**Data Access with Spring**

## Persistence and Storage

The field of Data Persistence has progressed enormously over the past 20 years. Not so long ago, if you were considering data storage and retrieval you were fairly limited to thinking in terms of a relational data store, or perhaps simple file storage.

Today there are much richer options, and Data Persistence now covers a myriad of different data models and data storage implementations.

Since you're reading this tutorial, it's likely that you are considering implementing a data storage using a component referred to as a [Repository], a pattern from [Domain Driven Design](http://en.wikipedia.org/wiki/Domain-driven_design), because:

* You have a data store where you need to access and potentially modify the data that's stored.
* You have an existing [Spring Data](http://projects.spring.io/spring-data/) application that you need to maintain.
* You want to investigate different data stores in the context of Spring Data.

This tutorial covers three data stores: the [MongoDB](http://www.mongodb.org) document store, a relational database using [JPA](http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html) and [Pivotal GemFire](http://gopivotal.com/pivotal-products/pivotal-data-fabric/pivotal-gemfire).

## What you'll build

The context for this tutorial is the Yummy Noodle Bar. As part of its planned expansion it needs to be able to store and update its Menu and store Orders.

You will extend the Yummmy Noodle Bar's application to:

* Store Menu data in MongoDB.
* Store Order data in a Relational Database.
* Track the Status of an Order using GemFire.

[Spring Data](http://projects.spring.io/spring-data/) will provide the bedrock of your persistence project, and you will discover how it makes your data access simpler, more consistent and more robust.

## What you'll need

To work through this tutorial you'll need a few things:

* About a half hour for each section. You might finish it in a week during your lunch break (or during the weekend!)
* An installation of [MongoDB](http://www.mongodb.org/)

### Downloading and running the code

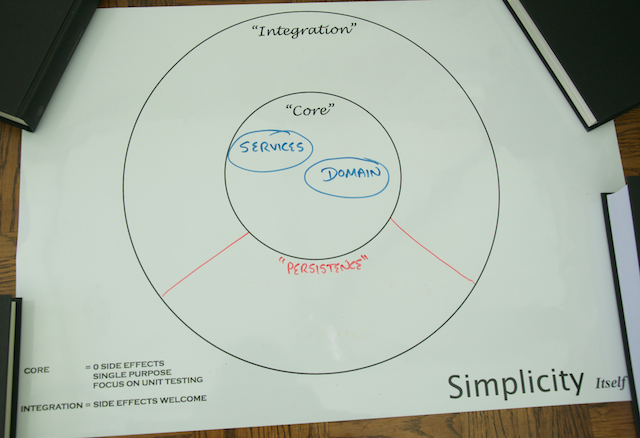
If you download the code base, you'll discover each section of this tutorial in a separate folder, numbered 1, 2, 3, etc. There are also several code drops: an **initial** one at the root, and a separate **complete** code drop for each section.

The **initial** code set contains:

* a project layout
* the core domain & event classes this tutorial starts off creating (the 'Yummy Noodle Bar Application', above)
* some basic unit tests for some of those classes

## The Home of Persistence in your Application Architecture

Repositories are an integration between your core application and external, persistent storage engines. Repositories can be seen as living in their own Persistence integration domain on the periphery of your applications core as shown in the following diagram



As an integration between your application core and the outside world there are a number of concerns that need to be addressed in the design and implementation of the components that make up your Repositories:

* The primary purpose of a Repository component is to integrate your application with the data stores in a natural and optimised (for data storage and retrieval) way.
* Components of your persistence domain will need to evolve at a rate that is appropriate for the data they are managing.
* Your Repository components should not contain any core logic. They should instead focus on collaborating with other components in the core domain of your application in order to support their core functionality.

