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.

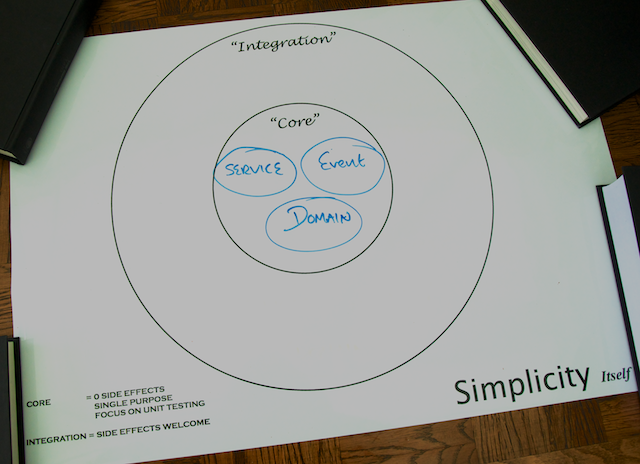
## Running the code

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:



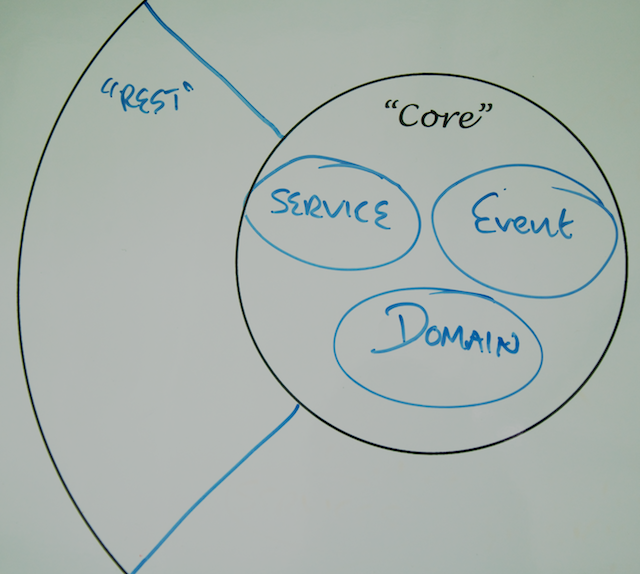
The Life Preserver diagram is a tool for building applications that following the principles of the [**Hexagonal Architecture, sometimes referred to as 'Ports and Adapters' originally characterised by Alistair Cockburn**](http://alistair.cockburn.us/Hexagonal+architecture). The Life Preserver diagram 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.

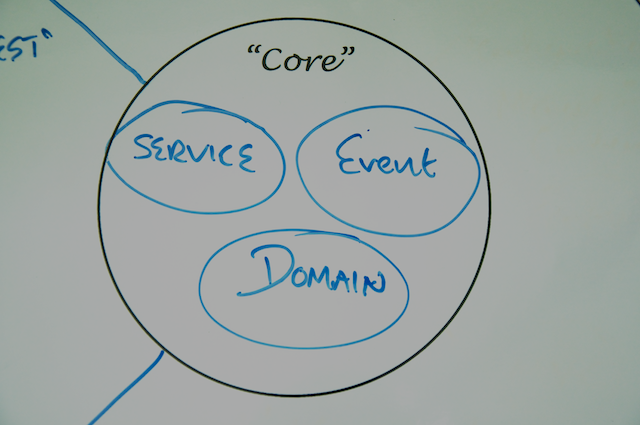
It is tempting simply to expose the Core Order domain to the outside world and work from there, but that would ignore the boundary between the Core and the RESTful service domain and would lead to the REST API being driven by the internal application structure, and so becoming coupled to that internal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

## Model the orders and order resources

There are three stages to modelling your RESTful service domain:

* Design your resources. What resources do you need to expose to the outside world?
* Design your URIs. How will your resources be publicly addressed?
* Add operations. What operations can you perform on your RESTfully exposed resources?

