先是应用程序服务必须协调，但不应包含业务逻辑。 相反，一个更好的地方是将最多三个愿望的检查放入用户中，在那里我们可以更好地控制用户与愿望之间的关系。

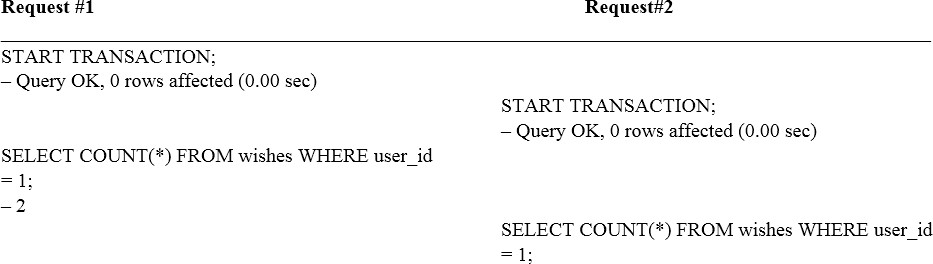
但是，对于这里显示的方法，代码似乎工作.

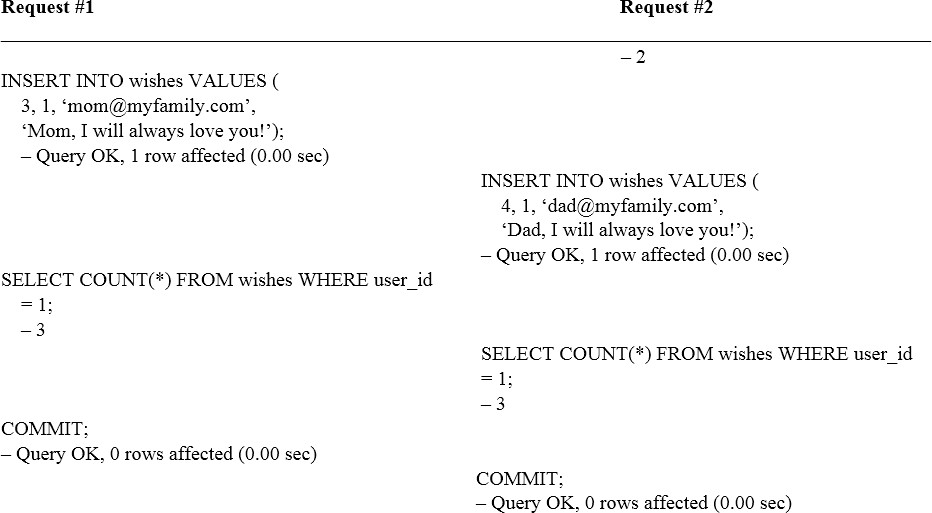
第二个问题是它在竞赛条件下不起作用。 暂时忘掉领域驱动设计。 这个代码在交通繁忙时有什么问题？ 想一下。 是否有可能违反用户的规则有三个以上的愿望？ 运行一些压力测试后，为什么你的QA会如此快乐？

您的质量检查会尝试执行两次希望功能，并以拥有两个愿望的用户结束。 这是正确的。 您的质量检查会继续测试该功能。 想象一下，他们在浏览器中打开两个选项卡，填写每个选项卡中的每个表单，并设法同时提交两个按钮。 突然，在两次请求后，用户在数据库中结束了四次希望。 这是错的！ 发生了什么？

认为是一个调试器，并考虑两个不同的请求获取if（$ count> 3）{line在同一时间。 这两个请求都会返回false，因为用户只有两个愿望。 所以这两个请求都会创建Wish，并且这两个请求都会将它添加到数据库中。 结果是一个用户的四个愿望。 这是不一致的!

我们知道你在想什么。 这是因为我们错过了将所有东西都放入交易中。 那么，想象一下id为1的用户已经有了两个愿望，所以剩下一个。 两个HTTP请求创建两个不同的愿望同时到达。 我们为每个请求启动一个数据库事务（我们将在第11章应用程序中回顾如何处理事务和请求）。 考虑以前的PHP代码将针对我们的数据库运行的所有查询。 请记住，如果您使用任何Visual Database Tool，则需要禁用任何自动提交标志:





How many wishes does the user with id 1 have? That's right, four. How did this happen? If you take this SQL block and execute it line by line in two different connections, you'll see how the wishes table is going to have four rows at the end of both executions. So it looks like it's not about protecting with a transaction. How could we fix this issue? As explained in the introduction, a concurrency control could help.

For those developers more advanced in database techniques, tweaking the isolation level could work. However, we consider that option too complex, as the problem could be solved with other approaches, and we're not always dealing with databases.

#### 悲观并发控制

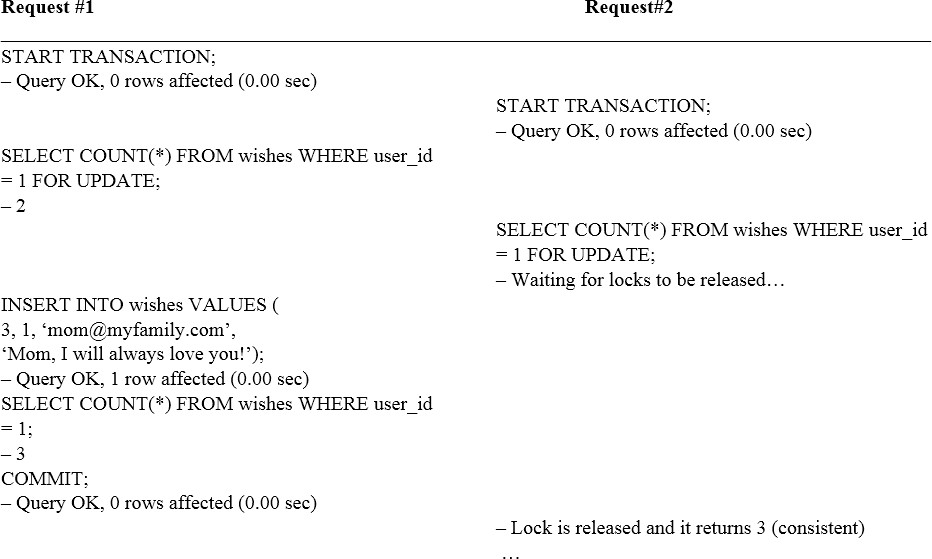
放置锁时有一个重要的考虑因素：尝试更新或查询相同数据的任何其他连接都将挂起，直到释放锁。 锁可以很容易地产生大多数性能问题。 例如，在MySQL中，对于放置锁有不同的选项：显式锁定表UN / LOCK表和锁定读取SELECT ... FOR UPDATE和SELECT ...在共享模式中锁定。

As we already shared above in the beginning, according to the book *Elasticsearch:* [The](https://github.com/elastic/elasticsearch-definitive-guide/blob/master/030_Data/40_Version_control.asciidoc) [Definitive Guide](https://github.com/elastic/elasticsearch-definitive-guide/blob/master/030_Data/40_Version_control.asciidoc) by Clinton Gormley and Zachary Tong:

Widely used by relational databases, this approach assumes that conflicting changes are likely to happen and so blocks access to a resource in order to prevent conflicts. A typical example is locking a row before reading its data, ensuring that only the thread that placed the lock is able to make changes to the data in that row.

We could [LOCK](http://dev.mysql.com/doc/refman/5.7/en/lock-tables.html) the table, but we consider such an approach complex and risky. When using locks, you have to be careful because you can end up with situations where two threads or requests are waiting for the other one to release the lock. This is what's called a deadlock.

Based on our experience, some developers use SELECT ... FOR UPDATE approaches. Let's see the same two request scenarios with this option:



As you can see, after the COMMIT of the first request, the count of the number of wishes of the second request is three. That's consistent, but the second request was waiting while the lock wasn't released. That means that in an environment with a lot of requests, it may generate performance issues. If the first request takes too much time to release the lock, the second request may fail due to a timeout:

ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction

The above looks like it's a valid option, but we need to be aware of the possible performance issues. Is there any other alternative?

#### 乐观并发控制

There's another alternative: not using locks at all. Consider adding a version attribute to our Aggregates. When we persist them, the persistence engine sets 1 as the version of the persisted Aggregate. Later, we retrieve the same Aggregate and perform some changes to it. We persist the Aggregate. The persistence engine checks that the version we have is the same as the one that's currently persisted, version 1. The persistence engine persists the Aggregate with the new state and updates its version to 2. If multiple requests retrieve the same Aggregate, make some changes to it, and then try to persist it, the first request will work, and the second will experiment and error. The last request just changed an outdated version, so the persistence engine throws an error. However, the second request can try to retrieve the Aggregate again, merge the new status, attempt to perform the changes, and then persist the Aggregate.

According to [Elasticsearch: The Definitive Guide](https://github.com/elastic/elasticsearch-definitive-guide/blob/master/030_Data/40_Version_control.asciidoc):

This approach assumes that conflicts are unlikely to happen and does not block operations from being attempted. However, if the underlying data has been modified between reading and writing, the update will fail. It is then up to the application to decide how it should resolve the conflict. For instance, it could reattempt the update, using the fresh data, or it could report the situation to the user.

This idea was covered before, but it bears repeating. If you try to apply Optimistic Concurrency to this scenario where we're checking maximum wishes in the Application Service, it's not going to work. Why? We're making a new wish, so two requests would create two different wishes. How can we make it work? Well, we need an object to centralize adding the wishes. We could apply the Optimistic Concurrency trick on that object, so it looks like we need a parent object that will hold wishes. Any ideas?

总而言之，在审查并发控制之后，有一种悲观的选择可行，但是对性能影响有一些担忧。 有一个乐观的选择，但我们需要找到一个父对象。 让我们考虑最后的MakeWishService，但做了一些修改:

class WishAggregateService

{

protected $userRepository;

public function construct(UserRepository $userRepository)

{

$this->userRepository = $userRepository;

}

protected function findUserOrFail($userId)

{

$user = $this->userRepository->ofId(new UserId($userId)); if (null === $user) {

throw new UserDoesNotExistException();

}

return $user;

}

}

class MakeWishService extends WishAggregateService

{

public function execute(MakeWishRequest $request)

{

$userId = $request->userId();

$address = $request->address();

$content = $request->content();

$user = $this->findUserOrFail($userId);

$user->makeWish($address, $content);

// Uncomment if your ORM can not flush

// the changes at the end of the request

// $this->userRepository->add($user);

}

}

我们没有通过WishId，因为它应该是用户内部的东西。 makeWish也不会返回Wish; 它在内部存储新的愿望。 执行应用程序服务之后，我们的ORM将把$用户执行的更改刷新到数据库。 取决于我们的ORM有多好，我们可能需要使用Repository再次显式地添加我们的用户实体。 需要对User类进行哪些更改？ 首先，应该有一个可以保存用户内所有愿望的集合:

class User

{

// ...

/\*\*

* @var ArrayCollection

\*/

protected $wishes;

public function construct(UserId $userId, $email, $password)

{

// ...

$this->wishes = new ArrayCollection();

// ...

}

// ...

}

The wishes property must be initialized in the User constructor. We could use a plain PHP array, but we've chosen to use an ArrayCollection. ArrayCollection is a PHP array with some extra features provided by the Doctrine Common Library, and it can be used separate from the ORM. We know that some of you may think that this could be a boundary leaking and that no references to any infrastructure should be here, but we really believe that's not the case. In fact, the same code works using plain PHP arrays. Let's see how the makeWish implementation is affected:

class User

{

// ...

/\*\*

* @return void

\*/

public function makeWish($address, $content)

{

if (count($this->wishes) >= 3) {

throw new MaxNumberOfWishesExceededException();

}

$this->wishes[] = new Wish( new WishId,

$this->id(),

$address,

$content

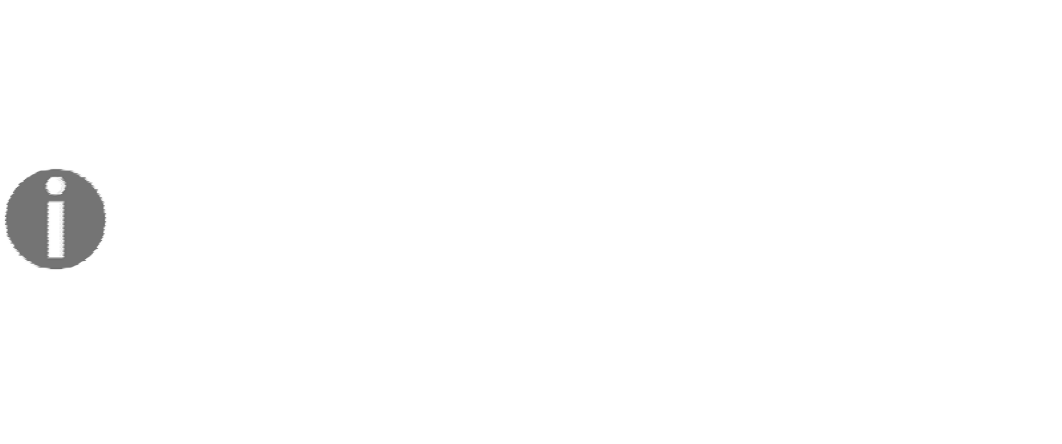
);

}

// ...

}

So far, so good. Now, it's time to review how the rest of the operations are implemented.



**Pushing for Eventual Consistency**

It looks like the business doesn't want a user to have more than three wishes. That's going to force us to consider User as the root Aggregate with Wish inside. This will affect our design, performance, scalability issues, and so on. Consider what would happen if we could just let users add as many wishes as they wanted, beyond the limit. We could check who is exceeding that limit and let them know they need to purchase a premium account. Allowing a user to go over the limit and warning them by telephone afterward would be a really nice commercial strategy. That might even allow the developers on your team to avoid designing User and Wish as part of the same Aggregate, with User as its root. You've already seen the benefits of not designing a single Aggregate: maximum performance.

class UpdateWishService extends WishAggregateService

{

public function execute(UpdateWishRequest $request)

{

$userId = $request->userId();

$wishId = $request->wishId();

$email = $request->email();

$content = $request->content();

$user = $this->findUserOrFail($userId);

$user->updateWish(new WishId($wishId), $email, $content);

}

}

Because User and Wish now form an Aggregate, the Wish to be updated is no longer retrieved using the WishRepository. We fetch the user using the UserRepository. The operation of updating a Wish is performed via the root Entity, which is the User in this case. The WishId is necessary in order to identify which Wish we want to update:

class User

{

// ...

public function updateWish(WishId $wishId, $email, $content)

{

foreach ($this->wishes as $wish) {

if ($wish->id()->equals($wishId)) {

$wish->changeContent($content);

$wish->changeAddress($address); break;

}

}

}

}

Depending on the features of your framework, this task may or may not be cheaper to perform. Iterating through all the wishes could mean making too many queries, or even worse, fetching too many rows, which will create a huge impact on memory. In fact, that's one of the main problems of having big Aggregates. So let's consider how to remove a Wish:

class RemoveWishService extends WishAggregateService

{

public function execute(RemoveWishRequest $request)

{

$userId = $request->userId();

$wishId = $request->wishId();

$user = $this->findUserOrFail($userId);

$user->removeWish($wishId):

}

}

As seen before, WishRepository is no longer necessary. We fetch the User using its Repository and perform the action of removing a Wish. In order to remove a Wish, we need to remove it from the inner collection. An option would be iterating through all the elements and matching the one with the same WishId:

class User

{

// ...

public function removeWish(WishId $wishId)

{

foreach ($this->wishes as $k => $wish) { if ($wish->id()->equals($wishId)) {

unset($this->wishes[$k]); break;

}

}

}

// ...

}

That's probably the most ORM-agnostic code possible. However, behind the scenes, Doctrine is fetching all the wishes and iterating through all of them. A more specific approach to fetch only the Entity needed that isn't so ORM agnostic would be the following: Doctrine mapping must also be updated in order to make all the magic work as expected.

While the Wish mapping remains the same, the User mapping has the new oneToMany

unidirectional relationship:

Lw\Domain\Model\Wish\Wish: type: entity

table: lw\_wish repositoryClass:

Lw\Infrastructure\Domain\Model\Wish\DoctrineWish\Repository

id:

wishId:

column: id type: WishId

fields:

address:

type: string content:

type: text userId:

type: UserId column: user\_id

Lw\Domain\Model\User\User: type: entity

id: userId:

column: id type: UserId

table: user repositoryClass:

Lw\Infrastructure\Domain\Model\User\DoctrineUser\Repository fields:

email:

type: string password:

type: string manyToMany:

wishes:

orphanRemoval: true cascade: ["all"]

targetEntity: Lw\Domain\Model\Wish\Wish joinTable:

name: user\_wishes joinColumns:

user\_id:

referencedColumnName: id inverseJoinColumns:

wish\_id:

referencedColumnName: id unique: true

In the code above, there are two important configurations: orphanRemoval and cascade. According to the Doctrine 2 ORM Documentation on [orphan removal](http://doctrine-orm.readthedocs.io/projects/doctrine-orm/en/latest/reference/working-with-associations.html#orphan-removal) and [transitive](http://doctrine-orm.readthedocs.io/projects/doctrine-orm/en/latest/reference/working-with-associations.html#transitive-persistence-cascade-operations) [persistence / cascade operations](http://doctrine-orm.readthedocs.io/projects/doctrine-orm/en/latest/reference/working-with-associations.html#transitive-persistence-cascade-operations):

If an Entity of type A contains references to privately owned Entities B then if the reference from A to B is removed the entity B should also be removed, because it is not used anymore. OrphanRemoval works with one-to-one, one-to-many and many-to-many associations. When using the orphanRemoval=true option Doctrine makes the assumption that the entities are privately owned and will NOT be reused by other entities. If you neglect this assumption your entities will get deleted by Doctrine even if you assigned the orphaned entity to another one.

Persisting, removing, detaching, refreshing and merging individual entities can become pretty cumbersome, especially when a highly interweaved object graph is involved. Therefore Doctrine 2 provides a mechanism for transitive persistence through cascading of these operations. Each association to another entity or a collection of entities can be configured to automatically cascade certain operations. By default, no operations are cascaded.

For more information, please take a closer look at the Doctrine 2 ORM 2 Documentation on

[working with associations](http://doctrine-orm.readthedocs.io/projects/doctrine-orm/en/latest/reference/working-with-associations.html).

Finally, let's see how we can get the wishes from a user:

class ViewWishesService extends WishService

{

/\*\*

* @return Wish[]

\*/

public function execute(ViewWishesRequest $request)

{

return $this

->findUserOrFail($request->userId())

->wishes();

}

}

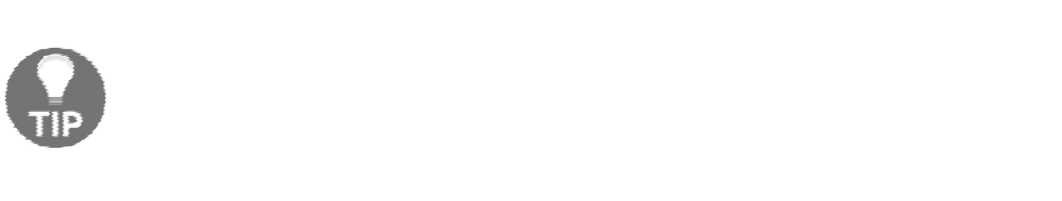
As mentioned before, especially in this scenario using Aggregates, returning a collection of Wishes is not the best solution. You should never return Domain Entities, as this will prevent code outside of your Application Services — such as Controllers or your UI — from unexpectedly modifying them. With Aggregates, it makes even more sense. Entities that aren't root — the ones that belong to the Aggregate but aren't root

— should appear private to others outside.

We'll go deeper into this in the [Chapter 11](#_bookmark358), *Application*. For now, to summarize, you have different options:

The Application Service returns a DTO build accessing Aggregates information. The Application Service returns a DTO returned by the Aggregate.

The Application Service uses an Output dependency where it writes the Aggregate. Such an Output dependency will handle the transformation to a DTO or other format.



Render the Number of Wishes As an exercise, consider that we want to render the number of wishes a user has made on their account page. How would you implement this, considering User and Wish don't form an Aggregate? How would you implement it if User and Wish did form an Aggregate? Consider how Eventual Consistency could help in your solutions.

## Transactions

We haven't shown beginTransaction, commit, or rollback in any of the examples. This is because transactions are handled at Application Service level. Don't worry for now; you'll find more details about this in [Chapter 11](#_bookmark358), *Application*.

## Wrap Up

Aggregates are all about persistence and transactions. In fact, you can't design Aggregates without thinking about how they're going to be persisted. The basic rules to design proper Aggregates are: make them small, find true business invariants, push for eventual consistency using Domain Events, reference other Aggregates by Identity, and modify one Aggregate per request. Review how the code changes if two Entities form a single Aggregate or not. Use factories to enrich your Entities. Finally, relax. In most of the PHP applications we've seen, only five percent of the Entities were Aggregates formed by two Entities or more. Discuss with your workmates when designing and implementing Aggregates.

# Factories

****

Factories are a powerful abstraction. They help decouple the client from the details of how to interact with the Domain. The client doesn't need to know how to build complex objects and Aggregates, so you can use Factories to create whole Aggregates, thereby enforcing their invariants.

## Factory Method on Aggregate Root

The [Factory Method](https://en.wikipedia.org/wiki/Factory_method_pattern) pattern, as defined in the classic, [Gang of Four](http://wiki.c2.com/?GangOfFour), is a creational pattern that:

Defines an interface for creating an object, but leaves the choice of its type to the subclasses, creation being deferred at run-time.

Adding a Factory Method in the Aggregate Root hides the internal implementation details of creating Aggregates from any external client. This also moves the responsibility for the integrity of the Aggregate back to the root.

In a Domain Model where we have a User Entity and a Wish Entity, the User acts as the Aggregate root. There's no Wish without User. The User Entity should manage its Aggregates.

The way to move the control of Wish back to the User Entity is by placing a Factory method in the Aggregate root:

class User

{

// ...

public function makeWish(WishId $wishId, $email, $content)

{

$wish = new WishEmail(

$wishId,

$this->id(),

$email,

$content

);

DomainEventPublisher::instance()->publish( new WishMade($wishId)

);

return $wish;

}

}

The client doesn't need to know the internal details of how the Aggregate Root handles the creation logic:

$wish = $aUser->makeWish(

$wishRepository->nextIdentity(), 'user@example.com',

'I want to be free!'