That's enough on the design constraints placed on the components that implement your Repositories,

# Step 1: Understanding the Core & Persistence Domains and the Relationships between them

Currently the core application internal domain of the Yummy Noodle Bar is made up of the following sub-domains:

* Currently the core application internal domain of the Yummy Noodle Bar is made up of the following sub-domains:
* **Services**. Contains the components that exercise the core domain classes to provide the unique functionality for the application.

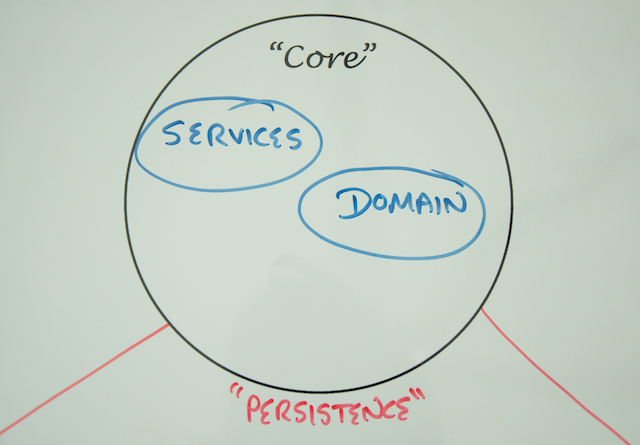
Focusing primarily on Orders, these can be acted upon by a number of events:

* **OrderCreatedEvent** Creates a new order for a number of menu-items.
* **OrderUpdatedEvent**. Updates an existing Order with some additional information, possibly payment information.
* **OrderDeletedEvent**. Deletes an existing order if it is not being cooked.
* **RequestAllCurrentOrdersEvent**. Requests the full list of all current orders be returned.

## Modelling the Persistence Domain

First up, the ability to create, update and remove Orders, MenuItems and OrderStatuses is your first task.

It can be tempting to simply map the core Order domain to the data stores and work from there, but that would ignore the boundary between the Core and the Persistence domain.



The data model of your persisted data will need to change at a rate that is manageable and technically feasible given the data store implementation. The core will need to evolve at the rate the Yummy Noodle bar system needs to internally evolve at. There is potentially friction between the two domains as they evolve at different rates.

To manage this friction you need to create concepts and components in the Persistence domain that are unique to, and can evolve at the rate needed by, the Persistence domain itself. This may result in similar types of components, but since their purpose will be very different the similarities are superficial.

## Understanding the Different Data Models and their Implementations

Modern data stores adopt one or more Data Models, typically only one per technology.

Your interactions with a data store, based on the features and limitations of its data model, will vary and so different capabilities and data access patterns are applied.

Some of the most common Data Models currently in use are:

* **Relational**

The relational model of data storage is characterised by the concepts of Tables, with structured Columns made up of Rows. This is is very analogous to a spreadsheet, where a table is like a worksheet, a row represents one record, and a column contains one specific piece of information. They are accessed by a dialect of the Structured Query Language (SQL).

What makes this type of data store relational is the way that Columns may define references to Columns in other tables, known as relations. These relations are usually strongly enforced by the database.

Relational Databases are highly structured, have an explicit Schema and almost invariably first class support for transactions.

* **Document**

A data store following the Document Data Model will have a far looser schema than a relational database, if it has one at all. Documents are structured, rich data structures that can contain nested documents, lists, maps and other constructs internally, all within a single document. Some elements of the documents can be optional, giving huge flexibility to the system designer.

Queries can be written against any value in the document structure, leading to very rich querying capabilities and document structures. A distinct difference with relational databases is that the logical collections of documents do not have to conform to a common schema. Documents aren't as heavily defined by relationships between columns, but instead more focused on the content of the documents.

* **Data Grid**

Data Grid is less well defined than the other models, but mostly commonly means a Key/Value store that has advanced replication and server side data processing built in. This is sometimes viewed as a caching data solution.

GemFire is such a store, and allows accessing data through either a Map interface, a rich query language, or writing code that executes on the GemFire cluster in a distributed fashion.