### Design your resources

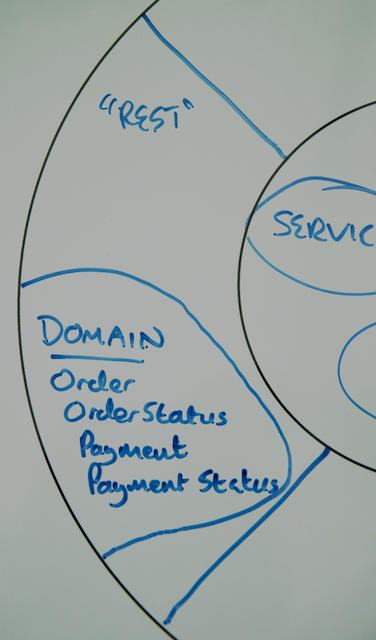
To determine the resources that you will support through your RESTful service, look for the relevant nouns in your domain. In the case of the Yummy Noodle Bar, the following nouns are candidates for exposure to the outside world from the core domain:

* Customer
* Order
* OrderStatus
* OrderStatusHistory
* Payment
* PaymentDetails
* PaymentStatus

The purpose of the Yummy Noodle RESTful service is to submit and track orders as they are executed and delivered. Therefore, we don't need to expose all of these domain concepts via REST. Instead we can focus on exposing the following subset:

* Order
* OrderStatus
* PaymentDetails
* PaymentStatus

The updated life preserver shows where these domain components will live in the design.



As mentioned before, although these concepts exist in both the Core domain and the RESTful Service domain, this is not duplication as the purpose of the implementations are very different from Core to REST.

### Design your resource URIs

While these resources will be represented by a class in code, you must first decide the REST API. That is because the REST API will drive the design of code and the implementation of the resources.

Each resource needs to be addressable by a URI. In addition, the address implies the relationship between each of the resources.

For your Yummy Noodle Bar RESTful Service domain, the resources will have the following URIs:

All Orders:

http://www.yummynoodlebar.com/aggregators/orders

One Order

http://www.yummynoodlebar.com/aggregators/orders/{Order ID}

Current OrderStatus for one Order

http://www.yummynoodlebar.com/aggregators/orders/{Order ID}/status

PaymentDetails for one Order

http://www.yummynoodlebar.com/aggregators/orders/{Order ID}/paymentdetails

PaymentStatus for one Order

http://www.yummynoodlebar.com/aggregators/orders/{Order ID}/paymentstatus

#### URI templates

Each of the above URIs are expressed as templates; they contain blocks demarcated with {} in the URI. Because the URI in a RESTful service should completely address the resource, without any additional query parameters, parts of the URI are specific to the resource itself.

For example, here the {} notation specifies where an Order with Order ID of 1 would have the following specific URL once the URI template is furnished with the Order Number:

http://www.yummynoodlebar.com/aggregators/orders/1

An Order with an Order ID of 37 would have the following specific URI:

http://www.yummynoodlebar.com/aggregators/orders/37

This quality of the URI changing to work with specific resources is what gives a resource the quality of being addressable.

### Add the operations (HTTP methods)

Along with the URLs you're going to expose for the RESTful service for each resource, you also need to decide what can be done to each resource.

RESTful services rely on the HTTP methods passed as part of an HTTP request header to tell the service what needs to be done with the addressed resource. The full set of HTTP methods allowed constitute the [uniform interface](http://en.wikipedia.org/wiki/Representational_state_transfer) to RESTful services.