);

### Forcing Invariants

Factory Methods in the Aggregate Root are also a good place for invariants.

In a Domain Model with Forum and Post Entities, where Post is an aggregated part of the Aggregate Root Forum, publishing a Post could look something like this:

class Forum

{

// ...

public function publishPost(PostId $postId, $content)

{

$post = new Post($this->id, $postId, $content);

DomainEventPublisher::instance()->publish( new PostPublished($postId)

);

return $post;

}

}

After talking with a Domain Expert, we came to the conclusion that a Post shouldn't be published when the Forum is closed. This is an invariant, and we could force it directly on Post creation, thereby preventing an inconsistent Domain state:

class Forum

{

// ...

public function publishPost(PostId $postId, $content)

{

if ($this->isClosed()) {

throw new ForumClosedException();

}

$post = new Post($this->id, $postId, $content); DomainEventPublisher::instance()->publish(

new PostPublished($postId)

);

return $post;

}

}

## Factory on Service

Decoupling creation logic also comes in handy in our Services.

### Building Specifications

Using Specifications in our Services might be the best example to illustrate how to use Factories within our Services.

Consider the following Service example. Given a request from the outside world, we want to build a feed based on the latest Posts added to the system:

namespace Application\Service;

use Domain\Model\Post;

use Domain\Model\PostRepository;

class LatestPostsFeedService

{

private $postRepository;

public function construct(PostRepository $postRepository)

{

$this->postRepository = $postRepository;

}

/\*\*

\* @param LatestPostsFeedRequest $request

\*/

public function execute($request)

{

$posts = $this->postRepository->latestPosts($request->since);

return array\_map(function(Post $post) { return [

'id' => $post->id()->id(),

'content' => $post->body()->content(), 'created\_at' => $post-> createdAt()

];

}, $posts);

}

}

Finder methods in Repositories like latestPosts have some limitations, as they keep adding complexity to our Repositories indefinitely. As we discuss in the [Chapter 10](#_bookmark315), *Repositories* Specifications are a better approach.

Lucky for us, we have a nice query method in our PostRepository that works with Specifications:

class LatestPostsFeedService

{

// ...

public function execute($request)

{

$posts = $this->postRepository->query($specification);

}

}

Using a concrete implementation for the Specification is a bad idea:

class LatestPostsFeedService

{

public function execute($request)

{

$posts = $this->postRepository->query(

new SqlLatestPostSpecification($request->since)

);

}

}

Coupling our high-level application Service with a low-level Specification implementation mixes layers and breaks the Separation of Concerns. In addition, this is a pretty bad way of coupling our Service to a concrete Infrastructure implementation. There's no way you could use this Service outside of the SQL persistence solution. What if we want to test our Service with an in-memory implementation?

The solution to this problem is to decouple Specification creation from the Service itself by using the [Abstract Factory pattern](https://en.wikipedia.org/wiki/Abstract_factory_pattern). According to [OODesign.com](http://www.oodesign.com/abstract-factory-pattern.html):

Abstract Factory offers the interface for creating a family of related objects, without explicitly specifying their classes.

As we might have multiple Specification implementations, we first need to create an interface for the Factory:

namespace Domain\Model;

interface PostSpecificationFactory

{

public function createLatestPosts(DateTimeImmutable $since);

}

Then we need to create Factories for each PostRepository implementation. As an example, a Factory for the in-memory PostRepository implementation could look like this:

namespace Infrastructure\Persistence\InMemory;

use Domain\Model\PostSpecificationFactory; class InMemoryPostSpecificationFactory

implements PostSpecificationFactory

{

public function createLatestPosts(DateTimeImmutable $since)

{

return new InMemoryLatestPostSpecification($since);

}

}

Once we have a centralized place for the creation logic, it's easy to decouple it from the Service:

class LatestPostsFeedService

{

private $postRepository;

private $postSpecificationFactory;

public function construct( PostRepository $postRepository,

PostSpecificationFactory $postSpecificationFactory

) {

$this->postRepository = $postRepository;

$this->postSpecificationFactory = $postSpecificationFactory;

}

public function execute($request)

{

$posts = $this->postRepository->query(

$this->postSpecificationFactory->createLatestPosts(

$request->since

)

);

}

}

Now, unit testing our Service through an in-memory PostRepository implementation is pretty easy:

namespace Application\Service;

use Domain\Model\Body; use Domain\Model\Post; use Domain\Model\PostId;

use Infrastructure\Persistence\InMemory\InMemoryPostRepositor;

class LatestPostsFeedServiceTest extends PHPUnit\_Framework\_TestCase

{

/\*\*

\* @var \Infrastructure\Persistence\InMemory\InMemoryPostRepository

\*/

private $postRepository;

/\*\*

\* @var LatestPostsFeedService

\*/

private $latestPostsFeedService;

public function setUp()

{

$this->latestPostsFeedService = new LatestPostsFeedService(

$this->postRepository = new InMemoryPostRepository()

);

}

/\*\*

\* @test

\*/

public function shouldBuildAFeedFromLatestPosts()

{

$this->addPost(1, 'first', '-2 hours');

$this->addPost(2, 'second', '-3 hours');

$this->addPost(3, 'third', '-5 hours');

$feed = $this->latestPostsFeedService->execute( new LatestPostsFeedRequest(

new \DateTimeImmutable('-4 hours')

)

);

$this->assertFeedContains([

['id' => 1, 'content' => 'first'],

['id' => 2, 'content' => 'second']

], $feed);

}

private function addPost($id, $content, $createdAt)

{

$this->postRepository->add(new Post( new PostId($id),

new Body($content),

new \DateTimeImmutable($createdAt)

));

}

private function assertFeedContains($expected, $feed)

{

foreach ($expected as $index => $contents) {

$this->assertArraySubset($contents, $feed[$index]);

$this->assertNotNull($feed[$index]['created\_at']);

}

}

}

### Building Aggregates

Entities are agnostic to the persistence mechanism. You don't want to couple and pollute your Entities with persistence details. Take a look at the next Application Service:

class SignUpUserService

{

private $userRepository;

public function construct(UserRepository $userRepository)

{

$this->userRepository = $userRepository;

}

/\*\*

\* @param SignUpUserRequest $request

\*/

public function execute( $request)

{

$email = $request->email();

$password = $request->password();

$user = $this->userRepository->userOfEmail($email); if (null !== $user) {

throw new UserAlreadyExistsException();

}

$this->userRepository->persist(new User(

$this->userRepository->nextIdentity(),

$email,

$password

));

return $user;

}

}

Imagine a User Entity like the following one:

class User

{

private $userId; private $email; private $password;

public function construct(UserId $userId, $email, $password)

{

// ...

}

// ...

}

Imagine we want to use Doctrine as our Infrastructure persistence mechanism. Doctrine requires having an id as a plain string instance variable in order to work properly. In our Entity, $userId is a UserId Value Object. Adding an additional id to our User Entity just because of Doctrine would couple our persistence mechanism with our Domain Model.

We saw in the [Chapter 4](#_bookmark101), *Entities* that we could solve this problem with a Surrogate ID by creating a wrapper around our User Entity in the Infrastructure layer:

class DoctrineUser extends User

{

private $surrogateUserId;

public function construct(UserId $userId, $email, $password)

{

parent:: construct($userId, $email, $password);

$this->surrogateUserId = $userId->id();

}

}

As creating the DoctrineUser in our Application Service would again couple the persistence layer with our Domain, we need to decouple the creation logic out of the Service with an Abstract Factory.

We could do this by creating an interface in our Domain:

interface UserFactory

{

public function build(UserId $userId, $email, $password);

}

Then, we place the implementation of it inside our Infrastructure layer:

class DoctrineUserFactory implements UserFactory

{

public function build(UserId $userId, $email, $password)

{

return new DoctrineUser($userId, $email, $password);

}

}

Once decoupled, we only need to inject the Factory into our Application Service:

class SignUpUserService

{

private $userRepository; private $userFactory;

public function construct( UserRepository $userRepository, UserFactory $userFactory

) {

$this->userRepository = $userRepository;

$this->userFactory = $userFactory;

}

/\*\*

\* @param SignUpUserRequest $request

\*/

public function execute($request)

{

// ...

$user = $this->userFactory->build(

$this->userRepository->nextIdentity(),

$email,

$password

);

$this->userRepository->persist($user); return $user;

}

}

## Testing Factories

You'll see a common pattern while writing your tests. This is because building Entities and complex Aggregates can be a very tedious and repetitive process. Inevitably, complexity and duplication will start creeping into your test suite. Consider the following Entity:

class Author

{

private $username; private $email ; private $fullName;

public function construct( Username $aUsername, FullName $aFullName, Email $anEmail

) {

$this->username = $aUsername;

$this->email = $anEmail ;

$this->fullName = $aFullName;

}

// ...

}

Somewhere in your system, you'll end up with a test looking like this:

class MyTest extends PHPUnit\_Framework\_TestCase

{

/\*\*

\* @test

\*/

public function itDoesSomething()

{

$author = new Author(

new Username('johndoe'),

new FullName('John', 'Doe' ), new Email('john@doe.com' )

);

//do something with author

}

}

Services inside boundaries share concepts like Entities, Aggregates, and Value Objects. Imagine the clutter of repeating the same building logic over and over across your tests. As we'll see, extracting the building logic out of tests comes in handy and prevents duplication.

### Object Mother

An [Object Mother](https://martinfowler.com/bliki/ObjectMother.html) is a catchy name for a Factory that creates fixed fixtures for your tests. Similar to the previous example, we could extract the duplicated logic to an Object Mother so it could be reused across tests:

class AuthorObjectMother

{

public static function createOne()

{

return new Author(

new Username('johndoe'),

new FullName('John', 'Doe'), new [Email('john@doe.com](mailto:Email(%27john@doe.com) )

);

}

}

class MyTest extends PHPUnit\_Framework\_TestCase

{

/\*\*

\* @test

\*/

public function itDoesSomething()

{

$author = AuthorObjectMother::createOne();

}

}

You'll notice that the more tests and situations you have, the more methods the Factory will have.

As Object Mothers aren't very flexible, they tend to grow in complexity quickly. Luckily, there's a more flexible alternative for your tests.

### Test Data Builder

Test Data Builders are just normal Builders with default values used exclusively in your test suites so that you don't have to specify irrelevant parameters on specific test cases:

class AuthorBuilder

{

private $username; private $email ; private $fullName;

private function construct()

{

$this->username = new Username('johndoe');

$this->email = new Email('john@doe.com');

$this->fullName = new FullName('John', 'Doe');

}

public static function anAuthor()

{

return new self();

}

public function withFullName(FullName $aFullName)

{

$this->fullName = $aFullName;

return $this;

}

public function withUsername(Username $aUsername)

{

$this->username = $aUsername;

return $this;

}

public function withEmail(Email $anEmail)

{

$this->email = $anEmail ;

return $this;

}

public function build()

{

return new Author($this->username, $this->fullName, $this->email);

}

}

class MyTest extends PHPUnit\_Framework\_TestCase

{

/\*\*

\* @test

\*/

public function itDoesSomething()

{

$author = AuthorBuilder::anAuthor()

->withEmail(new Email('other@email.com'))

->build();

}

}

We could even combine Test Data Builders to build more complicated Aggregates, like a

Post:

class Post {

private $id; private $author; private $body; private $createdAt;

public function construct(

PostId $anId, Author $anAuthor, Body $aBody

) {

$this->id = $anId;

$this->author = $anAuthor;

$this->body = $aBody;

$this->createdAt = new DateTimeImmutable();

}

}

Let's see the corresponding Test Data Builder for our Post. We could reuse the

AuthorBuilder for building a default Author:

class PostBuilder

{

private $postId; private $author; private $body;

private function construct()

{

$this->postId = new PostId();

$this->author = AuthorBuilder::anAuthor()->build();

$this->body = new Body('Post body');

}

public static function aPost()

{

return new self();

}

public function withAuthor(Author $anAuthor)

{

$this->author = $anAuthor;

return $this;

}

public function withPostId(PostId $aPostId)

{

$this->postId = $aPostId;

return $this;

}

public function withBody(Body $body)

{

$this->body = $body;

return $this;

}

public function build()

{

return new Post($this->postId, $this->author, $this->body);

}

}

This solution is now flexible enough to cover any test case, including the possibility of building inner Entities:

class MyTest extends PHPUnit\_Framework\_TestCase

{

/\*\*

\* @test

\*/

public function itDoesSomething()

{

$post = PostBuilder::aPost()

->withAuthor(AuthorBuilder::anAuthor()

->withUsername(new Username('other'))

->build())

->withBody(new Body('Another body'))

->build();

//do something with the post

}

}

## Wrap-Up

Factories are a powerful tool for decoupling construction logic from our business logic. The Factory Method pattern not only helps by moving creation responsibility to the Aggregate Root, but it could also force Domain invariants. Using the Abstract Factory pattern in our Services allows us to separate our Domain logic from Infrastructure creation details. A common use case is that of Specifications and their respective persistence implementations. We've seen that Factories come in handy on our test suites too. While we could extract building logic into Object Mother Factories, Test Data Builders provide more flexibility for our tests.

# Repositories

****

In order to interact with a Domain object, you need to hold a reference to it. One way of achieving this is by creation. Alternatively, you can traverse an association. In Object- Oriented programs, objects have links (references) to other objects, which makes them easily traversable, thereby contributing to the expressive power of our models. But here's the catch: you need a mechanism to retrieve the first object, the Aggregate Root.

Repositories act as storage locations, where a retrieved object is returned in the exact same state it was persisted in. In Domain-Driven Design, every Aggregate type typically has a unique associated Repository, which is used for its persistence and fetching needs.

However, in the case where it's required to share an Aggregate object hierarchy, the types might share a Repository.

Once you've successfully retrieved the Aggregate from the Repository, every change you make is persisted, which removes the need to go back to the Repository.

## Definition

Martin Fowler defines a Repository as:

The mechanism between the domain and data mapping layers, acting like an in- memory domain object collection. Client objects construct query specifications declaratively and submit them to Repository for satisfaction. Objects can be added to and removed from the Repository, as they can from a simple collection of objects, and the mapping code encapsulated by the Repository will carry out the appropriate operations behind the scenes. Conceptually, a Repository encapsulates the set of objects persisted in a data store and the operations performed over them, providing a more object-oriented view of the persistence layer. Repository also supports the objective of achieving a clean separation and one-way dependency between the domain and [data mapping layers](http://martinfowler.com/eaaCatalog/repository.html).

## Repositories Are Not DAOs

**Data Access Objects** (**DAOs**) are a common pattern for persisting Domain objects into the database. It's easy to confuse the DAO pattern with a Repository. The significant difference is that Repositories represent collections, while DAOs are closer to the database and are often far more table-centric. Typically, a DAO would contain CRUD methods for a particular Domain object. Let's see how a common interface for a DAO might look:

interface UserDAO

{

/\*\*

* @param string $username
* @return User

\*/

public function get($username); public function create(User $user); public function update(User $user);

/\*\*

* @param string $username

\*/

public function delete($username);

}

A DAO interface could have multiple implementations, which could range from using ORM constructions to using plain SQL queries. The main problem with DAOs is that their responsibilities are not clearly defined. DAOs are usually perceived as gateways to the database, so it's relatively easy to greatly decrease cohesion with many specific methods in order to query the database:

interface BloatUserDAO

{

public function get($username); public function create(User $user); public function update(User $user); public function delete($username);

public function getUserByLastName($lastName); public function getUserByEmail($email);

public function updateEmailAddress($username, $email);

public function updateLastName($username, $lastName);

}

As you can see, the more we add new methods to implement, the harder it becomes to unit test the DAO, and it becomes increasingly coupled to the User object. This problem will grow over time, with many other contributors collaborating in making the Big Ball of Mud even bigger.

## Collection-Oriented Repositories

Repositories mimic a collection by implementing their common interface characteristics. As a collection, a Repository shouldn't leak any intentions of persistence behavior, such as the notion of saving to a store.

The underlying persistence mechanism has to support this need. You shouldn't be required to handle changes to the objects over their lifetime. The collection references the most recent changes to the object, meaning that upon each access, you get the latest object state.

Repositories implement a concrete collection type, the Set. A Set is a data structure with an invariant that doesn't contain duplicate entries. If you try to add an element that's already present to a Set, it won't be added. This is useful in our use case, as each Aggregate has a unique identity that's associated with the Root Entity.

Consider, for example, that we have the following Domain Model:

namespace Domain\Model;

class Post

{

const EXPIRE\_EDIT\_TIME = 120; // seconds

private $id; private $body; private $createdAt;

public function construct(PostId $anId, Body $aBody)

{

$this->id = $anId;

$this->body = $aBody;

$this->createdAt = new \DateTimeImmutable();

}

public function editBody(Body $aNewBody)

{

if($this->editExpired()) {

throw new RuntimeException('Edit time expired');

}

$this->body = $aNewBody;

}

private function editExpired()

{

$expiringTime= $this->createdAt->getTimestamp() + self::EXPIRE\_EDIT\_TIME;

return $expiringTime < time();

}

public function id()

{

return $this->id;

}

public function body()

{

return $this->body;

}

public function createdAt()

{

return $this->createdAt;

}

}

class Body

{

const MIN\_LENGTH = 3; const MAX\_LENGTH = 250;

private $content;

public function construct($content)

{

$this->setContent(trim($content));

}

private function setContent($content)

{

$this->assertNotEmpty($content);

$this->assertFitsLength($content);

$this->content = $content;

}

private function assertNotEmpty($content)

{

if(empty($content)) {

throw new DomainException('Empty body');

}

}

private function assertFitsLength($content)

{

if(strlen($content) < self::MIN\_LENGTH) {

throw new DomainException('Body is too short');

}

if(strlen($content) > self::MAX\_LENGTH) {

throw new DomainException('Body is too long');

}

}

public function content()

{

return $this->content;

}

}

class PostId

{

private $id;

public function construct($id = null)

{

$this->id = $id : uniqid();

}

public function id()

{

return $this->id;

}

public function equals(PostId $anId)

{

return $this->id === $anId->id();

}

}

If we wanted to persist this Post Entity, a simple in-memory Post Repository could be created like this:

class SimplePostRepository

{

private $post = [];

public add(Post $aPost)

{

$this->posts[(string) $aPost->id()] = $aPost;

}

public function postOfId(PostId $anId)

{

if (isset($this->posts[(string) $anId])) { return $this->posts[(string) $anId];

}

return null;

}

}

And, as you would expect, it's handled as a collection:

$id = new PostId();

$repository = new SimplePostRepository();

$repository->add(new Post($id, 'Random content'));

// later ...

$post = $repository->postOfId($id);

$post->editBody('Updated content');

// even later ...

$post = $repository->postOfId($id); assert('Updated content' === $post->body());

正如您所看到的，从集合的角度来看，在Repository中不需要保存方法。 影响对象的更改由底层持久层正确处理。 面向集合的存储库是那些不需要添加之前保存的集合的存储库。 这主要发生在基于内存的存储库上，但我们也有办法用持久性存储库来实现。 我们会在一会儿看这个; 另外，我们将在第11章“应用程序”中更深入地介绍这一点.

设计存储库的第一步是为它定义一个类似于集合的接口。 界面需要定义通常的收集方法，像这样:

interface PostRepository

{

public function add(Post $aPost); public function addAll(array $posts); public function remove(Post $aPost);

public function removeAll(array $posts);

// ...

}

For implementing such an interface, you could also use an abstract class. In general, when we talk about an interface, we refer to the general concept and not just the specific PHP interface. To keep your design simple, don't add methods you don't need; the Repository interface definition and its corresponding Aggregate should be placed in the same Module.

Sometimes remove doesn't physically delete the Aggregate from the database. This strategy

- where the Aggregate has a status field that's updated to a *deleted* value - is known as a *soft delete*. Why is this approach interesting? It can be interesting for auditing changes and performance. In those cases, you can instead mark the Aggregate as disabled or *logically removed*. The interface could be updated accordingly by removing the removal methods or providing disable behavior in the Repository.

存储库的另一个重要方面是查找方法，如下所示：

interface PostRepository

{

// ...

/\*\*

\* @return Post

\*/

public function postOfId(PostId $anId);

/\*\*

\* @return Post[]

\*/

public function latestPosts(DateTimeImmutable $sinceADate);

}

正如我们在第4章实体中所建议的，我们更喜欢应用程序生成的身份。 生成聚合的新标识的最佳位置是其存储库。 因此，为了检索Post的全球唯一ID，包含它的合理位置在PostRepository中:

interface PostRepository

{

// ...

/\*\*

\* @return PostId

\*/

public function nextIdentity();

}

负责构建每个Post实例的代码调用nextIdentity来获取唯一标识符PostId:

$post = newPost($postRepository->nextIdentity(), $body);

一些开发人员喜欢将实现放在接口定义的附近，作为模块的一个子包。 但是，因为我们需要明确分离问题，所以我们建议将其放入基础架构层。

### 内存中实现

As Uncle Bob wrote in [Screaming Architecture](http://blog.8thlight.com/uncle-bob/2011/09/30/Screaming-Architecture.html):

A good software architecture allows decisions about frameworks, databases, web- servers, and other environmental issues and tools, to be deferred and delayed. A good architecture makes it unnecessary to decide on Rails, or Spring, or Hibernate, or Tomcat or MySql, until much later in the project. A good architecture makes it easy to change your mind about those decisions too. A good architecture emphasizes the use-cases and decouples them from peripheral concerns.

In the early stages of your application, a fast in-memory implementation could come in handy. It's something you could use to mature other parts of your system, allowing you to delay database decisions to the correct moment. An in-memory Repository is simple, fast, and easy to implement.