* **Key/Value**

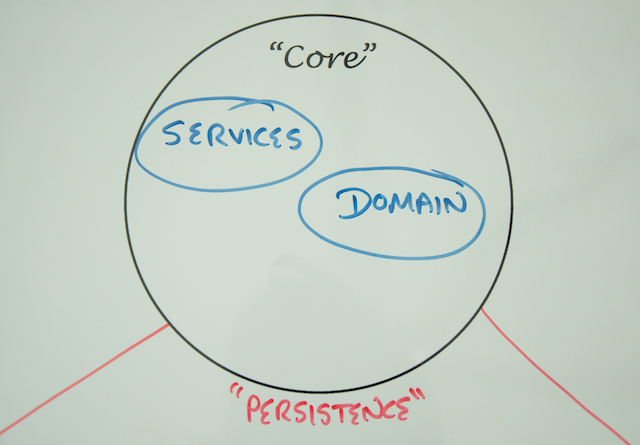
Key/Value is one of the simplest forms of Data Model. It is analogous to a Java Map, known in other languages as either a hash table, hash map, or an associative array. The data entries comprise a key with an associated value. They tend to be very fast and easily distributable. This is sometimes used as a caching data solution. Because many critical, high performance systems depend on such simple constructs, Key/Value data stores have risen quickly in popularity.

Some implementations offer a querying capability against the value data.

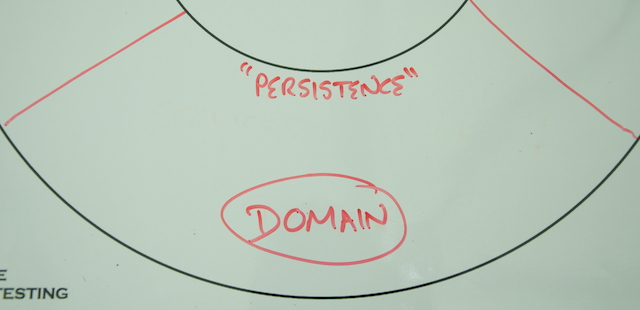
# Step 2: Storing Menu Data Using MongoDB

As you dive into writing code in this tutorial, the assumption is that The Yummy Noodle Bar application core has been implemented. You are now tackling the task of extending it to store data.

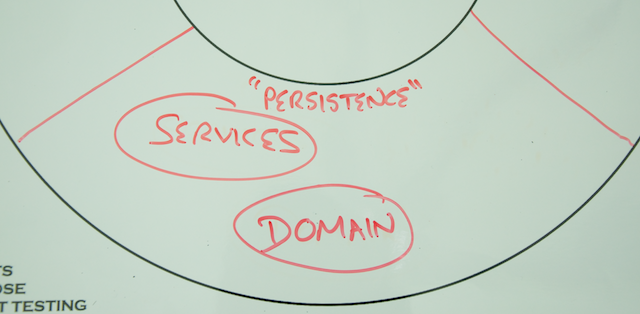
For this section, you will work within the Persistence domain to add this functionality:



In that domain you have a representation of MenuItem optimised for persistence that is contained in the Domain sub-domain:



In order for the Core Domain to send events to the Persistence Domain, you also have a Services sub-domain provided in the Persistence domain:



Inside the Persistence Domain's Service sub-domain, there is an event handler MenuPersistenceEventHandler, exchanging events with the application core, and the repository MenuItemRepository whose responsibility is to persist and retrieve MenuItem data for the rest of the application.

In this step you will implement MenuItemRepository using Spring Data MongoDB and integrate this with the MenuPersistenceEventHandler.

## About MongoDB

[MongoDB](http://www.mongodb.org) is a document-oriented database that stores data natively in a document format called [BSON (Binary JSON)](http://en.wikipedia.org/wiki/BSON). This is similar in structure to [JSON](http://en.wikipedia.org/wiki/JSON), and is ultimately derived from it.