The following methods are supported.

|  |  |
| --- | --- |
| **HTTP Method** | **Description** |
| GET | Retrieves a representation of the resource addressed by the URI used to submit the HTTP request upon. |
| POST | Creates a new resource under the URI used to submit the POST HTTP Request upon. |
| PUT | Updates the resource indicated by the URI used to submit the HTTP request upon. If a resource did not already exist at the specified URI, then a new resource at the specified address will be created. |
| DELETE | Removes the resource from the system where it is addressed by the URI used to submit the HTTP request upon. |
| HEAD and OPTIONS | Retrieve various meta-data about the resource addressed by the URI used to submit the HTTP request upon. |

The [HTTP 1.1 Specification](http://www.w3.org/Protocols/rfc2616/rfc2616.html) provides a detailed description of all the [HTTP Methods](http://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html#sec5.1.1).

POST and PUT have similar but not identical functions. You use POST to create new entities without knowing the final URI, and you use PUT to create and update entities in a previously known URI.

The following table describes what HTTP methods will be supported for each resource address.

|  |  |  |
| --- | --- | --- |
| **Resource URI** | **Supported HTTP Methods** | **Description** |
| /aggregators/orders | GET | Asks for a representation of all of the orders. |
| /aggregators/orders | POST | Attempt to create a new order, returning the location (in the Location HTTP Header) of the newly created resource. |
| /aggregators/orders/{id} | GET | Asks for a representation of a specific Order resource. |
| /aggregators/orders/{id} | DELETE | Requests the deletion of a specified Order resource. |
| /aggregators/order/{id}/status | GET | Asks for a representation of a specific Order's current status. |
| /aggregators/order/{id}/paymentdetails | GET | Asks for a representation of a specific Order's payment details resource. |
| /aggregators/order/{id}/paymentdetails | PUT | Updates a specific Order's payment details resource. |

## Capture status codes

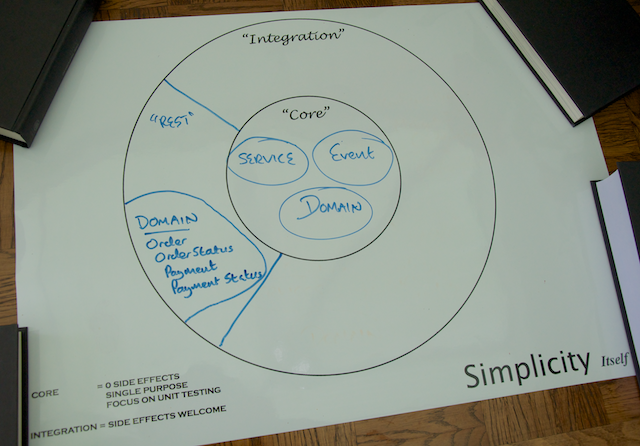
No RESTful design is complete without considering responses to requests. Now is the time to capture the [HTTP Status Codes](http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html) your service will respond with for a given combination of URI and HTTP method on a request.

The following table describes the HTTP status codes that each of your URIs and HTTP method combinations will respond with.

|  |  |  |
| --- | --- | --- |
| **Resource URI** | **Supported HTTP Methods** | **Supported HTTP Status Codes** |
| /aggregators/orders | GET | 200 (OK, Success) |
| /aggregators/orders | POST | 201 (Created) if the Order resource is successfully created, in addition to a Location header that contains the link to the newly created Order resource; 406 (Not Acceptable) if the format of the incoming data for the new resource is not valid |
| /aggregators/orders/{id} | GET | Asks for a representation of a specific Order resource. |
| /aggregators/orders/{id} | DELETE | 200 (OK); 404 (Not Found) if Order Resource not found |
| /aggregators/orders/{id}/status | GET | 200 (OK); 404 (Not Found) if Order Resource not found |
| /aggregators/orders/{id}/paymentdetails | GET | 200 (OK); 404 (Not Found) if Order Resource not found |
| /aggregators/orders/{id}/paymentdetails | PUT | 201 (Created); 406 (Not Acceptable) if there is a problem with the format of the incoming data on the new payment details; 404 (Not Found) if Order Resource not found |

## Summary

Congratulations! You've determined the resources you're going to expose and captured those in the REST domain as shown in the following Life Preserver:



# Step 2: Building Your First RESTful Web Service with Spring MVC

You've decided on your RESTful service's resources and captured them on your Life Preserver as the RESTful domain's components:

It's time to implement your Yummy Noodle Bar RESTful web service. The first step in building a service with Spring MVC is to construct and test one or more controllers that are responsible for handling each incoming HTTP request that you defined for your service in the previous step.