对于我们的Post Repository，内存中的哈希映射足以提供我们需要的所有功能:

namespace Infrastructure\Persistence\InMemory;

use Domain\Model\Post; use Domain\Model\PostId;

use Domain\Model\PostRepository;

class InMemoryPostRepository implements PostRepository

{

private $posts = [];

public function add(Post $aPost)

{

$this->posts[$aPost->id()->id()] = $aPost;

}

public function remove(Post $aPost)

{

unset($this->posts[$aPost->id()->id()]);

}

public function postOfId(PostId $anId)

{

if (isset($this->posts[$anId->id()])) { return $this->posts[$anId->id()];

}

return null;

}

public function latestPosts(\DateTimeImmutable $sinceADate)

{

return $this->filterPosts(

function (Post $post) use($sinceADate) { return $post->createdAt() > $sinceADate;

}

);

}

private function filterPosts(callable $fn)

{

return array\_values(array\_filter($this->posts, $fn));

}

public function nextIdentity()

{

return new PostId();

}

}

### 学说ORM

在过去的章节中我们已经讨论过很多教义。 Doctrine是一组用于数据库存储和对象映射的库。 它默认捆绑了流行的Symfony2 web框架，除了其他功能外，它还允许您轻松地将应用程序与持久层分离，这要感谢Data Mapper模式.

与此同时，ORM站在一个强大的数据库抽象层上，通过一种名为Doctrine Query Language（DQL）的SQL方言实现数据库交互，该方法受到着名的Java Hibernate框架.

如果我们要使用Doctrine ORM，首先要完成的任务是通过Composer将依赖关系添加到我们的项目中：

composer require doctrine/orm

#### 对象映射

您的域对象和数据库之间的映射可以被视为实现细节。 域生命周期不应该知道这些持久性细节。 因此，映射信息应该被定义为域之外的基础设施层的一部分，并且作为存储库的实现。

##### Doctrine Custom Mapping Types

As our Post Entity is composed of Value Objects like Body or PostId, it's a good idea to make Custom Mapping Types or use Doctrine Embeddables for them, as seen in the Value Objects chapter. This will make the object mapping considerably easier:

namespace Infrastructure\Persistence\Doctrine\Types;

use Doctrine\DBAL\Types\Type;

use Doctrine\DBAL\Platforms\AbstractPlatform; use Domain\Model\Body;

class BodyType extends Type

{

public function getSQLDeclaration(

array $fieldDeclaration, AbstractPlatform $platform

) {

return $platform->getVarcharTypeDeclarationSQL(

$fieldDeclaration

);

}

/\*\*

* @param string $value
* @return Body

\*/

public function convertToPHPValue(

$value, AbstractPlatform $platform

) {

return new Body($value);

}

/\*\*

* @param Body $value

\*/

public function convertToDatabaseValue(

$value, AbstractPlatform $platform

) {

return $value->content();

}

public function getName()

{

return 'body';

}

}

namespace Infrastructure\Persistence\Doctrine\Types;

use Doctrine\DBAL\Types\Type;

use Doctrine\DBAL\Platforms\AbstractPlatform; use Domain\Model\PostId;

class PostIdType extends Type

{

public function getSQLDeclaration(

array $fieldDeclaration, AbstractPlatform $platform

) {

return $platform->getGuidTypeDeclarationSQL(

$fieldDeclaration

);

}

/\*\*

* @param string $value
* @return PostId

\*/

public function convertToPHPValue(

$value, AbstractPlatform $platform

) {

return new PostId($value);

}

/\*\*

* @param PostId $value

\*/

public function convertToDatabaseValue(

$value, AbstractPlatform $platform

) {

return $value->id();

}

public function getName()

{

return 'post\_id';

}

}

Don't forget to implement the toString magic method on the PostId Value Object, as Doctrine requires this:

class PostId

{

// ...

public function toString()

{

return $this->id;

}

}

Doctrine offers multiple formats for the mapping, such as YAML, XML, or annotations. XML is our preferred choice, as it provides robust IDE autocompletion:

<xml version="1.0" encoding="UTF-8">

<doctrine-mapping

[xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping"](http://doctrine-project.org/schemas/orm/doctrine-mapping) [xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"](http://www.w3.org/2001/XMLSchema-instance) xsi:schemaLocation="

<http://doctrine-project.org/schemas/orm/doctrine-mapping> [http://raw.github.com/doctrine/doctrine2/master/doctrine-mapping.xsd">](http://raw.github.com/doctrine/doctrine2/master/doctrine-mapping.xsd)

<entity name="Domain\Model\Post" table="posts">

<id name="id" type="post\_id" column="id">

<generator strategy="NONE" />

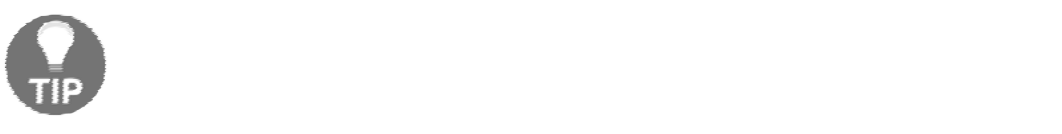
</id>

<field name="body" type="body" length="250" column="body"/>

<field name="createdAt" type="datetime" column="created\_at"/>

</entity>

</doctrine-mapping>



**Exercise**

Write down what the mapping would look like in the case of using the Doctrine Embeddables approach. Take a look at Chapter Value Objects or Chapter Entities if you need some help.

#### Entity Manager

The EntityManager is the central access point for the ORM functionality. Bootstrapping it is easy:

use Doctrine\DBAL\Types\Type; use Doctrine\ORM\EntityManager; use Doctrine\ORM\Tools;

Type::addType( 'post\_id',

'Infrastructure\Persistence\Doctrine\Types\PostIdType'

);

Type::addType( 'body',

'Infrastructure\Persistence\Doctrine\Types\BodyType'

);

$entityManager = EntityManager::create( [

'driver' => 'pdo\_sqlite',

'path'=> DIR

],

. '/db.sqlite',

Tools\Setup::createXMLMetadataConfiguration( ['/Path/To/Infrastructure/Persistence/Doctrine/Mapping'],

$devMode = true

)

);

Remember to configure it according to your needs and setup.

#### DQL Implementation

In the case of this Repository, we'll only need the EntityManager to retrieve Domain objects directly from the database:

namespace Infrastructure\Persistence\Doctrine;

use Doctrine\ORM\EntityManager; use Domain\Model\Post;

use Domain\Model\PostId;

use Domain\Model\PostRepository;

class DoctrinePostRepository implements PostRepository

{

protected $em;

public function construct(EntityManager $em)

{

$this->em = $em;

}

public function add(Post $aPost)

{

$this->em->persist($aPost);

}

public function remove(Post $aPost)

{

$this->em->remove($aPost);

}

public function postOfId(PostId $anId)

{

return $this->em->find('Domain\Model\Post', $anId);

}

public function latestPosts(\DateTimeImmutable $sinceADate)

{

return $this->em->createQueryBuilder()

->select('p')

->from('Domain\Model\Post', 'p')

->where('p.createdAt > :since')

->setParameter(':since', $sinceADate)

->getQuery()

->getResult();

}

public function nextIdentity()

{

return new PostId();

}

}

If you check some Doctrine examples out there, you may find that after running persist or remove, flush should be called. But as seen in our proposal, there's no call to flush.

Flushing and dealing with transactions is delegated to the Application Service. That's why you can work with Doctrine, considering that flushing all the changes on Entities will happen at the end of the request. In terms of performance, one flush call is best.

## 持久性存储库

有时候面向集合的存储库不适合我们的持久性机制。 如果你没有工作单位，跟踪总体变化是一件困难的事情。 坚持这种更改的唯一方法是明确调用save。

面向持久性存储库的接口定义与您将如何定义面向集合的对等方法类似：

interface PostRepository

{

public function nextIdentity();

public function postOfId(PostId $anId); public function save(Post $aPost); public function saveAll(array $posts); public function remove(Post $aPost); public function removeAll(array $posts);

}

In this case, we now have save and saveAll methods, which provide functionality similar to the previous add and addAll methods. However, the important difference is how the client uses them. Within a collection-oriented style, you use the add methods just once: when the Aggregate is created. In a persistence-oriented style, you'll not only use the save action after creating a new Aggregate, but also when an existing one is modified:

$post = new Post(/\* ... \*/);

$postRepository->save($post);

// later ...

$post = $postRepository->postOfId($postId);

$post->editBody(new Body('New body!'));

$postRepository->save($post);

Other than this difference, the details are only in the implementation.

### Redis Implementation

The [Redis](http://redis.io/) is an in-memory key value that can be used as a cache or store.

Depending on the circumstances, we could consider using Redis as a store for our Aggregates.

To get started, make sure you have a PHP client to connect to Redis. A good one that we recommend is [Predis](https://github.com/nrk/predis):

composer require predis/predis:~1.0 namespace Infrastructure\Persistence\Redis;

use Domain\Model\Post; use Domain\Model\PostId;

use Domain\Model\PostRepository; use Predis\Client;

class RedisPostRepository implements PostRepository

{

private $client;

public function construct(Client $client)

{

$this->client = $client;

}

public function save(Post $aPost)

{

$this->client->hset( 'posts',

(string) $aPost->id(), serialize($aPost)

);

}

public function remove(Post $aPost)

{

$this->client->hdel('posts', (string) $aPost->id());

}

public function postOfId(PostId $anId)

{

if($data = $this->client->hget('posts', (string) $anId)) { return unserialize($data);

}

return null;

}

public function latestPosts(\DateTimeImmutable $sinceADate)

{

$latest = $this->filterPosts(

function(Post $post) use ($sinceADate) { return $post->createdAt() > $sinceADate;

}

);

$this->sortByCreatedAt($latest);

return array\_values($latest);

}

private function filterPosts(callable $fn)

{

return array\_filter(array\_map(function ($data) { return unserialize($data);

},

$this->client->hgetall('posts')), $fn);

}

private function sortByCreatedAt(&$posts)

{

usort($posts, function (Post $a, Post $b) { if ($a->createdAt() == $b->createdAt()) {

return 0;

});

}

}

return ($a->createdAt() < $b->createdAt())  -1 : 1;

public function nextIdentity()

{

return new PostId();

}

}

### SQL Implementation

In a classic example, we could create a simple [PDO](http://php.net/manual/en/book.pdo.php) implementation for our

PostRepository just by using plain SQL queries:

namespace Infrastructure\Persistence\Sql;

use Domain\Model\Body; use Domain\Model\Post; use Domain\Model\PostId;

use Domain\Model\PostRepository;

class SqlPostRepository implements PostRepository

{

const DATE\_FORMAT = 'Y-m-d H:i:s'; private $pdo;

public function construct(\PDO $pdo)

{

$this->pdo = $pdo;

}

public function save(Post $aPost)

{

$sql ='INSERT INTO posts ' .

'(id, body, created\_at) VALUES ' . '(:id, :body, :created\_at)';

$this->execute($sql, [

'id' => $aPost->id()->id(),

'body' => $aPost->body()->content(), 'created\_at' => $aPost->createdAt()->format(

self::DATE\_FORMAT

)

]);

}

private function execute($sql, array $parameters)

{

$st = $this->pdo->prepare($sql);

$st->execute($parameters);

return $st;

}

public function remove(Post $aPost)

{

$this->execute('DELETE FROM posts WHERE id = :id', [ 'id' => $aPost->id()->id()

]);

}

public function postOfId(PostId $anId)

{

$st =$this->execute('SELECT \* FROM posts WHERE id = :id',[ 'id' => $anId->id()

]);

if($row = $st->fetch(\PDO::FETCH\_ASSOC)) { return $this->buildPost($row);

}

return null;

}

private function buildPost($row)

{

return new Post(

new PostId($row['id']),

new Body($row['body']),

new \DateTimeImmutable($row['created\_at'])

);

}

public function latestPosts(\DateTimeImmutable $sinceADate)

{

return $this->retrieveAll(

'SELECT \* FROM posts WHERE created\_at > :since\_date', [ 'since\_date' => $sinceADate->format(self::DATE\_FORMAT)

]

);

}

private function retrieveAll($sql, array $parameters = [])

{

$st = $this->pdo->prepare($sql);

$st->execute($parameters);

return array\_map(function ($row) { return $this->buildPost($row);

}, $st->fetchAll(\PDO::FETCH\_ASSOC));

}

public function nextIdentity()

{

return new PostId();

}

public function size()

{

return $this->pdo->query('SELECT COUNT(\*) FROM posts')

->fetchColumn();

}

}

As we don't have any mapping configuration, it would be very useful to have an initialization method for the schema within the same class. **Things that change together should remain together**:

class SqlPostRepository implements PostRepository

{

// ...

public function initSchema()

{

$this->pdo->exec(<<<SQL DROP TABLE IF EXISTS posts;

CREATE TABLE posts (

id CHAR(36) PRIMARY KEY, body VARCHAR (250) NOT NULL,

created\_at DATETIME NOT NULL

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4\_unicode\_ci; SQL);

}

}

### Extra Behavior

interface PostRepository

{

// ...

public function size();

}

The implementation could look like this:

class DoctrinePostRepository implements PostRepository

{

// ...

public function size()

{

return $this->em->createQueryBuilder()

->select('count(p.id)')

->from('Domain\Model\Post', 'p')

->getQuery()

->getSingleScalarResult();

}

}

Adding additional behavior to a Repository can be very beneficial. An example of this is the ability to count all the items in a given collection. You might think to add a method with the name count; however, as we're trying to mimic a collection, a better name would instead be size:

You're also able to place specific calculations, counters, read-optimized queries, or complex commands (INSERT, UPDATE, or DELETE) into the Repository. However, all behavior should still follow the Repositories' collection characteristics. You're encouraged to move as much logic into Domain-specific stateless Domain Services as possible, instead of simply adding these responsibilities to the Repository.

In some instances, you won't require the entire Aggregate for simply accessing small amounts of information. To solve this, you can add Repository methods to access these as shortcuts. You should make sure to only access data that could be retrieved by navigating through the Aggregate Root. As such, you shouldn't allow access to the private and internal areas of the Aggregate Root, as this would violate the laid out contractual agreement.

For some use cases, you'll require very specific queries that are compositions of multiple Aggregate types, each returning specific information. These queries can be run and then returned as a single Value Object. It's very common for Repositories to return Value Objects.

If you find yourself creating many use case optimal finder methods, you may be introducing a common code smell. This could be an indication of a misjudged Aggregate boundary. If, however, you're confident that the boundaries are correct, it could be time to explore CQRS.

## 查询存储库

经过比较，如果我们考虑他们的查询能力，存储库与集合不同。 存储库处理执行查询时通常不在内存中的大量对象。 将Domain对象的所有实例加载到内存中并对它们执行查询是不可行的。

一个好的解决方案是通过一个标准并让Repository处理实现细节以成功执行操作。 它可能会将标准转换为SQL或ORM查询或遍历内存中的集合。 但是，这并不重要，因为实施涉及它.

### 规格模式

规范对象的常见实现是规范模式。 规范是一个简单的谓词，它接受一个Domain对象并返回一个布尔值。 给定一个Domain对象，如果它指定了规范则返回true，否则返回false:

interface PostSpecification

{

/\*\*

* @return boolean

\*/

public function specifies(Post $aPost);

}

We just need to add a query method to our Repository:

interface PostRepository

{

// ...

public function query($specification);

}

#### In-Memory Implementation

As an example, if we wanted to replicate the latestPosts query method in our PostRepository by using a Specification for an in-memory implementation, it would look like this:

namespace Infrastructure\Persistence\InMemory; use Domain\Model\Post;

interface InMemoryPostSpecification

{

/\*\*

* @return boolean

\*/

public function specifies(Post $aPost);

}

The in-memory implementation for the latestPosts behavior could look like this:

namespace Infrastructure\Persistence\InMemory; use Domain\Model\Post;

class InMemoryLatestPostSpecification implements InMemoryPostSpecification

{

private $since;

public function construct(\DateTimeImmutable $since)

{

$this->since = $since;

}

public function specifies(Post $aPost)

{

return $aPost->createdAt() > $this->since;

}

}

The query method for our Repository implementation could look like this:

class InMemoryPostRepository implements PostRepository

{

// ...

/\*\*

* @param InMemoryPostSpecification $specification

\*

* @return Post[]

\*/

public function query($specification)

{

return $this->filterPosts(

function (Post $post) use($specification) { return $specification->specifies($post);

}

);

}

}

Retrieving all the latest posts from the Repository is as simple as creating a tailored instance of the above implementation:

$latestPosts = $postRepository->query(

new InMemoryLatestPostSpecification(new \DateTimeImmutable('-24'))

);

#### SQL Implementation

A standard specification works well for in-memory implementations. However, as we don't pre-load all the Domain objects in memory for an SQL implementation, we need a more specific specification for these cases:

namespace Infrastructure\Persistence\Sql;

interface SqlPostSpecification

{

/\*\*

* @return string

\*/

public function toSqlClauses();

}

The SQL implementation for this specification could look like this:

namespace Infrastructure\Persistence\Sql;

class SqlLatestPostSpecification implements SqlPostSpecification

{

private $since;

public function construct(\DateTimeImmutable $since)

{

$this->since = $since;

}

public function toSqlClauses()

{

return "created\_at >'" .

$this->since->format('Y-m-d H:i:s') . "'";

}

}

And here's an example of how to query an SQLPostRepository implementation:

class SqlPostRepository implements PostRepository

{

// ...

/\*\*

* @param SqlPostSpecification $specification

\*

* @return Post[]

\*/

public function query($specification)

{

return $this->retrieveAll(

'SELECT \* FROM posts WHERE ' .

$specification->toSqlClauses()

);

}

private function retrieveAll($sql, array $parameters = [])

{

$st = $this->pdo->prepare($sql);

$st->execute($parameters);

return array\_map(function ($row) { return $this->buildPost($row);

}, $st->fetchAll(\PDO::FETCH\_ASSOC));

}

}

## 管理交易

域模型不是管理交易的地方。 应用于域模型的操作应该与持久性机制无关。 解决此问题的常用方法是在应用层中放置Facade，从而将相关用例分组在一起。 当从UI层调用Facade的方法时，业务方法开始一个事务。 完成后，Facade通过提交事务来结束交互。 如果出现任何问题，交易将回滚：

use Doctrine\ORM\EntityManager;

class SomeApplicationServiceFacade

{

private $em;

public function construct(EntityManager $em)

{

$this->em = $em;

}

public function doSomeUseCaseTask()

{

try {

$this->em->getConnection()->beginTransaction();

// Use domain model

$this->em->getConnection()->commit();

} catch (Exception $e) {

$this->em->getConnection()->rollback(); throw $e;

}

}

}

The problem introduced with Facades is that we have to repeat the same boilerplate code over and over. If we unify the way we execute use cases, we could wrap them in a transaction using the [Decorator pattern](http://en.wikipedia.org/wiki/Decorator_pattern):

interface ApplicationService

{

/\*\*

* @param $request
* @return mixed

\*/

public function execute(BaseRequest $request);

}

class SomeApplicationService implements ApplicationService

{

public function execute(BaseRequest $request)

{

// do something

}

}

We don't want to couple our Application layer with the concrete transactional procedure, so instead we can create a simple interface for it:

interface TransactionalSession

{

/\*\*

* + @param callable $operation
  + @return mixed

\*/

public function executeAtomically(callable $operation);

}

A Decorator's pattern implementation that can make any Application Service transactional is as easy as this:

class TransactionalApplicationService implements ApplicationService

{

private $session; private $service;

public function construct( ApplicationService $service, TransactionalSession $session

) {

$this->session = $session;

$this->service = $service;

}

public function execute(BaseRequest $request)

{

$operation = function() use($request) { return $this->service->execute($request);

};

return $this->session->executeAtomically(

$operation->bindTo($this)

);

}

}

Following this, we could alternatively create a Doctrine transactional session implementation:

class DoctrineSession implements TransactionalSession

{

private $entityManager;

public function construct(EntityManager $entityManager)

{

$this->entityManager = $entityManager;

}

public function executeAtomically(callable $operation)

{

return $this->entityManager->transactional($operation);

}

}

Now we have everything to execute our use cases within a transaction:

$useCase = new TransactionalApplicationService( new SomeApplicationService(

// ...

),

new DoctrineSession(

// ...

)

);

$response = $useCase->execute();

## Testing Repositories

In order to be sure that the Repository will work in production, we'll need to test its implementation. To do this, we have to test the boundaries of the system, making sure that our expectations are correct.

In the case of a Doctrine test, the setup will be a little bit more sophisticated:

use Doctrine\DBAL\Types\Type; use Doctrine\ORM\EntityManager; use Doctrine\ORM\Tools;

use Domain\Model\Post;

class DoctrinePostRepositoryTest extends \PHPUnit\_Framework\_TestCase

{

private $postRepository;

public function setUp()

{

$this->postRepository = $this->createPostRepository();

}

private function createPostRepository()

{

$this->addCustomTypes();

$em = $this->initEntityManager();

$this->initSchema($em);

return new PrecociousDoctrinePostRepository($em);

}

private function addCustomTypes()

{

if (!Type::hasType('post\_id')) { Type::addType(

'post\_id', 'Infrastructure\Persistence\Doctrine\Types\PostIdType'

);

}

if (!Type::hasType('body')) { Type::addType(

'body', 'Infrastructure\Persistence\Doctrine\Types\BodyType'

);

}

}

protected function initEntityManager()

{

return EntityManager::create(

['url' => 'sqlite:///:memory:'], Tools\Setup::createXMLMetadataConfiguration(

['/Path/To/Infrastructure/Persistence/Doctrine/Mapping'],

$devMode = true

)

);

}

private function initSchema(EntityManager $em)

{

$tool = new Tools\SchemaTool($em);

$tool->createSchema([

$em->getClassMetadata('Domain\Model\Post')

]);

}

// ...

}

class PrecociousDoctrinePostRepository extends DoctrinePostRepository

{

public function persist(Post $aPost)

{

parent::persist($aPost);

$this->em->flush();

}

public function remove(Post $aPost)

{

parent::remove($aPost);

$this->em->flush();

}

}

Once we have this environment set up, we can continue to test the Repository's behavior:

class DoctrinePostRepositoryTest extends \PHPUnit\_Framework\_TestCase

{

// ...

/\*\*

* + @test

\*/

public function itShouldRemovePost()

{

$post = $this->persistPost('irrelevant body');

$this->postRepository->remove($post);

$this->assertPostExist($post->id());

}

private function assertPostExist($id)

{

$result = $this->postRepository->postOfId($id);

$this->assertNull($result);

}

private function persistPost(

$body,

\DateTimeImmutable $createdAt = null

) {

$this->postRepository->add(

$post = new Post(

$this->postRepository->nextIdentity(), new Body($body),

$createdAt

)

);

return $post;

}

}

Following our earlier assertion, if we save a Post, we expect to find it in the exact same state.

