Understanding REST

## 背景

REST (Representational State Transfer) was introduced and defined in 2000 by Roy Fielding in his [doctoral dissertation](http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm). REST is an architectural style for designing distributed systems. It is not a standard but a set of constraints, such as being stateless, having a client/server relationship, and a uniform interface. REST is not strictly related to HTTP, but it is most commonly associated with it.

## Principles of REST

* **Resources** expose easily understood directory structure URIs.
* **Representations** transfer JSON or XML to represent data objects and attributes.
* **Messages** use HTTP methods explicitly (for example, GET, POST, PUT, and DELETE).
* **Stateless** interactions store no client context on the server between requests. State dependencies limit and restrict scalability. The client holds session state.

## HTTP methods

Use HTTP methods to map CRUD (create, retrieve, update, delete) operations to HTTP requests.

### GET

Retrieve information. GET requests must be safe and [idempotent](http://en.wikipedia.org/wiki/Idempotence#Computer_science_meaning), meaning regardless of how many times it repeats with the same parameters, the results are the same. They can have side effects, but the user doesn't expect them, so they cannot be critical to the operation of the system. Requests can also be partial or conditional.

Retrieve an address with an ID of 1:

GET /addresses/1

### POST

Request that the resource at the URI do something with the provided entity. Often POST is used to create a new entity, but it can also be used to update an entity.

Create a new address:

POST /addresses

### PUT

Store an entity at a URI. PUT can create a new entity or update an existing one. A PUT request is idempotent. Idempotency is the main difference between the expectations of PUT versus a POST request.

Modify the address with an ID of 1:

PUT /addresses/1

### DELETE

Request that a resource be removed; however, the resource does not have to be removed immediately. It could be an asynchronous or long-running request.

Delete an address with an ID of 1:

DELETE /address/1

## HTTP status codes

Status codes indicate the result of the HTTP request.

* **1XX** - informational
* **2XX** - success
* **3XX** - redirection
* **4XX** - client error
* **5XX** - server error

## Media types

The Accept and Content-Type HTTP headers can be used to describe the content being sent or requested within an HTTP request. The client may setAccept to application/json if it is requesting a response in JSON. Conversely, when sending data, setting the Content-Type toapplication/xml tells the client that the data being sent in the request is XML.

# Designing and Implementing RESTful Web Services with Spring

It's likely that you want to implement a RESTful web service because:

* You're creating an API that clients need to consume across the web.
* You want to open up your organization's data to consumption by varied clients across the web.
* You need to integrate your application with other applications inside your own organization, but you don't have control over the languages, tools, or frameworks for those applications.

## What you'll need

* An installation of the [Gradle](http://www.gradle.org) build tool, version 1.6 or later.

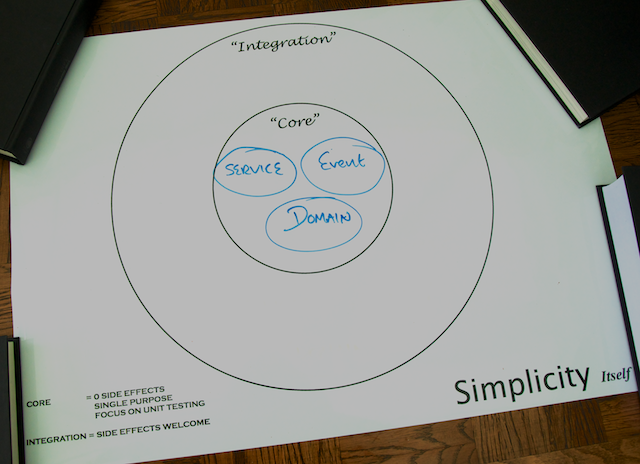
## Running the code

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

## Yummy Noodle Bar application architecture and the Core domain

The current architecture of the application is shown in the following "Life Preserver" diagram:



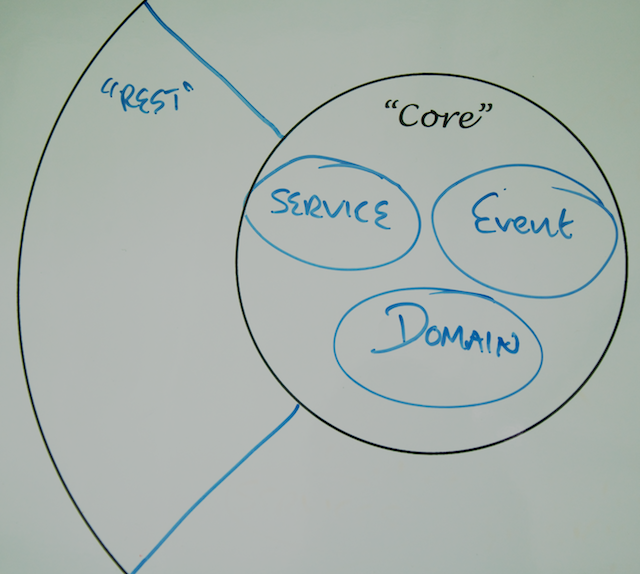
The Life Preserver diagram is a tool for building applications that following the principles of the [**Hexagonal Architecture, sometimes referred to as 'Ports and Adapters' originally characterised by Alistair Cockburn**](http://alistair.cockburn.us/Hexagonal+architecture). The Life Preserver diagram shows your application's core internal domains along with the surrounding 'integration' domains that map directly to the packages and components that you'll be working on throughout this tutorial, so it's a great way to understand where things are.