MongoDB does not enforce a schema or document structure beyond the concept of a Collection, which contains a set of documents.

MongoDB does not enforce a schema or document structure beyond the concept of a Collection, which contains a set of documents.

Large scale data transformation and analysis is a native part of MongoDB, and can be performed either declaratively or with JavaScript functions as part of a [MapReduce](http://en.wikipedia.org/wiki/MapReduce) implementation.

## Install MongoDB

## Import Spring Data MongoDB

## Start with a (failing) test: Introducing MongoTemplate

Before you can implement the Repository, you have to build something that will use it. In this case, you will start by writing a simple test case that attempts to store a MenuItem using MongoTemplate.

MongoTemplate is one of the core classes provided by [Spring Data Mongo](http://www.springsource.org/spring-data/mongodb). It follows the familiar [Template Method pattern](http://en.wikipedia.org/wiki/Template_method_pattern) that is used extensively in other parts of Spring, such as JmsTemplate, JdbcTemplate and RestTemplate.

With respect to MongoDB, MongoTemplate does the leg work of connecting to a MongoDB server and managing the necessary resources involved, while also exposing a simple API that provides a large amount of functionality.

Test Driven Development guides you to test the smallest piece of the system you can, and then build your tests outwards from that. This builds confidence in the system as a whole.

The smallest piece in this case is the Persistence domain class, MenuItem. It will contain mapping and configuration information describing how it should be persisted into the database.

There is an existing helper class

com.yummynoodlebar.persistence.domain.fixture.PersistenceFixture

and now that you have imported Spring Data Mongo, you can create another

com.yummynoodlebar.persistence.domain.fixture.MongoAssertions

They provide some methods we can use to make our tests a bit more readable.

## Implement a CRUD repository

MenuItem is now ready to be persisted. We could write an implementation of MenuItemRepository using MongoTemplate but Spring Data gives us a better option: It can create an implementation of the Repository interface for us at runtime.

You first need to update MenuItemRepository into something that Spring Data can handle.

Before you do that though, you need a (failing) test again!

Create a new component in the application'c Config Domain called com.yummynoodlebar.config.MongoConfiguration, leaving it empty for now.

CrudRepository is part of the Spring Data repository hierarchy. It acts as both a marker interface and it adds several key methods to provide us with a living, breathing implementation of a repository, with almost zero code.

## Extend your Repository with a Custom Finder

A late breaking requirement has been uncovered! Does that sound familiar...

Users are going to be given the opportunity to select dishes by the names of the ingredients that they contain.

Looking at MenuItem, the document that is stored in MongoDB will look similar to the following:

{

"\_id" : ObjectId("520d388bea7e3adc2a054886"),

"\_class" : "com.yummynoodlebar.persistence.domain.MenuItem",

"ingredients" : [

{

"name" : "Noodles",

"description" : "Crisp, lovely noodles"

},

{

"name" : "Peanuts",

"description" : "A Nut"

}

],

"cost" : "12.99",

"minutesToPrepare" : 0

}

A search based on ingredients will require querying deep inside the document structure to correctly identify the matching documents.

Before you do anything else, let's write a test that attempts to do this type of search:

## Extend the Repository with Map/Reduce

A more interesting requirement would be helping the user look up the ingredients used in the most dishes. MongoDB provides a system to perform this kind of analysis, MapReduce

To use MapReduce, you need to gain access to the MongoTemplate directly and add this into the Repository that Spring Data is currently managing for you.

This class references two JavaScript functions, the mapper and the reducer, respectively.