Now we can move on to test finding the latest posts by specifying a given date:

class DoctrinePostRepositoryTest extends \PHPUnit\_Framework\_TestCase

{

// ...

/\*\*

* + @test

\*/

public function itShouldFetchLatestPosts()

{

$this->persistPost(

'a year ago', new \DateTimeImmutable('-1 year')

);

$this->persistPost(

'a month ago', new \DateTimeImmutable('-1 month')

);

$this->persistPost(

'few hours ago', new \DateTimeImmutable('-3 hours')

);

$this->persistPost(

'few minutes ago', new \DateTimeImmutable('-2 minutes')

);

$posts = $this->postRepository->latestPosts( new \DateTimeImmutable('-24 hours')

);

$this->assertCount(2, $posts);

$this->assertEquals(

'few hours ago', $posts[0]->body()->content()

);

$this->assertEquals(

'few minutes ago', $posts[1]->body()->content()

);

}

}

## Testing Your Services with In-Memory Implementations

Setting up a fully persistent Repository implementation can be complex and result in slow execution. You should care about keeping your tests fast. Going through the whole database setup and then querying will slow you down enormously. Having an in-memory implementation could help delay persistence decisions until the end. We can test in the same manner as we did before, but this time, we'll use a full-featured fast and simple in- memory implementation:

class MyServiceTest extends \PHPUnit\_Framework\_TestCase

{

private $service;

public function setUp()

{

$this->service = new MyService( new InMemoryPostRepository()

);

}

}

## Wrap-Up

A Repository is a mechanism that acts as a storage location. The difference between a DAO and a Repository is that a DAO follows a database-first approach, decreasing cohesion with many low-level methods to query the database. Depending on the underlying persistence mechanics, we've seen different Repository approaches:

**Collection-oriented Repositories** tend to be purer to the Domain model, even if they persist Entities. From the client's point of view, a collection-oriented Repository looks like a collection (Set). There's no need for explicit persistence calls on Entity updates, as the Repository tracks changes on the objects. We explored how to use Doctrine as the underlying persistence mechanism for this type of Repository.

**Persistence-oriented Repositories** require explicit persistence calls, as they don't track object changes. We explored Redis and plain SQL implementations.

Along the way, we discovered Specifications as a pattern that helps us query the database without sacrificing flexibility and cohesion. We also studied how to manage transactions and how to test our services with simple and fast in-memory Repository implementations.

# Application

****

应用程序层是将域模型与查询或更改其状态的客户端分开的区域。 应用程序服务是这种层的构建块。 正如Vaughn Vernon所说：“应用程序服务是领域模型的直接客户。” 您可以将Application Service视为外部世界（HTML表单，API客户端，命令行，框架，UI等）与域模型本身之间的联系点。 通过考虑系统公开给世界的顶级用例可能会有所帮助，例如：“作为访客，我想注册”，“作为登录用户，我想购买产品”等等.

In this chapter, we'll explore how to implement Application Services, understand the role of the Command pattern, and establish the responsibilities of an Application Service. To do this, let's consider the use case of *signing up a new user*.

Conceptually, in order to register a new user, we need to:

Get an email and password from the client Check if the email is already in use

Create a new user

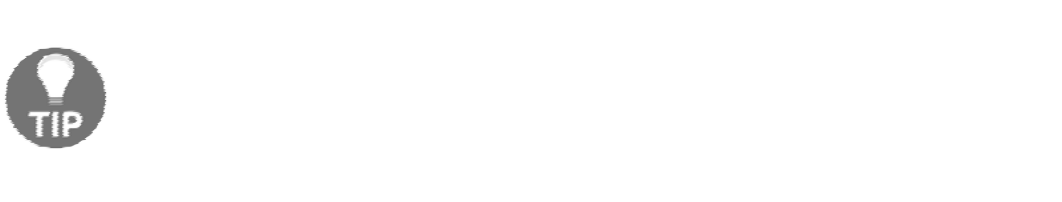
Add this new user to the existing user set Return the user we've just created

Let's go for it.

## Requests

我们需要将电子邮件和密码发送到应用程序服务。 从客户端（HTML表单，API客户端甚至命令行）执行这样的事情有很多种方法。

我们可以通过方法签名发送标准参数（电子邮件和密码），或者发送一个包含这些信息的数据结构。 后一种方法，发送一个DTO，带来一些有趣的功能到桌子上。 通过发送一个对象，可以将其序列化并排列在命令总线上。 也可以添加类型安全和一些IDE帮助。



**Data Transfer Object**

A DTO is a data structure that carries information between processes. Don't mistake it for a full-featured object. A DTO doesn't have any behavior except for storage and retrieval of its own data (accessors and mutators). DTOs are simple objects that shouldn't contain any business logic that would require testing.

As Vaughn Vernon [says](https://www.amazon.com/Implementing-Domain-Driven-Design-Vaughn-Vernon/dp/0321834577):

Application Service method signatures use only primitive types (int, strings, and so on.), and possibly DTOs. As an alternative to these approaches, however, a better approach may be to design Command objects instead. There is not necessarily a right or wrong way. It mostly depends on your tastes and goals.

The implementation for a DTO that holds the data required for the Application Service could be something like this:

namespace Lw\Application\Service\User;

class SignUpUserRequest

{

private $email; private $password;

public function construct($email, $password)

{

$this->email = $email;

$this->password = $password;

}

public function email()

{

return $this->email;

}

public function password()

{

return $this->password;

}

}

As you see, SignUpUserRequest has no behavior, only data. This could have come from an HTML form or an API endpoint, though we don't care which.

### Building Application Service Requests

Creating a request from the delivery mechanism, your favorite framework, should be pretty straightforward. On the web, you could pick up parameters from the controller request and pass them down to the Service inside a DTO. The same principle applies for a CLI command: read input parameters and send them down again.

With Symfony, we can extract the data we need from Request object from the

HttpFoundation component:

// ...

class UsersController extends Controller

{

/\*\*

* + @Route('/signup', name = 'signup')
  + @param Request $request
  + @return Response

\*/

public function signUpAction(Request $request)

{

// ...

$signUpUserRequest = new SignUpUserRequest(

$request->get('email'),

$request->get('password')

}

// ...

);

// ...

On a more elaborate Silex application that uses the Form component to capture and validate parameters, it would look like this:

// ...

$app->match('/signup', function (Request $request) use ($app) {

$form = $app['sign\_up\_form'];

$form->handleRequest($request);

if ($form->isValid()) {

$data = $form->getData();

try {

$app['sign\_in\_user\_application\_service']->execute( new SignUpUserRequest(

$data['email'],

$data['password']

)

);

return $app->redirect(

$app['url\_generator']->generate('login')

);

} catch (UserAlreadyExistsException $e) {

$form

->get('email')

->addError(

new FormError(

'Email is already registered by another user'

)

);

} catch (Exception $e) {

$form

->addError(

new FormError(

'There was an error, please get in touch with us'

)

);

}

}

return $app['twig']->render('signup.html.twig', [ 'form' => $form->createView(),

]);

});

### Request Design

When designing your request objects, you should always follow these principles: use primitives, design for serialization, and don't include business logic inside. This way, you'll be able to save unit testing dollars.

#### Use Primitives

We recommend using basic types to build up your request objects — that means strings, integers, booleans, and so on. We're just abstracting away input parameters. You should be able to consume Application Services independently from the delivery mechanism. Even pretty complicated HTML forms get translated into basic types all the time at the controller level. You don't want to mix up your framework and your business logic.

With certain scenarios, it's tempting to use Value Objects directly. Don't do it. Updates on the Value Object definition will affect all clients, and you'll be coupling clients with your Domain logic.

#### Serializable

A cool side effect of using basic types is that any request object can easily be serialized into a string, sent through the wire, and stored in a messaging system or database.

#### No Business Logic

Avoid putting any business logic — even validation — inside your request objects. Validation should take place inside your Domain — this is inside your Entities, Value Objects, Domain Services, etc. Validation is a way of enforcing business invariants and Domain constraints.

#### No Tests

Application requests are data structures, not objects. Unit testing data structures is like testing getters and setters. There's no behavior to test, so there isn't much value in trying to unit test request objects and DTOs. These structures will be covered as a side effect of more elaborate tests, such as Integration or Acceptance tests.

Commands are an alternative to request objects. We could design a Service with multiple Application methods, and each one of them with the parameters you'd put inside the Request. This is OK for simple applications, but we'll worry about this topic later.

## Anatomy of an Application Service

Once we have the data encapsulated in a request, it's time for the business logic. As Vaughn Vernon [says](https://www.amazon.com/Implementing-Domain-Driven-Design-Vaughn-Vernon/dp/0321834577): "Keep Application Services thin, using them only to coordinate tasks on the model."

The first thing to do is to extract the necessary information from the request, That is, the email and password. At a high level, we need to check if there's an existing user with a particular email. If this isn't the case, then we create and add the user to the UserRepository. In the special case of finding a user with the same email, we raise an exception so the client can treat it their own way — by displaying an error, retrying, or just ignoring it:

namespace Lw\Application\Service\User;

use Ddd\Application\Service\ApplicationService; use Lw\Domain\Model\User\User;

use Lw\Domain\Model\User\UserAlreadyExistsException; use Lw\Domain\Model\User\UserRepository;

class SignUpUserService

{

private $userRepository;

public function construct(UserRepository $userRepository)

{

$this->userRepository = $userRepository;

}

public function execute(SignUpUserRequest $request)

{

$email = $request->email();

$password = $request->password();

$user = $this->userRepository->ofEmail($email); if ($user) {

throw new UserAlreadyExistsException();

}

$this->userRepository->add(

new User(

$this->userRepository->nextIdentity(),

$email ,

$password

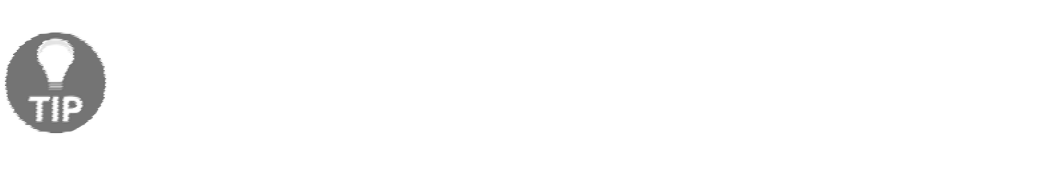
)

);

}

}

Nice! If you're wondering what this UserRepository thing is doing in the constructor, we'll show you that next.



**Handling Exceptions**

Exceptions raised by Application Services are a way of communicating unusual cases and flows to the client. Exceptions on this layer are related to business logic (like not finding a user), and not implementation details (like PDOException, PredisException, or DoctrineException).

### Dependency Inversion

Handling users is not the responsibility of the Service. As we saw in [Chapter](#_bookmark315)

[10](#_bookmark315), *Repositories*, there's a specialized class that deals with User collections: the User Repository. This is a dependency from the Application Service to the Repository. We don't want to couple the Application Service with a concrete implementation of the Repository, as then we'd be coupling our Service with Infrastructure details. So we depend on the contract (interface) that concrete implementations depend on, the UserRepository.

A specific implementation of the UserRepository will be built and passed in at runtime — for example, with DoctrineUserRepository, a specific implementation that uses Doctrine. Passing a specific implementation will also work when testing. For example, NotAvailableUserRepository can be a specific implementation that will throw exceptions each time an operation is performed. This way, we can test all Application Service behaviors, including *sad* paths, which is when the application must behave properly, even if something goes wrong.

Application Services could depend on Domain Services like GetBadgesByUser too. At runtime, the implementation for such a Service could be quite elaborate. Imagine an HttpGetBadgesByUser for integrating a Bounded Context through HTTP protocol.

Depending on abstractions, we'll make our Application Service immune to low-level Infrastructure changes.

### Instantiating Application Services

Instantiating just your Application Service is easy, but building the dependency tree might be tricky, depending on how complicated the dependencies are to build. For such a purpose, most frameworks come with a Dependency Injection Container. Without one, you'll end up with something like the following code somewhere in your controller:

$redisClient = new Predis\Client([ 'scheme' => 'tcp',

'host' => '10.0.0.1',

'port' => 6379

]);

$userRepository = new RedisUserRepository($redisClient);

$signUp = new SignUpUserService($userRepository);

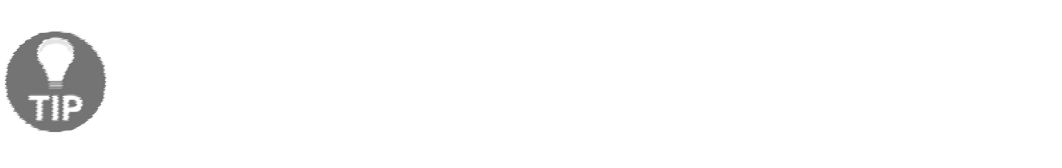
$signUp->execute(new SignUpUserRequest( 'user@example.com',

'password'

));

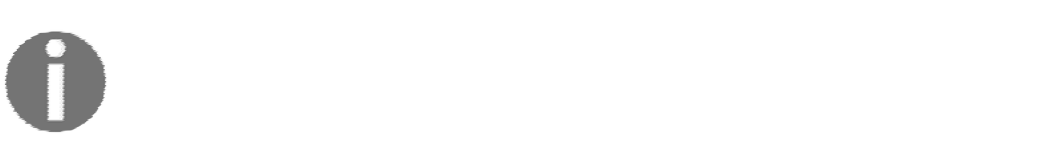
We decided to use the [Redis](http://redis.io/) implementation for the UserRepository. In the previous code example, we built all dependencies needed for building a Repository that uses Redis internally. Those dependencies are: a [Predis](https://github.com/nrk/predis) client, and all parameters to connect to our Redis server. This is not only inefficient, but it also spreads duplication across controllers.

You could refactor the construction logic into a Factory, or you could use a Dependency Injection Container — most modern frameworks come with it.



**Is It Bad to Use a Dependency Injection Container?**

Not at all. Dependency Injection Containers are just a tool. They help by abstracting away the complexities of building your dependencies. They come in handy for building Infrastructure artifacts. Symfony offers a complete solution.



Take into account the fact that passing the entire container as a whole to one of the Services is a bad practice. That would be like coupling the entire context of your application with the Domain. If a Service needs specific objects, build them from your framework and pass them as dependencies into the Service, but don't make that Service aware of the entire context.

Let's see how would we build dependencies in Silex:

$app = new \Silex\Application();

$app['redis\_parameters'] = [ 'scheme' => 'tcp', 'host' => '127.0.0.1',

'port' => 6379

];

$app['redis'] = $app->share(function ($app) {

return new Predis\Client($app['redis\_parameters']);

});

$app['user\_repository'] = $app->share(function($app) { return new RedisUserRepository(

$app['redis']

);

});

$app['sign\_up\_user\_application\_service'] = $app->share(function($app) { return new SignUpUserService(

$app['user\_repository']

);

});

// ...

$app->match('/signup' ,function (Request $request) use ($app) {

// ...

$app['sign\_up\_user\_application\_service']->execute( new SignUpUserRequest(

$request->get('email'),

$request->get('password')

)

);

// ...

});

As you can see, $app is used as the Service Container. We register all the components needed, along with their dependencies. sign\_up\_user\_application\_service depends on the definitions made above. Changing the implementation for the user\_repository is as easy as returning something else (MySQL, MongoDB, and so on.), so we don't need to change the Service code at all.

The equivalent for a Symfony application looks like this:

<xml version=" 1.0" >

<container [xmlns="http://symfony.com/schema/dic/services"](http://symfony.com/schema/dic/services) [xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"](http://www.w3.org/2001/XMLSchema-instance) xsi:schemaLocation="

<http://symfony.com/schema/dic/services> [http://symfony.com/schema/dic/services/services-1.0.xsd">](http://symfony.com/schema/dic/services/services-1.0.xsd)

<services>

<service

id="sign\_up\_user\_application\_service" class="SignUpUserService">

<argument type="service" id="user\_repository" />

</service>

<service

id="user\_repository" class="RedisUserRepository">

<argument type="service">

<service class="Predis\Client" />

</argument>

</service>

</services>

</container>

Now that you have the definition of your Application Service in the Symfony Service Container, getting it later is pretty straightforward. All delivery mechanisms — Web Controllers, REST Controllers, and even Console Commands — share the same definition. The Service is available on any class implementing the ContainerAware interface. Getting the Service is as easy as calling $this->get('sign\_up\_user\_application\_service'.

To summarize, how you build your Services (adhoc, using Service Containers, using Factories, and so on.) doesn't matter. However, it's important to keep your Application Services setup out of the Infrastructure boundary.

#### Customize an Application Service

The main way to customize your Application Service is by choosing which dependencies you're passing in. Depending on your Service Container capabilities, that could be a bit tricky, so you can also add a setter to change the dependency on the fly. For example, you may need to change an output dependency so that you can set up a default one and then change it afterward. If logic gets too complicated, you can create an Application Service Factory that will handle this situation for you.

### Execution

There are two different approaches for invoking Application Services: a dedicated class per use case with a single execution method, and multiple Application Services and use cases inside the same class.

#### One Class Per Application Service

This is our preferred approach, and probably the one that fits all scenarios:

class SignUpUserService

{

// ...

public function execute(SignUpUserRequest $request)

{

// ...

}

}

Using a dedicated class per Application Service makes the code more robust against external changes (Single Responsibility Principle). There are fewer reasons to change the class, as the Service does one and only one thing. The Application Service will be easier to test, seeing as it does less things. It's easier to implement a common Application Service contract, making class decoration easier (check out Sub section *Transactions* of [Chapter 10](#_bookmark315), *Repositories* ). This will also result in higher cohesion, as all dependencies are exclusively dedicated to a single use case.

The execution method could have a more expressive name, like signUp. However, the execute [Command pattern](http://martinfowler.com/bliki/DecoratedCommand.html) format standardizes a common contract across Application Services, thereby enabling easy decoration, which comes in handy for transactions.

#### Multiple Application Service Methods per Class

Sometimes it might be a good idea to group cohesive Application Services under the same class:

class UserService

{

// ...

public function signUp(SignUpUserRequest $request)

{

// ...

}

public function signIn(SignUpUserRequest $request)

{

// ...

}

public function logOut(LogOutUserRequest $request)

{

// ...

}

}

We don't recommend such an approach, as not all Application Services are 100 percent cohesive. Some Services will require different dependencies, and you'll end up with Application Services depending on things they don't need. Another issue is that this kind of class grows fast. As it violates the Single Responsibility Principle, there will be multiple reasons to change and maybe even break it.

### Returning Values

After signing up, we might be thinking about redirecting the user to a profile page. The natural way of passing the required information back to the controller is to return the User Entity directly from the Service:

class SignUpUserService

{

// ...

public function execute(SignUpUserRequest $request)

{

$user = new User(

$this->userRepository->nextIdentity(),

$email,

$password

);

$this->userRepository->add($user);

return $user;

}

}

Then, from the controller, we would pick up the id field and redirect to some other place. However, think twice about what we've just done. We returned a full-featured Entity to the controller, which will allow the delivery mechanism to bypass the Application Layer and interact directly with the Domain.

Imagine the User Entity offers an updateEmailAddress method. You could try to prevent it, but at some point in the future, somebody might think about using it:

$app-> match( '/signup' , function (Request $request) use ($app) {

// ...

$user = $app['sign\_up\_user\_application\_service']->execute( new SignUpUserRequest(

$request->get('email'),

$request->get('password'))

);

$user->updateEmailAddress('shouldnotupdate@email.com');

// ...

});

Not only that, but the data that the presentation layer needs is not the same that the Domain manages. We don't want to evolve and couple the Domain layer around the presentation layer. Instead, we want them to evolve freely.

To do this, we need a flexible way of decoupling both layers.

#### DTO from Aggregate Instances

We could return sterile data structures with the information the presentation layer needs. As we've seen before, DTOs fit with this scenario. We just need to compose them in the Application Service and return them to the client:

class UserDTO

{

private $email ;

// ...

public function construct(User $user)

{

$this->email = $user->email ();

// ...

}

public function email ()

{

return $this->email ;

}

}

The UserDTO will expose whatever read-only data we need from the User Entity on the presentation layer, thereby avoiding exposing behavior:

class SignUpUserService

{

public function execute(SignUpUserRequest $request)

{

// ...

$user = // ...

return new UserDTO($user);

}

}

Mission accomplished. Now we could pass parameters to the template engine and transform them into widgets, tags, or subtemplates, or do whatever we want with the data on the presentation side:

$app->match('/signup' , function (Request $request) use ($app) {

/\*\*

* + @var UserDTO $user

\*/

$userDto=$app['sign\_up\_user\_application\_service']->execute( new SignUpUserRequest(

$request->get('email'),

$request->get('password')

)

);

// ...

});

However, letting the Application Service decide how to build the DTO reveals another limitation. As building the DTO depends exclusively on the Application Service, adapting the DTO to different clients will be very difficult. Consider the data needed for a redirect on a Web Controller and the data needed for a REST response for the same use case. Not the same data at all.

Let's allow the client to define how to build the DTO by passing a specific DTO Assembler:

class SignUpUserService

{

private $userDtoAssembler;

public function construct( UserRepository $userRepository, UserDTOAssembler $userDtoAssembler

) {

$this->userRepository = $userRepository;

$this->userDtoAssembler = $userDtoAssembler;

}

public function execute(SignUpUserRequest $request)

{

$user = // ...

return $this->userDtoAssembler->assemble($user);

}

}

Now the client can customize the response by passing a specific UserDTOAssembler.

#### Data Transformers

There are some cases where generating intermediate DTOs for more complex responses like JSON, XML, CSV, and iCAL Contact could be seen as an unnecessary overhead. We could output the representation in a buffer and ask for it later on the delivery side.

Transformers help reduce this overhead by transforming high-level Domain concepts into low-level client details. Let's see an example:

interface UserDataTransformer

{

public function write(User $user);

/\*\*

* + @return mixed

\*/

public function read();

}

Consider the case of generating different data representations for a given product. Usually, the product information is served through a web interface (HTML), but we might be interested in offering other formats, like XML, JSON, or CSV. This might enable integrations with other Services.