Under the core application's top-level packages, com.yummynoodlebar.core, here's what the packages contain:

* **domain**. Components that cleanly capture the application's Core domain concepts. These classes are a manifestation of the [ubiquitous language](http://martinfowler.com/bliki/UbiquitousLanguage.html) of the Core domain.
* **repository**. Components that store and retrieve the current state of the system's domain objects.
* **event**. Components that are the events that the domain can receive and process.
* **service**. Components that handle the actions that can be performed when an event is received.

## RESTful web services domain

RESTful web services integrate your application and the all of the possible clients that need to consume your services. As such, RESTful services live in their own integration domain, outside your application's core, as shown in the following update to your life preserver.



Given the integration between your application and the outside world, consider the following design and implementation constraints:

* Your RESTful service API is your focus; the core application structure should not influence the design of the API.
* The components that make up your RESTful services need to evolve at a rate that is appropriate for the many consumers that rely on your services.
* Your RESTful service components should not contain any core logic for your application, but they will collaborate with other components in the Core domains of your application in order to orchestrate the necessary functionality for the service interface.

## Step 1: Modelling the Core and RESTful Web Service Domains

For the first version of your new Yummy Noodle Bar RESTful service, the ability to create, update, and remove Orders is the focus.

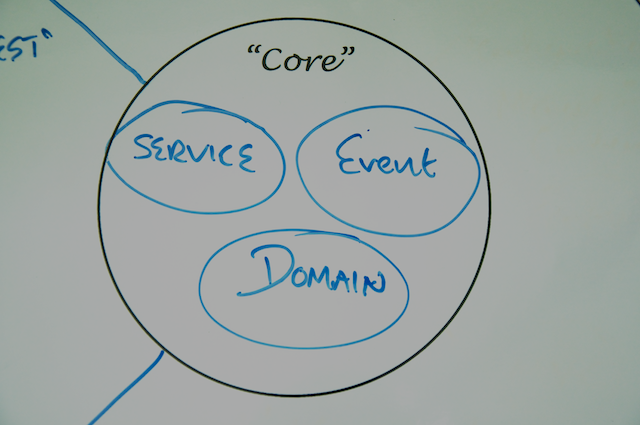
It is tempting simply to expose the Core Order domain to the outside world and work from there, but that would ignore the boundary between the Core and the RESTful service domain and would lead to the REST API being driven by the internal application structure, and so becoming coupled to that internal structure.

The public API of your service (the RESTful service domain) that you will expose to clients needs to change at a rate that is friendly to those clients. The Core needs to evolve at a rate that corresponds to the Yummy Noodle Bar system's need to evolve internally. Potential friction exists between the two domains as they may need to evolve at different rates.

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

In the Core domain the concepts are captured as part of the internal ubiquitous language of the application's domain. In the REST domain the concepts are captured as they are used purely for the purpose of exposing the public RESTful interface.

## Components of the Core application domain for Yummy Noodle Bar



Open the initial project. Under src/main/java/com/yummynoodlebar/core/domain, you see the components of the core, application-internal domain of Yummy Noodle Bar:

* **Order**. An individual order in the system that has an associated status and status history for tracking purposes.
* **OrderStatus**. Current status allocated to an Order.
* **Payment**. Payment that a customer wants to make for a given Order.
* **PaymentDetails**. Details of the Payment that a customer wants to make for a given Order.
* **PaymentStatus**. Current status of a Payment that a customer wants to make for a given Order.

This tutorial focuses on the Order domain class, which can be acted upon by a number of events under the com.yummynoodlebar.events.orders package as shown on the following diagram



Events in this case decouple the domain concepts in the core of the Yummy Noodle Bar application from the various integrations that may need to access and work upon the core.

The event components associated with Orders include:

* **RequestAllOrdersEvent** and **AllOrdersEvent**. Corresponding events to request the associated OrderDetails about all Orders and the response to that request.
* **CreateOrderEvent** and **OrderCreatedEvent**. Corresponding events to request the creation of a new Order, and a confirmation that the new Order has been created.
* **DeleteOrderEvent** and **OrderDeletedEvent**. Corresponding events to delete an existing Order and then to confirm that the Order has been deleted.
* **RequestOrderDetailsEvent** and **OrderDetailsEvent**. Corresponding events to request the current details of an Order, and then to receive those details.
* **RequestOrderStatusEvent** and **OrderStatusEvent**. Corresponding events to request the current status of an Order, and then to receive the current status.
* **SetOrderPaymentEvent**. Triggered when Payment is to be set on an existing Order.
* **OrderUpdatedEvent**. Triggered when an Order is updated.