Create 2 new JavaScript files, in the src/main/resources directory:

## Summary

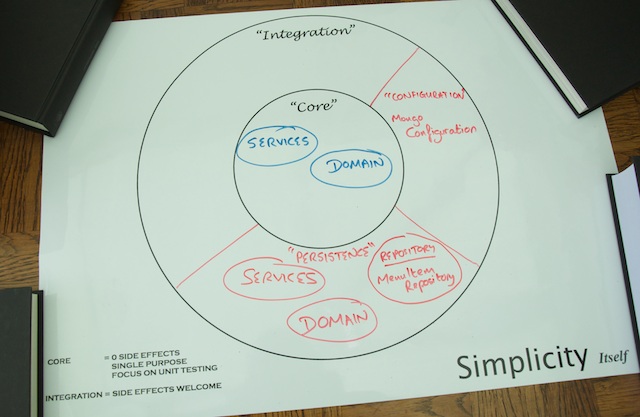
You have created a custom data analysis task against MongoDB. Interacting with MongoDB didn't require a whole lot of effort, thanks to Spring Data Mongo.

You've also added configuration to drive the MongoDB infrastructure by adding a new configuration component into your Configuration domain.

You now have your first repository that is contained in your new Repository sub-domain in your Persistence domain that houses the repository components responsible for storing and retrieving data. The new Repository sub-domain is shown in the Life Preserver below:



Your full Life Preserver diagram should look like the following now:



Now that the Menu data is safely stored in Mongo its time to turn your attention to storing and retrieving Orders using a whole different data model and data storage technology … JPA.

# Step 3: Storing Order Data Using the Java Persistence API (JPA)

Your next task for the Yummy Noodle Bar persistence project is to store Order data. Yummy Noodle Bar has decided to use [PostgreSQL](http://www.postgresql.org) to store this data, a freely available, robust, relational database.

Continuing from the [last section](http://spring.io/guides/tutorials/data/2/), your focus will be on developing components within the Persistence domain

In that domain you have a component that represents an Order, com.yummynoodle.persistence.domain.Order that you can optimise for persistence without affecting the rest of your application..

There is an event handler, OrderPersistenceEventHandler, that exchanges events between the application core and the repository OrdersRepository. The repository's responsibility is to persist and retrieve Order data for the rest of the application.

You will implement OrdersRepository using Spring Data JPA and integrate this with the OrderPersistenceEventHandler.

## A Word on JPA

[JPA](http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html) is the standard Java mechanism for working with relational databases. JPA provides an API to map Java Objects to relational concepts and comes with a rich vocabulary for querying across these objects, transparently translating to SQL (most of the time).

Since JPA is a standard there are many implementations referred to as JPA Providers.

For this tutorial, you will use [Spring Data JPA](http://projects.spring.io/spring-data-jpa/) along with [Hibernate](http://www.hibernate.org) as the JPA provider.

## Using H2 as an in memory database for Testing

PostgreSQL is a fully functional database that is suitable for production. However for testing, a lighter, embedded database is suitable. You will use [H2](http://www.h2database.com) for development purposes during this tutorial. H2 allows the easy creation and destruction of database instances in a lifecycle controlled by a test.

The JPA standard provides enough of an abstraction that you may use different databases in production and development.

We recommend to test application integration with the production database in addition to the interaction tests run against H2. This would normally be part of your minimal set of acceptance or 'smoke' tests.

## Importing Spring Data JPA

Import Spring Data JPA and the Hibernate JPA Provider into your project

## Start with a (failing) test: Introducing JPA

Following the pattern from the previous section, first create a test to drive your development that checks that the persistence mapping class correctly stores and retrieves records.

First, create a placeholder configuration class

It is important to have an understanding of how your object is being mapped against the database and test that it meets your expectations. If you know how and why the mapping is occurring you can create indexes and other optimisations and be safe in the knowledge that the data is where you expect it to be. You may also access the data outside of the JPA provider, directly querying the database.

The method DataSource dataSource() creates the embedded H2 database. This creates a new H2 instance within the same ApplicationContext and provides a DataSource interface to it, usable by JPA.

The method EntityManagerFactory entityManagerFactory() creates the EntityManagerFactory. This class is responsible for creating the EntityManager, and is JPA Provider specific. In this case, this shows the creation and setup of a Hibernate JPA Provider, including the provision of the datasource dataSource().