Consider a similar case for a blog. We might expose our potential as writers in HTML to the world, but some people will be interested in consuming our articles through RSS. The use cases — Application Services — remain the same. The representation doesn't.

DTOs are a clean and simple solution that could be passed to template engines for different representations, but this might complicate the logic of this last step of data transformation, as the logic for such templates could become a problem to maintain, test, and understand.

Data Transformers might be a better approach on specific cases. These are just black boxes with Domain concepts (Aggregates, Entities, and so on.) as inputs and read-only representations (XML, JSON, CSV, and so on.) as outputs. These transformers could be really easy to test:

class JsonUserDataTransformer implements UserDataTransformer

{

private $data;

public function write(User $user)

{

// More complex logic could be placed here

// As using JMSSerializer, native json, etc.

$this->data = json\_encode($user);

}

/\*\*

* + @return string

\*/

public function read()

{

return $this->data;

}

}

那很简单。 想知道XML或CSV的外观如何？ 我们来看看如何将Data Transformer与我们的应用服务集成:

class SignUpUserService

{

private $userRepository; private $userDataTransformer;

public function construct( UserRepository $userRepository,

UserDataTransformer $userDataTransformer

) {

$this->userRepository = $userRepository;

$this->userDataTransformer = $userDataTransformer;

}

public function execute(SignUpUserRequest $request)

{

$user = // ...

$this->userDataTransformer()->write($user);

}

/\*\*

* + @return UserDataTransformer

\*/

public function userDataTransformer()

{

return $this->userDataTransformer;

}

}

这与DTO汇编程序方法类似，但是这次没有返回具体的值。 数据转换器被用来保存和交互数据.

DTO的主要问题是编写它们的开销。 大多数时候，您的域概念和DTO表示将呈现相同的结构。 大多数时候，你会觉得这是不值得花时间做这样的映射。 也就是说，表示和聚合之间的关系不是1：1。 您可以用一种表示法将两个聚合表示在一起。 您也可以用多种方式表示相同的聚合。 你如何做它总是取决于你的用例.

However, according to [Martin Fowler](http://www.martinfowler.com/books/eaa.html):

One case where it is useful to use something like a DTO is **when you have a significant mismatch between the model in your presentation layer and the underlying domain model**. In this case it makes sense to make presentation specific facade/gateway that maps from the domain model and presents an interface that's convenient for the presentation. It fits in nicely with Presentation Model. This is worth doing, but it is only worth doing for screens that have this mismatch (in this case it isn't extra work, since you'd have to do it in the screen anyway.)

We think the long-term vision will be worth the investment. On medium to big projects, interface representations and Domain concepts change at very different rhythms. You might want to decouple them from each other to lower the friction for updates. Using DTOs or Data Transformers allows you to evolve your model freely without having to think about breaking the layout all the time.

### Multiple Application Services on Compound Layouts

Most of the time, no layout is as simple as a single Application Service. Our projects have pretty complicated interfaces.

Consider the homepage of a specific project. How can we render so many pieces and use cases? There are a few options, so let's check them out.

#### AJAX Content Integration

You could let the browser ask for different endpoints directly and combine the data in the layout right after through AJAX or [Hijax](https://en.wikipedia.org/wiki/Hijax). This will avoid mixing a lot of Application Services in your controllers, but it might have a performance penalty, depending on the number of requests triggered.

#### ESI Content Integration

**Edge Side Includes** (**ESI**) is a tiny markup language similar to the previous approach, but on the server side. It requires additional effort configuring extra middleware, like NGINX or Varnish, to make it work. Includes (ESI) is a tiny markup language similar to the previous approach, but on the server side. It requires additional effort configuring extra middleware, like NGINX or [Varnish](https://en.wikipedia.org/wiki/Edge_Side_Includes), to make it work.

#### Symfony Sub Requests

If you use Symfony, Sub Requests could be an interesting option. According to the [Symfony](http://symfony.com/doc/current/components/http_kernel/introduction.html#sub-requests) [Documentation](http://symfony.com/doc/current/components/http_kernel/introduction.html#sub-requests):

In addition to the main request that's sent into HttpKernel::handle, you can also send so-called sub request. A sub request looks and acts like any other request, but typically serves to render just one small portion of a page instead of a full page. You'll most commonly make sub-requests from your controller (or perhaps from inside a template, that's being rendered by your controller). This creates another full request-response cycle where this new Request is transformed into a Response. The only difference internally is that some listeners (Example: security) may only act upon the master request. Each listener is passed some sub-class of KernelEvent, whose isMasterRequest() can be used to check if the current request is a master or sub request.

This is great, as you'll get the benefits of invoking separate Application Services without AJAX penalties or complicated ESI configurations.

#### One Controller, Multiple Application Services

One last option could be managing multiple Application Services within the same controller, though the controller logic could get a little bit dirty, as it'll handle and merge the responses to pass to the view.

## Testing Application Services

As you're interested in testing the behavior of the Application Service itself, there's no need to turn it into an integration test with complicated setups going against a real database.

You're not interested in testing the low-level details, so most of the time, a unit test will be enough:

class SignUpUserServiceTest extends \PHPUnit\_Framework\_TestCase

{

/\*\*

* + @var \Lw\Domain\Model\User\UserRepository

\*/

private $userRepository;

/\*\*

* + @var SignUpUserService

\*/

private $signUpUserService;

public function setUp()

{

$this->userRepository = new InMemoryUserRepository();

$this->signUpUserService = new SignUpUserService(

$this->userRepository

);

}

/\*\*

* + @test
  + @expectedException
  + \Lw\Domain\Model\User\UserAlreadyExistsException

\*/

public function alreadyExistingEmailShouldThrowAnException()

{

$this->executeSignUp();

$this->executeSignUp();

}

private function executeSignUp()

{

return $this->signUpUserService->execute( new SignUpUserRequest(

'user@example.com', 'password'

)

);

}

/\*\*

* + @test

\*/

public function afterUserSignUpItShouldBeInTheRepository()

{

$user = $this->executeSignUp();

$this->assertSame(

$user,

$this->userRepository->ofId($user->id())

);

}

}

We've used an in-memory implementation for the User Repository. This is what is called a Fake: a fully functional implementation for the Repository that will make our test work as a unit. We don't need to go to the database to test the behavior of this class. That would make our test slow and fragile.

Checking for a Domain Events submission might be interesting too. If creating a user fires a user registered event, ensuring it's been triggered might be a good idea:

class SignUpUserServiceTest extends \PHPUnit\_Framework\_TestCase

{

// ...

/\*\*

* + @test

\*/

public function itShouldPublishUserRegisteredEvent()

{

$subscriber = new SpySubscriber();

$id = DomainEventPublisher::instance()->subscribe($subscriber);

$user = $this->executeSignUp();

$userId = $user->id();

DomainEventPublisher::instance()->unsubscribe($id);

$this->assertUserRegisteredEventPublished(

$subscriber, $userId

);

}

private function assertUserRegisteredEventPublished(

$subscriber, $userId

) {

$this->assertInstanceOf(

'UserRegistered', $subscriber->domainEvent

);

$this->assertTrue(

$subscriber->domainEvent->userId()->equals($userId)

);

}

}

class SpySubscriber implements DomainEventSubscriber

{

public $domainEvent;

public function handle($aDomainEvent)

{

$this->domainEvent = $aDomainEvent;

}

public function isSubscribedTo($aDomainEvent)

{

return true;

}

}

## Transactions

Transactions are an implementation detail related to the persistence mechanism. The Domain layer shouldn't be aware of this low-level implementation detail. Thinking about beginning, committing, or rolling back a transaction at this level is a big smell. This level of detail belongs to the Infrastructure layer.

The best way of handling transactions is to not handle them at all. We could wrap our Application Services with a Decorator implementation for handling the transaction session automatically.

We've implemented a solution to this problem in one of our repositories, and you can check it out [here](https://github.com/dddinphp/ddd):

interface TransactionalSession

{

/\*\*

* + @return mixed

\*/

public function executeAtomically(callable $operation);

}

This contract takes a piece of code and executes it atomically. Depending on your persistence mechanism, you'll end up with different implementations.

Let's see how we could do it with Doctrine ORM:

class DoctrineSession implements TransactionalSession

{

private $entityManager;

public function construct(EntityManager $entityManager)

{

$this->entityManager = $entityManager;

}

public function executeAtomically(callable $operation)

{

return $this->entityManager->transactional($operation);

}

}

This is how a client would use the previous code:

/\*\* @var EntityManager $em \*/

$nonTxApplicationService = new SignUpUserService(

$em->getRepository('BoundedContext\Domain\Model\User\User')

);

$txApplicationService = new TransactionalApplicationService(

$nonTxApplicationService, new DoctrineSession($em)

);

$response = $txApplicationService->execute( new SignUpUserRequest(

'user@example.com', 'password'

)

);

Now that we have the Doctrine implementation for transactional sessions, it would be great to create a Decorator for our Application Services. With this approach, we make transactional requests transparent to the Domain:

class TransactionalApplicationService implements ApplicationService

{

private $session; private $service;

public function construct(

ApplicationService $service, TransactionalSession $session

) {

$this->session = $session;

$this->service = $service;

}

public function execute(BaseRequest $request)

{

$operation = function () use ($request) { return $this->service->execute($request);

};

return $this->session->executeAtomically($operation);

}

}

A nice side effect of using Doctrine Session is that it automatically manages the flush method, so you don't need to add the flush inside your Domain or Infrastructure.

## Security

In case you're wondering how to manage and handle user credentials and security in general, unless it's the responsibility of your Domain, we recommend letting the framework handle it. The user session is a concern of the delivery mechanism. Polluting the Domain with such concepts will make it harder to develop.

## 域名事件

在执行应用程序服务之前，必须配置域事件侦听器，否则将不会注意到任何人。 在执行应用程序服务之前，有些情况下您必须显式地配置侦听器:

// ...

$subscriber = new SpySubscriber(); DomainEventPublisher::instance()->subscribe($subscriber);

$applicationService = // ...

$applicationService->execute(...);

大多数情况下，这将通过配置依赖注入容器来完成。

## Command Handlers

An interesting way of executing Application Services is through a Command Bus library. A good one is [Tactician](https://tactician.thephpleague.com/). From the Tactician website:

What is a Command Bus? The term is mostly used when we combine the [Command](https://en.wikipedia.org/wiki/Command_pattern) [pattern](https://en.wikipedia.org/wiki/Command_pattern) with a [service layer](http://martinfowler.com/eaaCatalog/serviceLayer.html). Its job is to take a Command object (which describes what the user wants to do) and match it to a Handler (which executes it). This can help structure your code neatly.

— our Application Services are the Service Layer, and our Request objects look pretty much like Commands.

Fair enough — our Application Services are the Service Layer, and our Request objects look pretty much like Commands. Wouldn't it be great if we had a mechanism to link all the Application Services, and then based on the Request, execute the correct one? Well, that's actually what a Command Bus is.

### 战术图书馆和其他选项

Tactician is a Command Bus library, which allows you to use the Command pattern for your Application Services. It's especially convenient for Application Services, but you could use any kind of input.

我们来看一个来自Tactician网站的例子:

// You build a simple message object like this: class PurchaseProductCommand

{

protected $productId; protected $userId;

// ...and constructor to assign those properties...

}

// And a Handler class that expects it: class PurchaseProductHandler

{

public function handle(PurchaseProductCommand $command)

{

// use command to update your models, etc

}

}

// And then in your Controllers, you can fill in the command using your

favorite

// form or serializer library, then drop it in the CommandBus and you're done!

$command = new PurchaseProductCommand(42, 29);

$commandBus->handle($command);

That's it. Tactician is the $commandBus Service. It does all the plumbing for finding the right handler and method, which can avoid a lot of boilerplate code. Here, Commands and Handlers are just normal classes, but you can configure whichever one fits your app better.

In summary, we can conclude that Commands are just Request objects, and Command Handlers are just Application Services.

A cool thing about Tactician (and Command Buses in general) is that they're really easy to extend. Tactician provides plug-ins for common tasks, like logging and database transactions. That way, you can forget about setting up the wiring on every handler.

Another interesting plug-in for [Tactician is Bernard](http://bernard.readthedocs.org/) integration. Bernard is an asynchronous job queue that allows you to leave some tasks for later processing. Heavy processes block the response. Most of the time, we can branch and delay their execution for later. For the best experience, answer the customer as fast as possible and let them know once the branched processes are done.

Matthias Noback has developed another similar project, called [SimpleBus](http://simplebus.github.io/MessageBus/), that can be used as an alternative to Tactician. The main difference is that SimpleBus Command Handlers don't have a return value.

## Wrap-Up

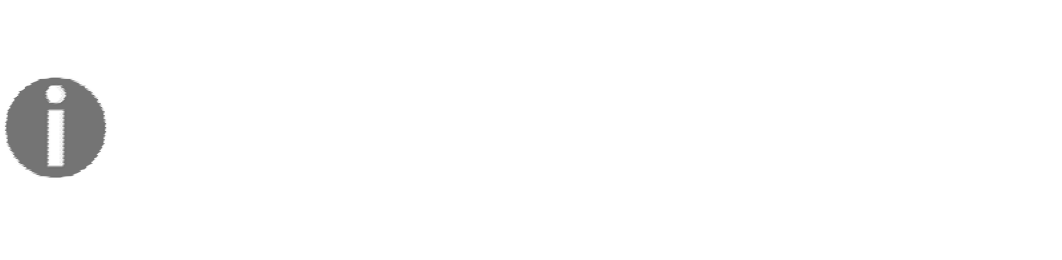
Application Services represent the Application layer of your Bounded Context. These high- level use cases should be relatively simple and skinny, as their purpose evolves around Domain coordination. Application Services are the entry point for Domain logic interaction. We've seen that Requests and Commands keep things organized; that DTOs and Data Transformers allow us to decouple data representation from Domain conceptualization; that building Application Services is pretty straightforward with Dependency Injection Containers; and that we have plenty of options for combining Application Services in complex layouts.

# 集成有界上下文

****

Every enterprise application is typically composed of several areas in which the company operates. Areas such as *billing*, *inventory*, *shipping management*, *catalog*, and so on are common examples. The easiest manner in which to manage all these concerns may seem to lean toward a **monolithic system**. But, you might wonder, does it have to be this way?

What if any friction garnered between teams working on these separate areas could be reduced by splitting this big monolithic application into smaller, independent chunks? In this chapter, we'll explore how to do this, so be prepared for insights and heuristics around **strategical design**.



**Dealing with Distributed Systems**

Dealing with distributed systems is **hard**. Breaking a system into independent autonomous parts has its benefits, but it also increases complexity. For example, the coordination and synchronization of distributed systems is not trivial, and as a result, should be considered carefully. As Martin Fowler said in the [PoEAA](https://www.martinfowler.com/books/eaa.html) book, the first law of distributed systems is always: **Don't distribute**.

## 通过数据存储集成

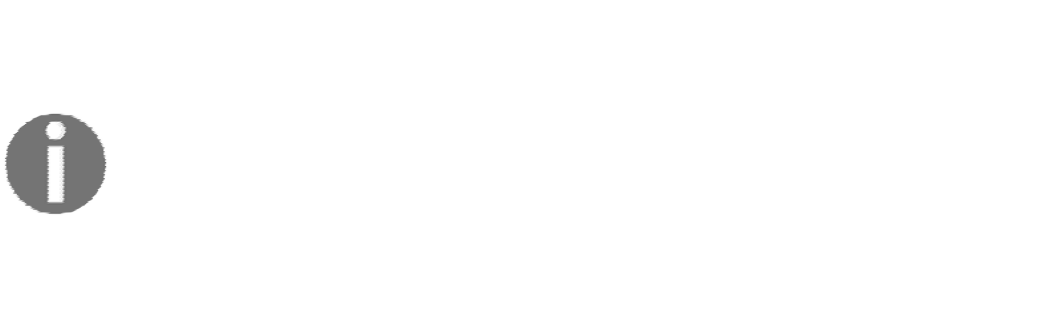
集成应用程序不同部分的最常用技术之一一直是共享相同的数据存储以及相同的代码库。 这通常被称为单一应用程序，并且通常最终只有一个数据存储，它承载了与应用程序中所有关注点相关的数据.

Consider an e-commerce application. A shared data store would contain all concerns (Example: tables within a relational database) surrounding the catalog, billing, inventory, and so on. There's nothing wrong with this approach per se—for example, in small linear applications where the complexity is not too high. However, within complex Domains, some issues can arise. If you share data across many tables touching multiple application concerns, transactions will have a big impact on performance.

Another less technical problem that could develop is in regard to the Ubiquitous Language. The main advantage of the separation of Bounded Contexts is having **a single Ubiquitous Language for each one**. In doing so, models will be separated into their own Contexts.

Mixing all models together within the same Context can lead to ambiguity and confusion.

Going back to the e-commerce system, imagine we want to introduce the concept of a t- shirt. Within the catalogue Context, a t-shirt would be a *product* with properties like *color*, *size*, *material*, and maybe some fancy *pictures*. In the *inventory* system, however, we don't really want to concern ourselves with these things. Here, a *product* has a different meaning, where we care about different properties like *weight*, *location in the warehouse*, or *dimensions*. Mixing both Contexts together will tangle concepts and complicate the design. In Domain- Driven Design terms, mixing concepts in this manner is what is called a Shared Kernel.



**Shared Kernel**

Designate some subset of the domain model that the teams agree to share. Of course this includes, along with this subset of the model, the subset of code or of the database design associated with that part of the model. This explicitly shared stuff has special status, and shouldn't be changed without consultation with the other team. Integrate a functional system frequently, but somewhat less often than the pace of CONTINUOUS INTEGRATION within the teams. At these integrations, run the tests of both teams. Eric Evans - [Domain-Driven Design: Tackling Complexity](https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215) [in the Heart of Software](https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215)

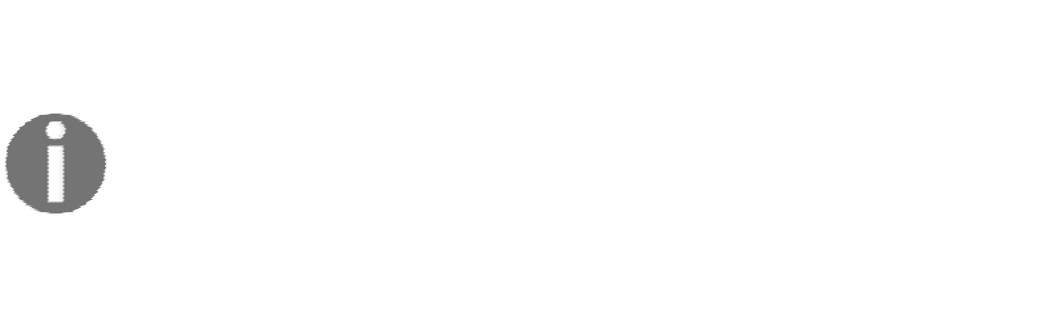
We don't recommend using a Shared Kernel, as multiple teams can collide within the development of it, which not only results in maintenance issues but also becomes a point of friction. However, if you opt to use a Shared Kernel, changes should be agreed upon beforehand and between all parties involved. Conceptually, this approach has other problems, such as people seeing it as a bag to place *stuff* that doesn't belong anywhere else, and this grows indefinitely. A better way of dealing with the ever-growing complexity of the monolith is to break it up in different autonomous pieces, such as communicating through REST, RPC, or messaging systems. This requires drawing clear boundaries, with each Context likely ending up with its own Infrastructure—data stores, servers, messaging middleware, and so on — and even its own team.

As you might imagine, this could lead to some degree of duplication, but that's a tradeoff that we're willing to make in order to reduce complexity. In Domain-Driven Design, we call these independent pieces **Bounded Contexts**.

## 整合关系

### 客户 - 供应商

When there's a unidirectional integration between two Bounded Contexts, where one acts as a provider (**upstream**) and the other as a client (**downstream**), we'll end up with **Customer - Supplier Development Teams**.

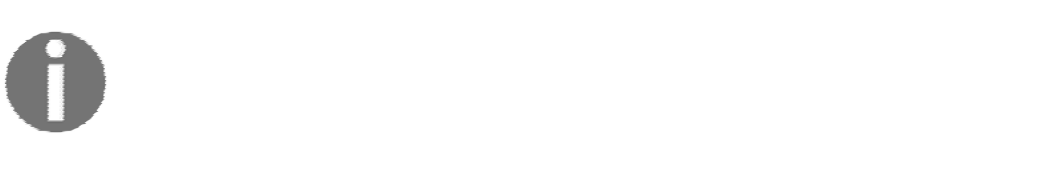


Establish a clear customer/supplier relationship between the two teams. In planning sessions, make the downstream team play the customer role to the upstream team. Negotiate and budget tasks for downstream requirements so that everyone understands the commitment and schedule. Jointly develop automated acceptance tests that will validate the interface expected. Add these tests to the upstream team's test suite, to be run as part of its' continuous integration. This testing will free the upstream team to make changes without fear of side effects downstream. Eric Evans - [Domain-Driven Design: Tackling Complexity in the Heart of](https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215) [Software](https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215).

Customer - Supplier Development Teams are the most common way of integrating Bounded Contexts and usually represent a win-win situation when teams work closely.

### 独立的方式

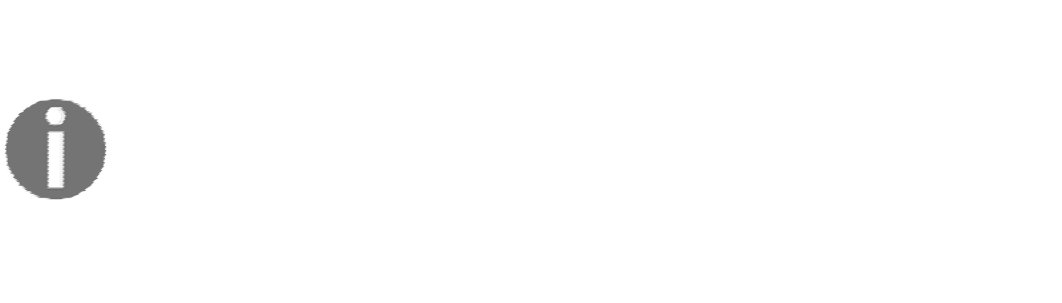
Continuing with the e-commerce example, think about reporting revenue to an old legacy retailer financial system. The integration could be incredibly expensive, resulting in it not being worth the effort to implement. In Domain-Driven Design strategic terms, this is known as **Separate Ways**.



Integration is always expensive. Sometimes the benefit is small. So Declare a BOUNDED CONTEXT to have no connection to the others at all, allowing developers to find simple, specialized solutions within this small scope. Eric Evans - *Domain-Driven Design:* [Tackling Complexity in the](https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215) [Heart of Software](https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215).

### Conformist

Consider again the e-commerce example and integration with a third-party shipping service. Both Domains differ in models, teams, and Infrastructure. The team responsible for maintaining the third-party shipping service will not participate in your product planning or provide any solutions to the e-commerce system. These teams don't have a close relationship. We could choose to accept and *conform* to their Domain Model. In strategic design, this is what we call a **Conformist Integration**.



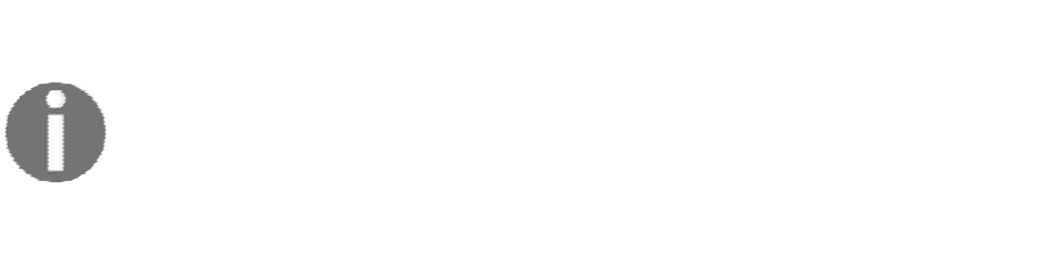
Eliminate the complexity of translation between BOUNDED CONTEXTS by slavishly adhering to the model of the upstream team. Although this cramps the style of the downstream designers and probably does not yield the ideal model for the application, choosing CONFORMITY enormously simplifies integration. Also, you will share a UBIQUITOUS LANGUAGE with your supplier team. The supplier is in the driver's seat, so it is good to make communication easy for them. Altruism may be sufficient to get them to share information with you. Eric Evans - *Domain-Driven Design:* [Tackling Complexity in the Heart of Software](https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215).

## 实现有界上下文集成

To make things easier, we'll assume Bounded Contexts have a Customer - Supplier relationship.

### 现代RPC

With modern RPC, we refer to RPC through RESTful resources. A Bounded Context reveals a clear interface to interact with to the outside world. It exposes resources that could be manipulated through HTTP verbs. We could say that the Bounded Context offers a set of services and operations. In strategical terms, this is what is called an **Open Host Service**.



**Open Host Service**

Define a protocol that gives access to your subsystem as a set of SERVICES. Open the protocol so that all who need to integrate with you can use it. Enhance and expand the protocol to handle new integration requirements, except when a single team has idiosyncratic needs. Then, use a one-off translator to augment the protocol for that special case so that the shared protocol can stay simple and coherent. Eric Evans -

*Domain-Driven Design:* [Tackling Complexity in the Heart of Software](https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215)*.*

Let's explore an example provided within the [Last Wishes application](https://github.com/dddinphp/last-wishes) that comes with this book's GitHub organization.

The application is a web platform with the purpose of letting people save their last wills before they die. There are two Contexts: one responsible for handling wills—the Will Bounded Context—and one in charge of giving points to the users of the system—the [Gamification Context](https://github.com/dddinphp/last-wishes-gamify). In the Will Context, the user could have badges related to the number of points the user made on the Gamification Context. This means that we need to integrate both Contexts together in order to show the badges a user has on the Will Context.

The Gamification Context is a full-fledged event-driven application powered by a custom event sourcing engine. It's a full-stack Symfony application that uses [FOSRestBundle](http://symfony.com/doc/current/bundles/FOSRestBundle/index.html), [BazingaHateoasBundle](https://github.com/willdurand/BazingaHateoasBundle), [JMSSerializerBundle](https://github.com/schmittjoh/JMSSerializerBundle), [NelmioApiDocBundle](https://github.com/schmittjoh/JMSSerializerBundle), and [OngrElasticsearchBundle](https://github.com/schmittjoh/JMSSerializerBundle) to provide a level 3 and up REST API (commonly known as the Glory of REST), according to the [Richardson Maturity *Mo*del](https://martinfowler.com/articles/richardsonMaturityModel.html). All the Events triggered within this Context are projected against an Elasticsearch server, in order to produce the data needed for the views. We'll expose the number of points made for a given user through an endpoint like [http://gamification.context.host/api/users/{id}](http://gamification.context.host/api/users/).

We'll also fetch the user projection from Elasticsearch and serialize it to a format previously negotiated with the client:

namespace AppBundle\Controller;

use FOS\RestBundle\Controller\Annotations as Rest; use FOS\RestBundle\Controller\FOSRestController; use Nelmio\ApiDocBundle\Annotation\ApiDoc;

class UsersController extends FOSRestController

{

/\*\*

* @ApiDoc(
* resource = true,
* description = "Finds a user given a user ID",
* statusCodes = {
* 200 = "Returned when the user have been found",
* 404 = "Returned when the user could not be found"

\* }

\* )

\*

* @Rest\View(
* statusCode = 200

\* )

\*/

public function getUserAction($id)

{

$repo = $this->get('es.manager.default.user');

$user = $repo->find($id);

if (!$user) {

throw $this->createNotFoundException( sprintf(

'A user with an ID of %s does not exist',

$id

)

);

}

return $user;

}

}

As we explained in the [Chapter 2](#_bookmark21), *Architectural Styles* reads are treated as an Infrastructure concern, so there's no need to wrap them inside a Command / Command Handler flow.

The resulting JSON+HAL representation of a user will be like this:

{

"id": "c3c587c6-610a-42df", "points": 0,

"\_links": {

"self": {

"href":

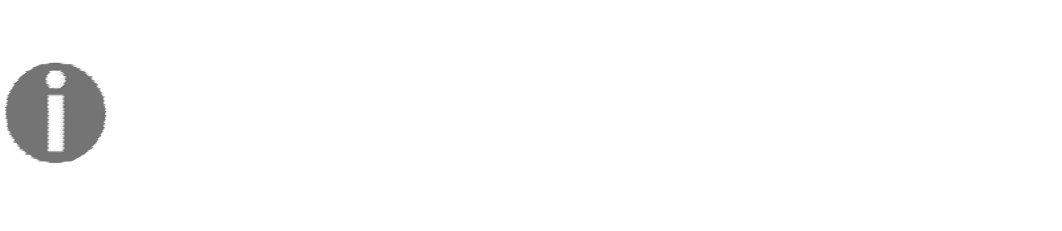
["http://gamification.ctx/api/users/c3c587c6-610a-42df"](http://gamification.ctx/api/users/c3c587c6-610a-42df)

}

}

}

Now we're in a good position to integrate both Contexts. We just need to write the client in the Will Context for consuming the endpoint we've just created. Should we mix both Domain Models? Digesting the Gamification Context directly will mean adapting the Will Context to the Gamification one, resulting in a **Conformist** integration. However, separating these concerns seems worth the effort. We need a layer for guaranteeing the integrity and the consistency of the Domain Model within the Will Context, and we need to translate *points* (Gamification) to *badges* (Will). In Domain-Driven Design, this translation mechanism is what's called an **Anti-Corruption layer**.



**Anti-Corruption Layer**

Create an isolating layer to provide clients with functionality in terms of their own domain model. The layer talks to the other system through its existing interface, requiring little or no modification to the other system. Internally, the layer translates in both directions as necessary between the two models. Eric Evans - *Domain-Driven Design:* [Tackling Complexity in](https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215) [the Heart of Software.](https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215)

So, what does the Anti-Corruption layer look like? Most of the time, Services will be interacting with a combination of Adapters and Facades. The Services encapsulate and hide the low-level complexities behind these transformations. Facades aid in hiding and encapsulating access details required for fetching data from the Gamification model.

Adapters translate between models, often using specialized Translators.

Let's see how to define a User Service within the Will's model that will be responsible for retrieving the badges earned by a given user:

namespace Lw\Domain\Model\User;

interface UserService

{

public function badgesFrom(UserId $id);

}

Now let's look at the implementation on the Infrastructure side. We'll use an adapter for the transformation process:

namespace Lw\Infrastructure\Service;

use Lw\Domain\Model\User\UserId;

use Lw\Domain\Model\User\UserService;

class TranslatingUserService implements UserService

{

private $userAdapter;

public function construct(UserAdapter $userAdapter)

{

$this->userAdapter = $userAdapter;

}

public function badgesFrom(UserId $id)

{

return $this->userAdapter->toBadges($id);

}

}

And here's the HTTP implementation for the UserAdapter:

namespace Lw\Infrastructure\Service; use GuzzleHttp\Client;

class HttpUserAdapter implements UserAdapter

{

private $client;

public function construct(Client $client)

{

$this->client = $client;

}

public function toBadges( $id)

{

$response = $this->client->get( sprintf('/users/%s', $id), [

'allow\_redirects' => true, 'headers' => [

'Accept' => 'application/hal+json'

]

]

);

$badges = [];

if (200 === $response->getStatusCode()) {

$badges =

(new UserTranslator())

->toBadgesFromRepresentation( json\_decode(

$response->getBody(), true

)

);

}

return $badges;

}

}

As you can see, the Adapter acts as a **Facade to the Gamification Context** too. We did it this way, as fetching the User resource on the Gamification side is pretty straightforward. The Adapter uses the UserTranslator to perform the translation:

namespace Lw\Infrastructure\Service;

use Lw\Infrastructure\Domain\Model\User\FirstWillMadeBadge; use Symfony\Component\PropertyAccess\PropertyAccess;

class UserTranslator

{

public function toBadgesFromRepresentation($representation)

{

$accessor = PropertyAccess::createPropertyAccessor();

$points = $accessor->getValue($representation, 'points');

$badges = [];

if ($points > 3) {

$badges[] = new FirstWillMadeBadge();

}

return $badges;

}

}

The Translator specializes in transforming the points coming from the Gamification Context into badges.

We've shown how to integrate two Bounded Contexts where respective teams share a **Customer-Supplier** relationship. The Gamification Context exposes the integration through an **Open Host Service** implemented by a RESTful protocol. On the other side, the Will Context consumes the service through an **Anti-Corruption layer** responsible for translating the model from one Domain to the other, ensuring the Will Context's integrity.

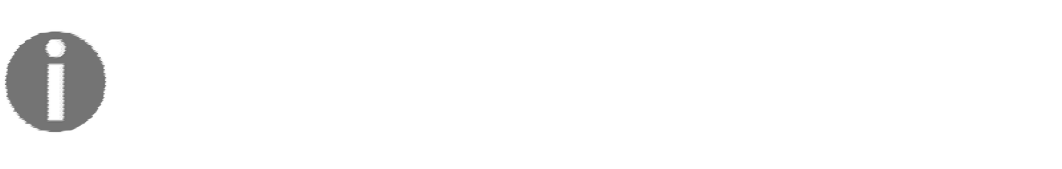
### 消息队列

RESTful资源不是启用有界上下文之间集成的唯一方式。 正如我们将看到的，消息传递中间件支持不同上下文之间的分离集成.

继续使用Last Wishes应用程序，我们刚刚实现了两个团队之间的单向关系，以在各自的上下文中管理点和徽章。 但是，我们在目的范围内留下了一个重要功能：每次他们发出愿望时都会奖励用户.

我们可以通过拉动战略进入另一个开放主机服务。 意志上下文将定期拉动游戏化上下文以获得徽章同步（例如：通过像Cron这样的调度器）。 该解决方案将影响用户的体验，并会浪费大量不必要的资源.

更好的方法是使用消息传递中间件。 通过这个解决方案，上下文可以将消息推送到中间件（通常是消息队列）。 有兴趣的人可以按需求订阅，检查和消费信息。 为了做到这一点，我们需要一种专门的，共享的和通用的通信语言，以便所有各方都能理解所传输的信息。 这就是所谓的发布语言.



**发布语言**

使用记录良好的共享语言，可以将必要的域信息表达为通信媒介，并根据需要翻译为该语言。 埃里克埃文斯 - 领域驱动设计：解决软件中心的复杂性.

在思考这些消息的格式并仔细观察我们的领域模型时，我们意识到我们已经拥有了我们需要的东西：第6章，域 - 事件。 没有必要定义有界上下文之间的新的通信方式。 相反，我们可以使用域事件来定义跨上下文的通用语言。 域专家关心的事情的定义恰好与我们正在寻找的内容完全吻合：正式的发布语言.

在我们的例子中，我们可以使用RabbitMQ作为消息中间件。 这可能是那里最可靠和最健壮的消息AMQP协议之一。 我们还将包含广泛使用的PHP库php-amqplib和RabbitMQBundle.

让我们从意志上下文开始，因为它是在用户注册或者希望时触发事件的那个。 正如我们在第6章“域事件”中已经看到的那样，将域事件存储到持久机制中是一个好主意，所以我们会假设这是做完的。 我们需要一个消息发布者来将事件存储中存储的域事件提取并发布到消息传递中间件。 我们已经在第6章域名事件中与RabbitMQ进行了整合，所以我们只需要在游戏化上下文中实现代码。 我们将监听由意志上下文触发的事件。 在我们使用Symfony框架时，我们利用了名为RabbitMQBundle的Symfony包.

我们为“用户注册”和“愿意为人”活动定义了两个消息使用者:

namespace AppBundle\Infrastructure\Messaging\PhpAmqpLib;

use Lw\Gamification\Command\SignupCommand;

use OldSound\RabbitMqBundle\RabbitMq\ConsumerInterface; use PhpAmqpLib\Message\AMQPMessage;

class PhpAmqpLibLastWillUserRegisteredConsumer implements ConsumerInterface

{

private $commandBus;

public function construct($commandBus)

{

$this->commandBus = $commandBus;

}

public function execute(AMQPMessage $message)

{

$type = $message->get('type');

if('Lw\Domain\Model\User\UserRegistered' === $type) {

$event = json\_decode($message->body);

$eventBody = json\_decode($event->event\_body);

$this->commandBus->handle(

new SignupCommand($eventBody->user\_id->id)

);

return true;

}

return false;

}

}

Note that in this case, we're only processing messages with the type of

Lw\Domain\Model\User\UserRegistered:

namespace AppBundle\Infrastructure\Messaging\PhpAmqpLib;

use Lw\Gamification\Command\RewardUserCommand;

use Lw\Gamification\Domain\Model\AggregateDoesNotExist; use OldSound\RabbitMqBundle\RabbitMq\ConsumerInterface; use PhpAmqpLib\Message\AMQPMessage;

class PhpAmqpLibLastWillWishWasMadeConsumer implements ConsumerInterface

{

private $commandBus;

public function construct($commandBus)

{

$this->commandBus = $commandBus;

}

public function execute(AMQPMessage $message)

{

$type = $message->get('type');

if ('Lw\Domain\Model\Wish\WishWasMade' === $type) {

$event = json\_decode($message->body);

$eventBody = json\_decode($event->event\_body);

try {

$points = 5;

$this->commandBus->handle( new RewardUserCommand(

$eventBody->user\_id->id,

$points

)

);

} catch (AggregateDoesNotExist $e) {

// Noop

}

return true;

}

return false;

}

}

Again, we're only interested in tracking Lw\Domain\Model\Wish\WishWasMade events.

In both cases, we use a Command Bus, which we discussed in the Chapter, Application. However, we can summarize it as a highway that decouples the Command and Receiver. The **when** and **how** a Command is executed is independent from **who** triggered it.

The Gamification Context uses [Tactician](http://tactician.thephpleague.com/) (and [TacticianBundle](https://github.com/thephpleague/tactician-bundle)), a simple Command Bus that can be extended and adapted to your system. So now we're almost ready to start consuming Events from the Will Context.

The only thing we still need to do is define the RabbitMQBundle configuration in Symfony's

config.yml file:

services:

last\_will\_user\_registered\_consumer: class:

AppBundle\Infrastructure\Messaging\ PhpAmqpLib\PhpAmqpLibLastWillUserRegisteredConsumer

arguments:

- @tactician.commandbus

last\_will\_wish\_was\_made\_consumer: class:

AppBundle\Infrastructure\Messaging\ PhpAmqpLib\PhpAmqpLibLastWillWishWasMadeConsumer

arguments:

- @tactician.commandbus

old\_sound\_rabbit\_mq: connections:

default:

host: " %rabbitmq\_host%" port: " %rabbitmq\_port%" user: " %rabbitmq\_user%"

password: " %rabbitmq\_password%" vhost: " %rabbitmq\_vhost%"

lazy: true

consumers:

last\_will\_user\_registered: connection: default

callback: last\_will\_user\_registered\_consumer

exchange\_options: name: last-will type: fanout

queue\_options: name: last-will

last\_will\_wish\_was\_made: connection: default

callback: last\_will\_wish\_was\_made\_consumer

exchange\_options: name: last-will type: fanout

queue\_options: name: last-wil

The most convenient RabbitMQ configuration is probably the [[Publish /](https://www.rabbitmq.com/tutorials/tutorial-three-php.html)

[Subscribe](https://www.rabbitmq.com/tutorials/tutorial-three-php.html)] pattern. All messages published by the Will Context will be delivered to all connected consumers. This is called **fanout** in the RabbitMQ exchange configuration.

The exchange consists of an agent being in charge of delivering messages to the corresponding queues:

> php app/console rabbitmq:consumer --messages=1000 last\_will\_user\_registered

> php app/console rabbitmq:consumer --messages=1000 last\_will\_wish\_was\_made

With those two commands, Symfony will execute both consumers and they'll start listening for Domain Events. We've specified a limit of 1,000 messages to consume, as PHP isn't the best platform for executing long-running processes. It also might be a good idea to use something like [Supervisor](http://supervisord.org/) to monitor and restart processes periodically.

## 包起来

虽然我们只看到其中的一小部分，但战略设计是领域驱动设计的核心和灵魂。 这是帮助开发更好和更多语义模型的重要部分。 我们建议使用消息传递中间件来集成有界上下文，因为这自然会导致更简单，分离和事件驱动的体系结构.

# 六边形建筑与PHP

*The following article was posted in php¦architect magazine in June 2014 by Carlos* *Buenosvinos.*

## Introduction

With the rise of **Domain-Driven Design** (**DDD**), architectures promoting domain centric designs are becoming more popular. This is the case with **Hexagonal Architecture**, also known as **Ports and Adapters**, that seems to have being rediscovered just now by PHP developers. Invented in 2005 by Alistair Cockburn, one of the Agile Manifesto authors, the Hexagonal Architecture allows an application to be equally driven by users, programs, automated tests or batch scripts, and to be developed and tested in isolation from its eventual run-time devices and databases. This results into agnostic infrastructure web applications that are easier to test, write and maintain. Let's see how to apply it using real PHP examples.

Your company is building a brainstorming system called *Idy*. Users add and rate ideas so the most interesting ones can be implemented in a company. It is Monday morning, another sprint is starting and you are reviewing some user stories with your team and your Product Owner. **As a not logged in user, I want to rate an idea and the author should be notified by email**, that's a really important one, isn't it?

## 第一种方法

As a good developer, you decide to divide and conquer the user story, so you'll start with the first part, *I want to rate an idea*. After that, you will face *the author should be notified by email*. That sounds like a plan.

In terms of business rules, rating an idea is as easy as finding the idea by its identifier in the ideas repository, where all the ideas live, add the rating, recalculate the average and save the idea back. If the idea does not exist or the repository is not available we should throw an exception so we can show an error message, redirect the user or do whatever the business asks us for.

In order to *execute* this *UseCase*, we just need the idea identifier and the rating from the user. Two integers that would come from the user request.

Your company web application is dealing with a Zend Framework 1 legacy application. As most of companies, probably some parts of your app may be newer, more SOLID, and others may just be a big ball of mud. However, you know that it does not matter at all which framework you are using, it is all about writing clean code that makes maintenance a low cost task for your company.

You're trying to apply some Agile principles you remember from your last conference, how it was, yeah, I remember "make it work, make it right, make it fast". After some time working you get something like Listing 1.

class IdeaController extends Zend\_Controller\_Action

{

public function rateAction()

{

// Getting parameters from the request

$ideaId = $this->request->getParam('id');

$rating = $this->request->getParam('rating');

// Building database connection

$db = new Zend\_Db\_Adapter\_Pdo\_Mysql([ 'host' => 'localhost', 'username' => 'idy',

'password' => '', 'dbname' => 'idy'

]);

// Finding the idea in the database

$sql = 'SELECT \* FROM ideas WHERE idea\_id = ';

$row = $db->fetchRow($sql, $ideaId); if (!$row) {

throw new Exception('Idea does not exist');

}

// Building the idea from the database

$idea = new Idea();

$idea->setId($row['id']);

$idea->setTitle($row['title']);

$idea->setDescription($row['description']);

$idea->setRating($row['rating']);

$idea->setVotes($row['votes']);

$idea->setAuthor($row['email']);

// Add user rating

$idea->addRating($rating);

// Update the idea and save it to the database

$data = [

'votes' => $idea->getVotes(), 'rating' => $idea->getRating()

];

$where['idea\_id = '] = $ideaId;

$db->update('ideas', $data, $where);

// Redirect to view idea page

$this->redirect('/idea/' . $ideaId);

}

}

I know what readers are thinking: *Who is going to access data directly from the controller? This is a 90's example!*, ok, ok, you're right. If you are already using a framework, it is likely that you are also using an ORM. Maybe done by yourself or any of the existing ones such as Doctrine, Eloquent, Zend, and so on. If this is the case, you are one step further from those who have some Database connection object but don't count your chickens before they're hatched.

For newbies, Listing 1 code just works. However, if you take a closer look at the Controller, you'll see more than business rules, you'll also see how your web framework routes a request into your business rules, references to the database or how to connect to it. So close, you see references to your **infrastructure**.

Infrastructure is the **detail that makes your business rules work**. Obviously, we need some way to get to them (API, web, console apps, and so on.) and effectively we need some physical place to store our ideas (memory, database, NoSQL, and so on.). However, we should be able to exchange any of these pieces with another that behaves in the same way but with different implementations. What about starting with the Database access?