The EntityManagerFactory is responsible for identifying the JPA Entities to be made available, the classes to be treated as database mapping/ persistence beans.

The method EntityManager entityManager() creates the core class of JPA. EntityManager is the public interface of JPA, providing methods to persist, delete, update and query, and is used in the tests below for this purpose.

transactionManager() initialises the JPA transaction manager. This integrates with the declarative Transaction Management features of Spring, permitting the use of @Transactional and associated classes and configuration, for more information, [click here](http://static.springsource.org/spring/docs/3.2.4.RELEASE/spring-framework-reference/html/transaction.html)

Spring provides an exception translation framework to translate exceptions from many different sources into a consistent set that your application can use. In this case, the JPA Configuration expects a bean that provides these translations, which is provided by hibernateExceptionTranslator()

@Entity(name = "NOODLE\_ORDERS") declares this class as a JPA *Entity*. This is a class that is mapped to a database and able to be consumed by EntityManager.

@Column(name = "SUBMISSION\_DATETIME") is a JPA customisation that alters the name of the column this field will be mapped to. The default is the name of the field converted from lower camel (aFieldName) to uppsercase underscore case (A\_FIELD\_NAME)

## Implement a CRUD repository

Now that your JPA Order entity is working properly, you can now implement a repository to work with your entity.

Taking the same approach that you applied earlier for MongoDB, Spring Data provides a way to automatically create JPA backed repositories and fully fledged queries given only an interface as a starting point.

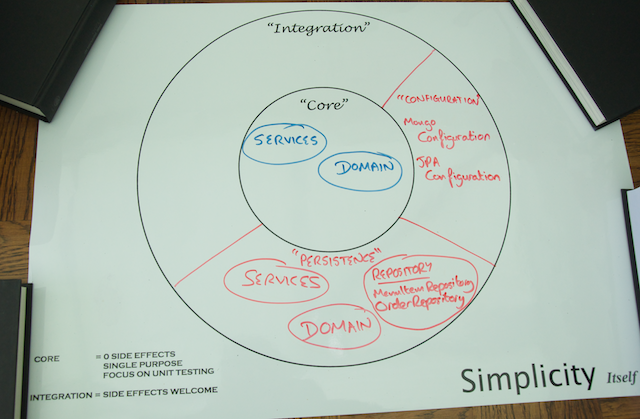
Create a new test class to test the repository that you've not written yet:

## Extend the Repository with a Custom Finder

Users need to be able to find Orders that contain certain menu items by Menu Item ID.

## Summary

Order data is safely stored and retrievable using a JPA managed relational database. You've added an OrdersRepository component and a JPAConfiguration component to your application, as shown by the following Life Preserver diagram:



# Step 4: Storing the Order Status in GemFire using Spring Data GemFire

Now it's time to look at how the status of your various orders will be stored, and for that task you're going to use GemFire.

These statuses will be coming into the application from the kitchen and order processing side of the business as opposed to the orders themselves that will come from the system that accepts orders from clients.

## A Word on GemFire

[Pivotal GemFire](http://gopivotal.com/pivotal-products/pivotal-data-fabric/pivotal-gemfire) is a high-performance distributed data grid. It scales from a small embedded cache implementation to large-scale wide area network implementations with data residency and access control.

[Spring Data GemFire](http://projects.spring.io/spring-data-gemfire/) allows the creation of both server and client connections, data access, caching and deep integration with the Spring Application Context.

In this step you will create a Spring Data interface to a GemFire server and then extend this our to use Continuous Queries.

## Import Spring Data GemFire

## Run a GemFire Cache Server

In order to run the tests and perform GemFire development you need to have access to a GemFire server.

While it would be possible to download a full distribution for the purposes of this tutorial, instead you're going to set up a server within this project.

This configures a basic GemFire server and creates a Region, a logical partition within GemFire, that we have named 'YummyNoodleOrder'.