All those Zend\_DB\_Adapter connections (or straight MySQL commands if that's your case) are asking to be promoted to some sort of object that encapsulates fetching and persisting Idea objects. They are begging for being a Repository.

## 存储库和持久性边缘

Whether there is a change in the business rules or in the infrastructure, we must edit the same piece of code. Believe me, in CS, you don't want many people touching the same piece of code for different reasons. Try to make your functions do one and just one thing so it is less probable having people messing around with the same piece of code. You can learn more about this by having a look at the **Single Responsibility Principle** (**SRP**). For more information about this principle: [http://www.objectmentor.com/resources/articles/sr](http://www.objectmentor.com/resources/articles/srp.pdf) [p.pdf](http://www.objectmentor.com/resources/articles/srp.pdf)

Listing 1 is clearly this case. If we want to move to Redis or add the author notification feature, you'll have to update the rateAction method. Chances to affect aspects of the rateAction not related with the one updating are high. Listing 1 code is fragile. If it is common in your team to hear *If it works, don't touch it*, SRP is missing.

So, we must decouple our code and encapsulate the responsibility for dealing with fetching and persisting ideas into another object. The best way, as explained before, is using a Repository. Challenged accepted! Let's see the results in Listing 2:

class IdeaController extends Zend\_Controller\_Action

{

public function rateAction()

{

$ideaId = $this->request->getParam('id');

$rating = $this->request->getParam('rating');

$ideaRepository = new IdeaRepository();

$idea = $ideaRepository->find($ideaId); if (!$idea) {

throw new Exception('Idea does not exist');

}

$idea->addRating($rating);

$ideaRepository->update($idea);

$this->redirect('/idea/' . $ideaId);

}

}

class IdeaRepository

{

private $client;

public function construct()

{

$this->client = new Zend\_Db\_Adapter\_Pdo\_Mysql([ 'host' => 'localhost',

'username' => 'idy', 'password' => '', 'dbname' => 'idy'

]);

}

public function find($id)

{

$sql = 'SELECT \* FROM ideas WHERE idea\_id = ';

$row = $this->client->fetchRow($sql, $id); if (!$row) {

return null;

}

$idea = new Idea();

$idea->setId($row['id']);

$idea->setTitle($row['title']);

$idea->setDescription($row['description']);

$idea->setRating($row['rating']);

$idea->setVotes($row['votes']);

$idea->setAuthor($row['email']);

return $idea;

}

public function update(Idea $idea)

{

$data = [

'title' => $idea->getTitle(), 'description' => $idea->getDescription(), 'rating' => $idea->getRating(),

'votes' => $idea->getVotes(), 'email' => $idea->getAuthor(),

];

$where = ['idea\_id = ' => $idea->getId()];

$this->client->update('ideas', $data, $where);

}

}

The result is nicer. The rateAction of the IdeaController is more understandable. When read, it talks about business rules. IdeaRepository is a **business concept**. When talking with business guys, they understand what an IdeaRepository is: A place where I put Ideas and get them.

A Repository *mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects*. as found in Martin Fowler's pattern catalog.

If you are already using an ORM such as Doctrine, your current repositories extend from an EntityRepository. If you need to get one of those repositories, you ask Doctrine EntityManager to do the job. The resulting code would be almost the same, with an extra access to the EntityManager in the controller action to get the IdeaRepository.

At this point, we can see in the landscape one of the edges of our hexagon, the *persistence* edge. However, this side is not well drawn, there is still some relationship between what an IdeaRepository is and how it is implemented.

In order to make an effective separation between our *application boundary* and the *infrastructure boundary* we need an additional step. We need to explicitly decouple behavior from implementation using some sort of interface.

## 解耦业务和持久性

Have you ever experienced the situation when you start talking to your Product Owner, Business Analyst or Project Manager about your issues with the Database? Can you remember their faces when explaining how to persist and fetch an object? They had no idea what you were talking about.

The truth is that they don't care, but that's ok. If you decide to store the ideas in a MySQL server, Redis or SQLite it is your problem, not theirs. Remember, from a business standpoint, **your infrastructure is a detail**. Business rules are not going to change whether you use Symfony or Zend Framework, MySQL or PostgreSQL, REST or SOAP, and so on.

That's why it is important to decouple our IdeaRepository from its implementation. The easiest way is to use a proper interface. How can we achieve that? Let's take a look at Listing 3.

class IdeaController extends Zend\_Controller\_Action

{

public function rateAction()

{

$ideaId = $this->request->getParam('id');

$rating = $this->request->getParam('rating');

$ideaRepository = new MySQLIdeaRepository();

$idea = $ideaRepository->find($ideaId); if(!$idea) {

throw new Exception('Idea does not exist');

}

$idea->addRating($rating);

$ideaRepository->update($idea);

$this->redirect('/idea/' . $ideaId);

}

}

interface IdeaRepository

{

/\*\*

* @param int $id
* @return null|Idea

\*/

public function find($id);

/\*\*

* @param Idea $idea

\*/

public function update(Idea $idea);

}

class MySQLIdeaRepository implements IdeaRepository

{

// ...

}

Easy, isn't it? We have extracted the IdeaRepository behavior into an interface, renamed the IdeaRepository into MySQLIdeaRepository and updated the rateAction to use our MySQLIdeaRepository. But what's the benefit?

We can now exchange the repository used in the controller with any implementing the same interface. So, let's try a different implementation.

## 将我们的坚持迁移到Redis

During the sprint and after talking to some mates, you realize that using a NoSQL strategy could improve the performance of your feature. Redis is one of your best friends. Go for it and show me your Listing 4:

class IdeaController extends Zend\_Controller\_Action

{

public function rateAction()

{

$ideaId = $this->request->getParam('id');

$rating = $this->request->getParam('rating');

$ideaRepository = new RedisIdeaRepository();

$idea = $ideaRepository->find($ideaId); if (!$idea) {

throw new Exception('Idea does not exist');

}

$idea->addRating($rating);

$ideaRepository->update($idea);

$this->redirect('/idea/' . $ideaId);

}

}

interface IdeaRepository

{

// ...

}

class RedisIdeaRepository implements IdeaRepository

{

private $client;

public function construct()

{

$this->client = new Predis\Client();

}

public function find($id)

{

$idea = $this->client->get($this->getKey($id)); if (!$idea) {

return null;

}

return unserialize($idea);

}

public function update(Idea $idea)

{

$this->client->set(

$this->getKey($idea->getId()), serialize($idea)

);

}

private function getKey($id)

{

return 'idea:' . $id;

}

}

Easy again. You've created a RedisIdeaRepository that implements IdeaRepository interface and we have decided to use Predis as a connection manager. Code looks smaller, easier and faster. But what about the controller? It remains the same, we have just changed which repository to use, but it was just one line of code.

As an exercise for the reader, try to create the IdeaRepository for SQLite, a file or an in- memory implementation using arrays. Extra points if you think about how ORM Repositories fit with Domain Repositories and how ORM *@annotations* affect this architecture.

## 将业务和Web框架分开

我们已经看到，从一个持久性策略转换到另一个持久性策略是多么容易。 但是，坚持不是我们Hexagon的唯一优势。 那么用户如何与应用程序交互呢?

您的CTO已经在您的团队转向Symfony2的路线图中设置好了，因此当您在当前的ZF1应用程序中开发新功能时，我们希望让迁移过程更加轻松。 这很棘手，让我看看你的清单5:

class IdeaController extends Zend\_Controller\_Action

{

public function rateAction()

{

$ideaId = $this->request->getParam('id');

$rating = $this->request->getParam('rating');

$ideaRepository = new RedisIdeaRepository();

$useCase = new RateIdeaUseCase($ideaRepository);

$response = $useCase->execute($ideaId, $rating);

$this->redirect('/idea/' . $ideaId);

}

}

interface IdeaRepository

{

// ...

}

class RateIdeaUseCase

{

private $ideaRepository;

public function construct(IdeaRepository $ideaRepository)

{

$this->ideaRepository = $ideaRepository;

}

public function execute($ideaId, $rating)

{

try {

$idea = $this->ideaRepository->find($ideaId);

} catch(Exception $e) {

throw new RepositoryNotAvailableException();

}

if (!$idea) {

throw new IdeaDoesNotExistException();

}

try {

$idea->addRating($rating);

$this->ideaRepository->update($idea);

} catch(Exception $e) {

throw new RepositoryNotAvailableException();

}

return $idea;

}

}

Let's review the changes. Our controller is not having any business rules at all. We have pushed all the logic inside a new object called RateIdeaUseCase that encapsulates it. This object is also known as Controller, Interactor or Application Service.

The magic is done by the execute method. All the dependencies such as the RedisIdeaRepository are passed as an argument to the constructor. All the references to an IdeaRepository inside our UseCase are pointing to the interface instead of any concrete implementation.

That's really cool. If you take a look inside RateIdeaUseCase, there is nothing talking about MySQL or Zend Framework. No references, no instances, no annotations, nothing. It is like your infrastructure does not mind. It just talks about business logic.

Additionally, we have also tuned the Exceptions we throw. Business processes also have exceptions. NotAvailableRepository and IdeaDoesNotExist are two of them. Based on the one being thrown we can react in different ways in the framework boundary.

Sometimes, the number of parameters that a UseCase receives can be too many. In order to organize them, it is quite common to build a *UseCase request* using a **Data Transfer Object** (**DTO**) to pass them together. Let's see how you could solve this in Listing 6:

class IdeaController extends Zend\_Controller\_Action

{

public function rateAction()

{

$ideaId = $this->request->getParam('id');

$rating = $this->request->getParam('rating');

$ideaRepository = new RedisIdeaRepository();

$useCase = new RateIdeaUseCase($ideaRepository);

$response = $useCase->execute(

new RateIdeaRequest($ideaId, $rating)

);

$this->redirect('/idea/' . $response->idea->getId());

}

}

class RateIdeaRequest

{

public $ideaId; public $rating;

public function construct($ideaId, $rating)

{

$this->ideaId = $ideaId;

$this->rating = $rating;

}

}

class RateIdeaResponse

{

public $idea;

public function construct(Idea $idea)

{

$this->idea = $idea;

}

}

class RateIdeaUseCase

{

// ...

public function execute($request)

{

$ideaId = $request->ideaId;

$rating = $request->rating;

// ...

return new RateIdeaResponse($idea);

}

}

The main changes here are introducing two new objects, a Request and a Response. They are not mandatory, maybe a UseCase has no request or response. Another important detail is how you build this request. In this case, we are building it getting the parameters from ZF request object.

Ok, but wait, what's the real benefit? it is easier to change from one framework to other, or execute our UseCase from another *delivery mechanism*. Let's see this point.

## Rating an Idea Using the API

During the day, your Product Owner comes to you and says: *by the way, a user should be able to rate an idea using our mobile app. I think we will need to update the API, could you do it for this sprint?*. Here's the PO again. *No problem!*. Business is impressed with your commitment.

As Robert C. Martin says: *The Web is a delivery mechanism [...] Your system architecture should be as ignorant as possible about how it is to be delivered. You should be able to deliver it as a console app, a web app, or even a web service app, without undue complication or any change to the fundamental architecture*.

Your current API is built using Silex, the PHP micro-framework based on the Symfony2 Components. Let's go for it in Listing 7:

require\_once DIR .'/../vendor/autoload.php';

$app = new Silex\Application();

// ... more routes

$app->get(

'/api/rate/idea/{ideaId}/rating/{rating}', function ($ideaId, $rating) use ($app) {

$ideaRepository = new RedisIdeaRepository();

$useCase = new RateIdeaUseCase($ideaRepository);

$response = $useCase->execute(

new RateIdeaRequest($ideaId, $rating)

);

return $app->json($response->idea);

}

);

$app->run();

Is there anything familiar to you? Can you identify some code that you have seen before? I'll give you a clue:

$ideaRepository = new RedisIdeaRepository();

$useCase = new RateIdeaUseCase($ideaRepository);

$response = $useCase->execute(

new RateIdeaRequest($ideaId, $rating)

);

*Man! I remember those 3 lines of code. They look exactly the same as the web application*. That's right, because the UseCase encapsulates the business rules you need to prepare the request, get the response and act accordingly.

We are providing our users with another way for rating an idea; another *delivery mechanism*. The main difference is where we created the RateIdeaRequest from. In the first example, it was from a ZF request and now it is from a Silex request using the parameters matched in the route.

## Console App Rating

Sometimes, a UseCase is going to be executed from a Cron job or the command line. As examples, batch processing or some testing command lines to accelerate the

development. While testing this feature using the web or the API, you realize that it would be nice to have a command line to do it, so you don't have to go through the browser.

If you are using shell scripts files, I suggest you to check the Symfony Console component. What would the code look like:

namespace Idy\Console\Command;

use Symfony\Component\Console\Command\Command;

use Symfony\Component\Console\Input\InputArgument; use Symfony\Component\Console\Input\InputInterface; use Symfony\Component\Console\Output\OutputInterface;

class VoteIdeaCommand extends Command

{

protected function configure()

{

$this

->setName('idea:rate')

->setDescription('Rate an idea')

->addArgument('id', InputArgument::REQUIRED)

->addArgument('rating', InputArgument::REQUIRED);

}

protected function execute( InputInterface $input, OutputInterface $output

) {

$ideaId = $input->getArgument('id');

$rating = $input->getArgument('rating');

$ideaRepository = new RedisIdeaRepository();

$useCase = new RateIdeaUseCase($ideaRepository);

$response = $useCase->execute(

new RateIdeaRequest($ideaId, $rating)

);

$output->writeln('Done!');

}

}

Again those 3 lines of code. As before, the UseCase and its business logic remain untouched, we are just providing a new *delivery mechanism.* Congratulations, you've discovered the *user side* hexagon edge.

There is still a lot to do. As you may have heard, a real craftsman does TDD. We have already started our story so we must be ok with just testing after.

## Testing Rating an Idea UseCase

Michael Feathers introduced a definition of legacy code as *code without tests*. You don't want your code to be legacy just born, do you?

In order to unit test this UseCase object, you decide to start with the easiest part, what happens if the repository is not available? How can we generate such behavior? Do we stop our Redis server while running the unit tests? No. We need to have an object that has such behavior. Let's use a *mock* object in Listing 9:

class RateIdeaUseCaseTest extends \PHPUnit\_Framework\_TestCase

{

/\*\*

* @test

\*/

public function whenRepositoryNotAvailableAnExceptionIsThrown()

{

$this->setExpectedException('NotAvailableRepositoryException');

$ideaRepository = new NotAvailableRepository();

$useCase = new RateIdeaUseCase($ideaRepository);

$useCase->execute(

new RateIdeaRequest(1, 5)

);

}

}

class NotAvailableRepository implements IdeaRepository

{

public function find($id)

{

throw new NotAvailableException();

}

public function update(Idea $idea)

{

throw new NotAvailableException();

}

}

Nice. NotAvailableRepository has the behavior that we need and we can use it with

RateIdeaUseCase because it implements IdeaRepository interface.

Next case to test is what happens if the idea is not in the repository. Listing 10 shows the code:

class RateIdeaUseCaseTest extends \PHPUnit\_Framework\_TestCase

{

// ...

/\*\*

* @test

\*/

public function whenIdeaDoesNotExistAnExceptionShouldBeThrown()

{

$this->setExpectedException('IdeaDoesNotExistException');

$ideaRepository = new EmptyIdeaRepository();

$useCase = new RateIdeaUseCase($ideaRepository);

$useCase->execute(

new RateIdeaRequest(1, 5)

);

}

}

class EmptyIdeaRepository implements IdeaRepository

{

public function find($id)

{

return null;

}

public function update(Idea $idea)

{

}

}

Here, we use the same strategy but with an EmptyIdeaRepository. It also implements the same interface but the implementation always returns null regardless which identifier the find method receives.

Why are we testing these cases?, remember Kent Beck's words: *Test everything that could possibly break*.

Let's carry on with the rest of the feature. We need to check a special case that is related with having a read available repository where we cannot write to. Solution can be found in Listing 11:

class RateIdeaUseCaseTest extends \PHPUnit\_Framework\_TestCase

{

// ...

/\*\*

* @test

\*/

public function whenRatingAnIdeaNewRatingShouldBeAdded()

{

$ideaRepository = new OneIdeaRepository();

$useCase = new RateIdeaUseCase($ideaRepository);

$response = $useCase->execute( new RateIdeaRequest(1, 5)

);

$this->assertSame(5, $response->idea->getRating());

$this->assertTrue($ideaRepository->updateCalled);

}

}

class OneIdeaRepository implements IdeaRepository

{

public $updateCalled = false;

public function find($id)

{

$idea = new Idea();

$idea->setId(1);

$idea->setTitle('Subscribe to php[architect]');

$idea->setDescription('Just buy it!');

$idea->setRating(5);

$idea->setVotes(10);

$idea->setAuthor('john@example.com');

return $idea;

}

public function update(Idea $idea)

{

$this->updateCalled = true;

}

}

Ok, now the key part of the feature is still remaining. We have different ways of testing this, we can write our own mock or use a mocking framework such as Mockery or Prophecy.

Let's choose the first one. Another interesting exercise would be to write this example and the previous ones using one of these frameworks:

class RateIdeaUseCaseTest extends \PHPUnit\_Framework\_TestCase

{

// ...

/\*\*

* @test

\*/

public function whenRatingAnIdeaNewRatingShouldBeAdded()

{

$ideaRepository = new OneIdeaRepository();

$useCase = new RateIdeaUseCase($ideaRepository);

$response = $useCase->execute( new RateIdeaRequest(1, 5)

);

$this->assertSame(5, $response->idea->getRating());

$this->assertTrue($ideaRepository->updateCalled);

}

}

class OneIdeaRepository implements IdeaRepository

{

public $updateCalled = false;

public function find($id)

{

$idea = new Idea();

$idea->setId(1);

$idea->setTitle('Subscribe to php[architect]');

$idea->setDescription('Just buy it!');

$idea->setRating(5);

$idea->setVotes(10);

$idea->setAuthor('john@example.com');

return $idea;

}

public function update(Idea $idea)

{

$this->updateCalled = true;

}

}

Bam! 100% Coverage for the UseCase. Maybe, next time we can do it using TDD so the test will come first. However, testing this feature was really easy because of the way decoupling is promoted in this architecture.

Maybe you are wondering about this:

$this->updateCalled = true;

We need a way to guarantee that the update method has been called during the UseCase execution. This does the trick. This *test double* object is called a *spy*, *mocks* cousin.

When to use mocks? As a general rule, use mocks when crossing boundaries. In this case, we need mocks because we are crossing from the domain to the persistence boundary.

What about testing the infrastructure?

## Testing Infrastructure

If you want to achieve 100% coverage for your whole application you will also have to test your infrastructure. Before doing that, you need to know that those unit tests will be more coupled to your implementation than the business ones. That means that the probability to be broken with implementation details changes is higher. So it is a trade-off you will have to consider.

So, if you want to continue, we need to do some modifications. We need to decouple even more. Let's see the code in Listing 13:

class IdeaController extends Zend\_Controller\_Action

{

public function rateAction()

{

$ideaId = $this->request->getParam('id');

$rating = $this->request->getParam('rating');

$useCase = new RateIdeaUseCase( new RedisIdeaRepository(

new Predis\Client()

)

);

$response = $useCase->execute(

new RateIdeaRequest($ideaId, $rating)

);

$this->redirect('/idea/' . $response->idea->getId());

}

}

class RedisIdeaRepository implements IdeaRepository

{

private $client;

public function construct($client)

{

$this->client = $client;

}

// ...

public function find($id)

{

$idea = $this->client->get($this->getKey($id)); if (!$idea) {

return null;

}

return $idea;

}

}

如果我们想100％单元测试RedisIdeaRepository，我们需要能够将Predis \ Client作为参数传递给存储库，而无需指定TypeHinting，因此我们可以传递一个模拟来强制覆盖所有情况所需的代码流.

这迫使我们更新Controller来构建Redis连接，将其传递给存储库并将结果传递给UseCase.

现在，这完全是关于创建模拟，测试案例和乐于做断言.

## Arggg，这么多的依赖！

手工创建这么多的依赖关系是否正常？ 不是。通常使用依赖注入组件或具有此功能的服务容器。 再次，Symfony来救援，但是，你也可以检查PHP-DI 4.

在将Symfony Service Container组件应用于我们的应用程序之后，我们来看清单14中的结果代码:

class IdeaController extends ContainerAwareController

{

public function rateAction()

{

$ideaId = $this->request->getParam('id');

$rating = $this->request->getParam('rating');

$useCase = $this->get('rate\_idea\_use\_case');

$response = $useCase->execute(

new RateIdeaRequest($ideaId, $rating)

);

$this->redirect('/idea/' . $response->idea->getId());

}

}

控制器已被修改为可以访问容器，这就是为什么它从一个新的基本控制器ContainerAwareController继承而来，它具有get方法来检索包含的每个服务:

<xml version="1.0" >

<container [xmlns="http://symfony.com/schema/dic/services"](http://symfony.com/schema/dic/services) [xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"](http://www.w3.org/2001/XMLSchema-instance) xsi:schemaLocation="

<http://symfony.com/schema/dic/services> [http://symfony.com/schema/dic/services/services-1.0.xsd">](http://symfony.com/schema/dic/services/services-1.0.xsd)

<services>

<service

id="rate\_idea\_use\_case" class="RateIdeaUseCase">

<argument type="service" id="idea\_repository" />

</service>

<service

id="idea\_repository" class="RedisIdeaRepository">

<argument type="service">

<service class="Predis\Client" />

</argument>

</service>

</services>

</container>

在清单15中，您还可以找到用于配置服务容器的XML文件。 这非常容易理解，但如果您需要更多信息，请参阅中的Symfony Service Container Component站点。

## 域服务和通知六边形

我们是否忘记了一些东西？ 应该通过电子邮件通知作者，是的！ 确实如此。 我们在清单16中看看如何更新UseCase来完成这项工作:

class RateIdeaUseCase

{

private $ideaRepository; private $authorNotifier;

public function construct( IdeaRepository $ideaRepository, AuthorNotifier $authorNotifier

) {

$this->ideaRepository = $ideaRepository;

$this->authorNotifier = $authorNotifier;

}

public function execute(RateIdeaRequest $request)

{

$ideaId = $request->ideaId;

$rating = $request->rating;

try {

$idea = $this->ideaRepository->find($ideaId);

} catch(Exception $e) {

throw new RepositoryNotAvailableException();

}

if (!$idea) {

throw new IdeaDoesNotExistException();

}

try {

$idea->addRating($rating);

$this->ideaRepository->update($idea);

} catch(Exception $e) {

throw new RepositoryNotAvailableException();

}

try {

$this->authorNotifier->notify(

$idea->getAuthor()

);

} catch(Exception $e) {

throw new NotificationNotSentException();

}

return $idea;

}

}

如您所知，我们已经添加了一个新参数来传递AuthorNotifier服务，该服务会将电子邮件发送给作者。 这是端口和适配器命名中的端口。 我们还更新了执行方法中的业务规则.

存储库不是唯一可以访问基础架构的对象，应该使用接口或抽象类来解耦。 域服务也可以。 当您的域中只有一个实体不具有明确拥有的行为时，您应该创建一个域服务。 一个典型的模式是编写一个抽象的域服务，它具有一些具体的实现和适配器将实现的一些其他抽象方法.

As an exercise, define the implementation details for the AuthorNotifier abstract service. Options are SwiftMailer or just plain mail calls. It is up to you.

## 让我们回顾一下

为了创建一个干净的架构来帮助您创建易于编写和测试应用程序，我们可以使用六边形架构。 为了实现这一点，我们将用户故事业务规则封装在UseCase或Interactor对象中。 我们从框架请求中构建UseCase请求，实例化UseCase及其所有依赖关系，然后执行它。 我们得到回应，并据此采取相应行动。 如果我们的框架有一个依赖注入组件，你可以用它来简化代码.

为了使用户能够访问不同客户端的功能（Web，API，控制台等），可以使用不同的传递机制使用相同的UseCase对象。

要进行测试，请使用与所定义的所有接口行为相似的模拟函数，以便涵盖特殊情况或错误流程。 享受完成的好工作.

## 六角形建筑

在几乎所有的博客和书籍中，您都可以找到代表不同领域软件的同心圆的图纸。 正如罗伯特C.马丁在他的清洁架构文章中解释的那样，外层圈子就是您的基础设施所在的地方。 内圈是你的实体居住的地方。 使这个架构有效的最重要的规则是依赖规则。 这条规则说源代码依赖关系只能指向内部。 内圈没有任何东西可以在外圈知道任何事情.

## 关键点

如果100％的单元测试代码覆盖率对您的应用程序很重要，请使用此方法。 此外，如果您希望能够切换您的存储策略，Web框架或任何其他类型的第三方代码。 该体系结构对于需要跟上不断变化的需求的持久应用程序尤其有用.

## 下一步是什么?

如果您有兴趣进一步了解六角形建筑和其他近似概念，您应该阅读文章开头提供的相关URL，查看CQRS和事件采购。 此外，不要忘记订阅谷歌组和RSS关于DDD的信息，例如http：//dddinphp.org，关注@VaughnVernon和@ ericevans0等Twitter用户.

# 参考书目

Beck, Kent. [Test-Driven Development: By Example](http://www.amazon.com/Test-Driven-Development-Kent-Beck/dp/0321146530). Addison-Wesley Professional, 2002. Brandolini, Alberto. [Introducing EventStorming](https://leanpub.com/introducing_eventstorming). Leanpub, 2016.

Evans, Eric. [Domain-Driven Design Reference: Definitions and Pattern Summaries](http://www.amazon.com/Domain-Driven-Design-Reference-Definitions-Summaries/dp/1457501198). Dog Ear Publishing, 2014.

Evans, Eric. [Domain-Driven Design: Tackling Complexity in the Heart of Software](http://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215). Addison-Wesley Professional, 2003.

Fowler, Martin. [Patterns of Enterprise Application Architecture](http://www.amazon.com/Patterns-Enterprise-Application-Architecture-Martin-ebook/dp/B000OZ0NAI). Addison-Wesley Professional, 2002.

Hohpe, Gregor, and Bobby Woolf. [Enterprise Integration Patterns: Designing,](http://www.amazon.com/Enterprise-Integration-Patterns-Designing-Addison-Wesley-ebook/dp/B007MQLL4E) [Building, and Deploying Messaging Solutions](http://www.amazon.com/Enterprise-Integration-Patterns-Designing-Addison-Wesley-ebook/dp/B007MQLL4E). Addison-Wesley Professional, 2012.

Martin, Robert C. [Agile Software Development, Principles, Patterns, and Practices](http://www.amazon.com/Software-Development-Principles-Patterns-Practices/dp/0135974445). Pearson, 2002.

Martin, Robert C. [Clean Code: A Handbook of Agile Software Craftsmanship](http://www.amazon.com/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882). Prentice Hall, 2008.

Meszaros, Gerard. [xUnit Test Patterns: Refactoring Test Code](https://www.amazon.com/xUnit-Test-Patterns-Refactoring-Code/dp/0131495054). Addison-Wesley Professional, 2007.

Newman, Sam. [Building Microservices](http://www.amazon.com/Building-Microservices-Sam-Newman/dp/1491950358). O’Reilly Media, 2015.

*Bibliography*

Nilsson, Jimmy. [Applying Domain-Driven Design and Patterns: With Examples in C#](http://www.amazon.com/Applying-Domain-Driven-Design-Patterns-Examples/dp/0321268202) [and .NET](http://www.amazon.com/Applying-Domain-Driven-Design-Patterns-Examples/dp/0321268202). Addison-Wesley Professional, 2006.

Sadalage, Pramod J., and Martin Fowler. [NoSQL Distilled: A Brief Guide to the](http://www.amazon.com/NoSQL-Distilled-Emerging-Polyglot-Persistence/dp/0321826620) [Emerging World of Polyglot Persistence](http://www.amazon.com/NoSQL-Distilled-Emerging-Polyglot-Persistence/dp/0321826620). Addison-Wesley Professional, 2012.

Vernon, Vaughn. [Domain-Driven Design Distilled](http://www.amazon.com/Domain-Driven-Design-Distilled-Vaughn-Vernon/dp/0134434420). Addison-Wesley Professional, 2016.

Vernon, Vaughn. [Implementing Domain-Driven Design](http://www.amazon.com/Implementing-Domain-Driven-Design-Vaughn-Vernon-ebook/dp/B00BCLEBN8). Addison-Wesley Professional, 2013.

###### [ 361 ]

The End

恭喜，你已经完成了这本书！ 我们想个人感谢你，没有你的支持和反馈，这本书将永远不可能。 这是一段难以置信的旅程，我们非常幸运地相信你。 我们真的希望你像我们一样享受这次旅程.

我们一直痴迷于为读者提供尽可能好的体验。 我们根据您令人难以置信的反馈重复了本书的内容。 如果有什么我们可以改进的，请通过在我们的Github项目中打开一个问题来帮助我们.

If you liked it and was useful to you, it might be useful for others too! Feel free to share your experience in Twitter or [give us a review in Goodreads](https://www.goodreads.com/book/show/26032410-domain-driven-design-in-php).

Thanks again!

– Carlos, Christian and Keyvan

# Index

#### A

aggregate [213](#_bookmark274)

Anemic Domain Model avoiding [142](#_bookmark175)

code reuse, false sense [141](#_bookmark171)

encapsulating [141](#_bookmark171)

versus Rich Domain Model [136](#_bookmark168), [137](#_bookmark170), [141](#_bookmark171)

Anti-Corruption layer [328](#_bookmark425), [330](#_bookmark426)

API

using [347](#_bookmark447), [348](#_bookmark449)

application services about [125](#_bookmark157), [126](#_bookmark159), [128](#_bookmark160)

AJAX Content Integration [313](#_bookmark391)

anatomy [301](#_bookmark370)

customizing [305](#_bookmark377)

dependency inversion [302](#_bookmark372) ESI Content Integration [314](#_bookmark395) execution [306](#_bookmark380)

instantiating [303](#_bookmark375), [304](#_bookmark376), [305](#_bookmark377) multiple application services [314](#_bookmark395) on compound layouts [313](#_bookmark391) request [297](#_bookmark360)

request objects, design [300](#_bookmark365) request, building [298](#_bookmark361), [299](#_bookmark363) Symfony Sub Requests [314](#_bookmark395) testing [315](#_bookmark399), [316](#_bookmark400)

values, returning [307](#_bookmark383)

Atomicity, Consistency, Isolation, Durability(ACID)

[86](#_bookmark95), [200](#_bookmark249)

attribute validation [116](#_bookmark144)

#### B

Bounded applications [184](#_bookmark229), [185](#_bookmark231)

Bounded Context Integrations implementing [325](#_bookmark420)

message queues [330](#_bookmark426), [331](#_bookmark428), [332](#_bookmark429), [335](#_bookmark430)

modern RPC [325](#_bookmark420), [326](#_bookmark424), [328](#_bookmark425)

Bounded Contexts [184](#_bookmark229), [185](#_bookmark231), [324](#_bookmark416)

business

decoupling [341](#_bookmark440), [344](#_bookmark444), [346](#_bookmark446)

#### C

collection backed, by Join Table by database entity [85](#_bookmark91)

with Ad Hoc ORM [85](#_bookmark91)

with Doctrine [82](#_bookmark90)

collection-oriented Repositories about [265](#_bookmark319), [266](#_bookmark321), [269](#_bookmark322), [270](#_bookmark323)

Doctrine ORM [272](#_bookmark326)

in-memory implementation [271](#_bookmark325)

command handlers about [320](#_bookmark407)

Tactician library [320](#_bookmark407), [321](#_bookmark409)

concurrency, key concepts

Optimistic Concurrency Control (OCC) [207](#_bookmark266)

Pessimistic Concurrency Control (PCC) [205](#_bookmark260)

Conformist integration [325](#_bookmark420), [328](#_bookmark425) Console App Rating [349](#_bookmark451) Customer-Supplier [330](#_bookmark426)

#### D

Data Access Objects (DAOs) [264](#_bookmark317), [265](#_bookmark319)

data store

integrating [322](#_bookmark413), [323](#_bookmark415), [324](#_bookmark416) Data Transfer Object (DTO) [346](#_bookmark446) DateTimes, entites

dates, passing as parameters [110](#_bookmark136)

external fake [113](#_bookmark140)

reflection [115](#_bookmark143)

test class [111](#_bookmark139)

dependencies creating [355](#_bookmark459)

dependency inversion [302](#_bookmark372)

Dependency Rule [359](#_bookmark465)

Doctrine Events [152](#_bookmark195)

Doctrine ORM about [272](#_bookmark326)

object mapping [273](#_bookmark329)

Doctrine Query Language (DQL) [272](#_bookmark326)

Doctrine

entities mapping, XML used [102](#_bookmark126), [103](#_bookmark128)

entities, mapping [100](#_bookmark123) entity identity, mapping [103](#_bookmark128) file, mapping [105](#_bookmark131)

setting up [99](#_bookmark120)

Domain Events characteristics about [148](#_bookmark187)

immutability [149](#_bookmark191)

naming conventions [148](#_bookmark187)

Ubiquitous language [149](#_bookmark191)

Domain Events about [122](#_bookmark152), [319](#_bookmark403)

defining [144](#_bookmark180), [145](#_bookmark182)

DomainEventListeners, setting up [163](#_bookmark208)

Event Store [155](#_bookmark199)

example [146](#_bookmark185), [148](#_bookmark187)

metaphor [146](#_bookmark185)

modeling [149](#_bookmark191), [151](#_bookmark194)

persisting [154](#_bookmark197)

publisher, working [161](#_bookmark205)

publishing, from application services [161](#_bookmark205) publishing, from Domain model [158](#_bookmark201) publishing, from Domain services [161](#_bookmark205) publishing, from entity [158](#_bookmark201)

testing [165](#_bookmark210)

Domain Services [358](#_bookmark462)

Domain services

about [125](#_bookmark157), [128](#_bookmark160), [130](#_bookmark163) code reuse, issues [133](#_bookmark165) testing [135](#_bookmark167)

Domain-Driven Design (DDD) about [7](#_bookmark4), [8](#_bookmark6), [336](#_bookmark434)

considering [10](#_bookmark11)

reference link [7](#_bookmark4)

strategic design [8](#_bookmark6)

strategical, overview [11](#_bookmark14)

tactical design [8](#_bookmark6)

three pillars [8](#_bookmark6)

tricky parts [11](#_bookmark14)

ubiquitous language [8](#_bookmark6)

Ubiquitous Language [9](#_bookmark9)

#### E

entire object validation [118](#_bookmark147)

entities

about [88](#_bookmark102), [122](#_bookmark152)

DateTimes [109](#_bookmark134)

Doctrine, setting up [99](#_bookmark120)

mapping, Annotated Code used [100](#_bookmark123), [102](#_bookmark126)

persisting [99](#_bookmark120)

testing [106](#_bookmark133)

entity

versus value objects [51](#_bookmark43)

execution, application services class method [306](#_bookmark380)

multiple class method [306](#_bookmark380)

#### F

factories

Object Mother [258](#_bookmark309) Test Data Builder [259](#_bookmark312) testing [257](#_bookmark308)

Factory Method

aggregates, building [254](#_bookmark304), [255](#_bookmark306)

invariants, forcing [248](#_bookmark297)

on Aggregate Root [247](#_bookmark295), [248](#_bookmark297)

on service [249](#_bookmark299)

specifications, building [249](#_bookmark299), [251](#_bookmark302), [253](#_bookmark303)

fanout [335](#_bookmark430)

first-level namespacing [180](#_bookmark223)

#### H

Hexagonal Architecture [336](#_bookmark434)

about [359](#_bookmark465)

approach [337](#_bookmark437), [339](#_bookmark438)

exploring [359](#_bookmark465)

reviewing [358](#_bookmark462)

hydration [159](#_bookmark204)

#### I

identity operation about [92](#_bookmark107)

application, generating [96](#_bookmark115), [97](#_bookmark117), [98](#_bookmark118)

Bounded Context [99](#_bookmark120) identity, provided by client [95](#_bookmark113) persistence mechanism [92](#_bookmark107)

Immutable [148](#_bookmark187)

in-memory implementation [271](#_bookmark325)

infrastructure services [125](#_bookmark157), [130](#_bookmark163)

infrastructure testing [354](#_bookmark457)

integrated development environment (IDEs) [79](#_bookmark83)

Integration Relationships about [324](#_bookmark416)

Conformist [325](#_bookmark420)

Customer-Supplier [324](#_bookmark416)

Separate Ways [324](#_bookmark416)

International Standard Book Number (ISBN) [96](#_bookmark115)

#### J

JavaScript Object Notation (JSON) [86](#_bookmark95)

#### K

key concepts about [200](#_bookmark249)

ACID [201](#_bookmark252)

concurrency [205](#_bookmark260)

isolation levels [204](#_bookmark257)

locking [205](#_bookmark260)

referential integrity [204](#_bookmark257)

transactions [201](#_bookmark252), [202](#_bookmark254), [203](#_bookmark255)

key points [359](#_bookmark465)

#### L

leverage modules

first-level namespacing [180](#_bookmark223)

in PHP [180](#_bookmark223)

PEAR-style namespacing [180](#_bookmark223), [181](#_bookmark227)

#### M

message queues [331](#_bookmark428), [332](#_bookmark429), [335](#_bookmark430)

messaging middleware [331](#_bookmark428)

Microservices [12](#_bookmark16), [13](#_bookmark18)

modern RPC [325](#_bookmark420), [326](#_bookmark424), [328](#_bookmark425)

modules

application layer [198](#_bookmark244)

code, structuring in [185](#_bookmark231), [186](#_bookmark233)

design guidelines [187](#_bookmark234), [188](#_bookmark236)

infrastructure layer [190](#_bookmark237), [191](#_bookmark239), [195](#_bookmark240)

overview [180](#_bookmark223)

technologies, mixing [196](#_bookmark241)

monolithic system [322](#_bookmark413)

#### N

NoSQL, value objects characteristics MySQL JSON type [86](#_bookmark95) PostgreSQL JSONB type [86](#_bookmark95)

Notification Hexagon Edge [358](#_bookmark462)

#### O

object compositions validating [122](#_bookmark152)

object mapping, Doctrine ORM

Doctrine Custom Mapping Types [273](#_bookmark329), [275](#_bookmark331)

DQL implementation [276](#_bookmark333)

entity manager [276](#_bookmark333)

Online Travel Agency (OTA) [11](#_bookmark14) Open Closed Principle (OCP) [144](#_bookmark180) Open Host Service [325](#_bookmark420), [330](#_bookmark426) Optimistic Concurrency Control (OCC)

about [207](#_bookmark266)

reference link [207](#_bookmark266)

#### P

PEAR-style namespacing about [180](#_bookmark223), [181](#_bookmark227)

PSR-0 namespacing [181](#_bookmark227)

PSR-4 namespacing [181](#_bookmark227)

persistence edge [339](#_bookmark438), [341](#_bookmark440)

persistence mechanism, identity operation Active Record, versus Data Mapper for Rich

Domain Models [94](#_bookmark111)

Surrogate Identity [93](#_bookmark109)

persistence-oriented Repository about [278](#_bookmark336)

extra behavior [284](#_bookmark341)

Redis implementation [278](#_bookmark336)

SQL implementation [280](#_bookmark338)

persistence decoupling [341](#_bookmark440)

migrating, to Redis [343](#_bookmark443)

Pessimistic Concurrency Control (PCC), concurrency

with Doctrine [206](#_bookmark264), [210](#_bookmark270)

with Elasticsearch [207](#_bookmark266), [208](#_bookmark268), [209](#_bookmark269), [210](#_bookmark270), [211](#_bookmark272), [212](#_bookmark273)

PHP Standard Recommendation (PSR) [181](#_bookmark227)

PHP

leverage modules in [180](#_bookmark223) Ports and Adapters [336](#_bookmark434) primitive type

versus value objects [90](#_bookmark104)

Published Language [331](#_bookmark428)

#### R

remote Bounded Contexts about [166](#_bookmark212)

Domain services, syncing with REST [176](#_bookmark218)

messaging [166](#_bookmark212), [171](#_bookmark215), [172](#_bookmark216), [173](#_bookmark217)

replaceability, value objects characteristics side-effect-free behaviour [59](#_bookmark58), [60](#_bookmark60)

repositories [339](#_bookmark438), [341](#_bookmark440)

Repositories defining [264](#_bookmark317)

querying [284](#_bookmark341)

specification pattern [285](#_bookmark344)

testing [291](#_bookmark352)

request objects

business logic, avoiding [300](#_bookmark365)

designing [300](#_bookmark365)

primitives, using [300](#_bookmark365)

serializable [300](#_bookmark365)

tests, avoiding [300](#_bookmark365), [301](#_bookmark370)

Rich Domain Model

versus Anemic Domain Model [136](#_bookmark168), [137](#_bookmark170), [141](#_bookmark171)

#### S

security [319](#_bookmark403)

Security [87](#_bookmark98)

Self-Contained Systems (SCS) [12](#_bookmark16), [13](#_bookmark18)

separate ways [324](#_bookmark416)

Separation of Concerns (Soc) [15](#_bookmark22)

Serialize LOB, value objects characteristics Doctrine Custom Type [76](#_bookmark81)

Doctrine Object Mapping Type [74](#_bookmark78)

value objects, collection persisting [79](#_bookmark83)

value objects, collection serializing into single

column [80](#_bookmark85)

Serialized LOB, value objects characteristics improved serialization, with JMS Serializer [73](#_bookmark76)

services

testing, with in-memory implementations [295](#_bookmark355)

Single Responsibility Principle (SRP) about [339](#_bookmark438)

reference link [339](#_bookmark438)

specialized, shared, and common communication language [331](#_bookmark428)

specification pattern, Repositories

in-memory implementation [285](#_bookmark344), [286](#_bookmark346)

SQL implementation [287](#_bookmark348)

strategical design [322](#_bookmark413)

#### T

Tactician library [320](#_bookmark407), [321](#_bookmark409)

transactions about [317](#_bookmark401)

managing [288](#_bookmark349), [289](#_bookmark351)

#### U

Ubiquitous Language about [9](#_bookmark9)

event storming [9](#_bookmark9), [10](#_bookmark11)

UseCase

testing [350](#_bookmark452), [351](#_bookmark454), [352](#_bookmark455), [353](#_bookmark456), [354](#_bookmark457)

#### V

validation messages decoupling [120](#_bookmark149)

validation

about [116](#_bookmark144)

attribute validation [116](#_bookmark144)

entire object validation [118](#_bookmark147)

object compositions, validating [122](#_bookmark152)

validation messages, decoupling [120](#_bookmark149)

value objects characteristics about [54](#_bookmark47)

Ad Hoc ORM [72](#_bookmark75)

basic types [61](#_bookmark61)

collection backed, by Join Table [81](#_bookmark88)

conceptual whole [56](#_bookmark52)

embedded value, with Ad Hoc ORM [64](#_bookmark66)

embedded value, with Doctrine 2.4 [69](#_bookmark71), [71](#_bookmark73)

embedded value, with Doctrine 2.5 [67](#_bookmark69)

immutability [54](#_bookmark47), [55](#_bookmark51)

money object, describes [54](#_bookmark47) money object, measures [54](#_bookmark47) money object, quantifies [54](#_bookmark47) NoSQL [86](#_bookmark95)

persisting [63](#_bookmark65), [64](#_bookmark66)

replaceability [59](#_bookmark58)

Serialized LOB [72](#_bookmark75)

Serialized LOB, with Doctrine [74](#_bookmark78) single value objects, persisting [64](#_bookmark66) testing [61](#_bookmark61)

value equality [57](#_bookmark54), [58](#_bookmark56)

value objects defining [50](#_bookmark41), [51](#_bookmark43)

example [52](#_bookmark45)

versus entity [51](#_bookmark43)

versus primitive type [90](#_bookmark104)

values, application services

data transformers [310](#_bookmark387), [311](#_bookmark389), [312](#_bookmark390), [313](#_bookmark391)

DTO, from aggregate instances [308](#_bookmark385), [310](#_bookmark387)

#### W

web framework decoupling [344](#_bookmark444), [346](#_bookmark446)