This server will have access to the classpath of the project, most importantly the OrderStatus class. It is necessary for the GemFire server to have access to this class if we want to persist it within the grid.

When you create a standalone GemFire grid you need to provide any classes you wish to persist within a jar file on the classpath of every GemFire server.

## Start with a (failing) test: Introducing GemFireTemplate

In a similar way as with MongoDB and JPA, the first test you need to write checks that OrderStatus can be correctly persisted into GemFire.

This test uses GemFireTemplate, seen in this test via its API interface GemFireOperations. This follows the same pattern as other Spring Template classes, exposing the most common operations using consistent, simple methods, and also providing access to the low level GemFire API in a managed way through the use of callbacks.

## Implementing a CRUD repository

You have seen the creation of two Repository implementations against both MongoDB and JPA and the process for creating a Spring Data GemFire Repository will seem very familiar!

This test generates a new OrderStatus with a known key and passes it to OrderStatusRepository for persisting. It then retrieves the data using the method findOne, which will query against the key that is passed into the GemFire Region Map structure.

Data is being managed explicitly in the test rather than using the declarative transaction management that was introduced in the JPA tests.

While GemFire does integrate with the Spring provided transactions, it only supports **Isolation.READ\_COMMITTED**. This means that once you write data, it cannot be read, by any thread or process, until the surrounding transaction is committed. Any test that wrote data within a transaction would be unable to read it until the transaction finished.

For this reason the test is not marked as @Transactional so all data access will not be transactionally managed within the tests. At the start and end of the test, the region is purged by using the repository deleteAll method generated by Spring Data.

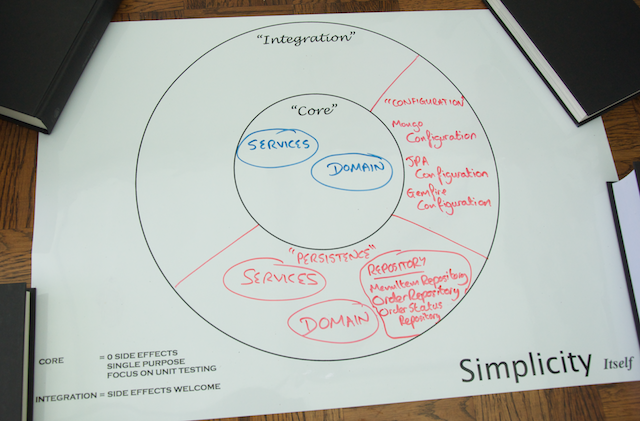
## Extend the Repository with a Custom Finder

An Order has a history of the status updates made to it. A history is a list of OrderStatus objects in date order

You now need a more complex query than simply by ID or Order ID. You'll also need to sort by date.

## Summary

In this step you've added the functionality to store OrderStatus data using GemFire.



# Step 5: Extending the Persistence Domain to Send Events

## Data Grids and Continuous Queries

Usually an application would have to regularly poll to receive updated data from a data store. This is inefficient as many queries will be executed unnecessarily and also introduces latency in between polls. More often than not applications will likely introduce some separate messaging infrastructure, such as [RabbitMQ](http://rabbitmq.com), to distribute notifications.

GemFire is a distributed data grid. It can be clustered and provides a feature called Continuous Querying for just this circumstance.

Continuous Querying allows you to register a GemFire Query with the cluster and then for a simple POJO to receive events whenever a new piece of data is added that matches your query.

## Writing a continuous query

Whenever an OrderStatus instance is saved into GemFire, the method OrderStatusUpdateService.setOrderStatus() needs to be called with an appropriate event.

First, create a stub implementation of OrderStatusUpdateService. This stub will receive events and count them off against a [CountDownLatch](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CountDownLatch.html) to ensure that the correct number of events are received in the given time. When all expected threads submit their countdown, the latch proceeds to completion.