## Start with a (failing) test

[Test Driven Development (TDD)](http://en.wikipedia.org/wiki/Test-driven_development) teaches us that if you haven't got a failing test then there's no code to write! So before you dive into implementing the service, create a few tests that justify and encourage you to write some code to make the test pass.

### Separate commands from queries

Before you start creating tests, consider the categories of requests that your service will respond to. You are going to be writing tests that look for all the HTTP RESTful interactions that you designed in [Step 1](http://spring.io/guides/tutorials/rest/1/).

These interactions can be split into five categories:

* Requests that read, or query, the current state of an Order
* Requests that change the state of a new or existing Order
* Requests that read an Order's status
* Requests that read an Order's status
* Requests that change the state of an Order's Payment Details

You can separate these interactions into two categories:

* Requests that change a resource's state (a Command)
* Requests that query a resource's state (a Query)

It's possible to implement these two categories of interactions using one controller for each resource. However, the [Command Query Responsibility Segregation (CQRS)](http://martinfowler.com/bliki/CQRS.html) pattern advises you to split these responsibilities into different routes through your application. In this tutorial you will implement these concerns separately.

### Implement failing test(s) for a controller with MockMVC

You won't implement all tests needed for your RESTful service here; the full breadth of tests needed would confuse things. You can see the full listing in the complete project. Instead you use two unit tests that look for an example of each category of interaction through the RESTful service, commands, and queries.

#### Create placeholders and starter Order

First, create two empty classes

com.yummynoodlebar.rest.controller.OrderCommandsController and com.yummynoodlebar.rest.controller.OrderQueriesController. These are left as placeholders so the tests written below can compile. against them.

#### Set up test fixtures

You will need some test data to use in the tests. A test fixture is a well known piece of test data that is extracted into its own setup method or class. You will need to create two test fixtures.

#### Test GET HTTP method HTTP requests

The first test ensures that a request to view an order's details is possible, so call the class ViewOrderIntegrationTest

Why an integration test? Because you're going to be testing the controller within the constraints of a mock Spring MVC environment. This way you can test the mappings of your incoming requests to the appropriate handler methods while still getting the benefits of testing a real container.

Next, add an instance of MockMvc to the test class and set up a mock controller and OrderService.

In the @Before annotated setup() method, you set up Mockito and generate a mock Spring MVC environment, including adding JSON message conversion, because you expect JSON when you ask for the current state of an Order.

MockMvc is a relatively new part of Spring MVC. It provides a test method with a Spring MVC Controller, including all of its annotations, routing, and URI templates. It does this by initializing the MVC Controller classes in a full MVC environment, including the DispatcherServlet and then running assertions against that. The only piece missing from this testing puzzle is the web context itself, which is covered in [Step 4](http://spring.io/guides/tutorials/rest/4/).

Finally you can implement a test method that performs an HTTP Request on the controller and asserts that the response from that invocation contains the JSON that was requested.

Now look at the final call in the above method, the usage of MockMvc, in a little more detail.

The mockMvc object is performing the following tasks, in sequence:

* Perform a mock HTTP Request with a GET HTTP Method on the URI **/aggregators/orders/{id}**.
* Replace the {id} marker in the URI template with the contents of the response to the key.toString() call.
* Specify in the 'accept' HTTP Header that the service should respond with JSON.
* Analyze the content of the returned JSON to ensure that some mocked data is present, as provided by the mock collaborators that were set up at the start of the test method.

The Spring MockMvc component makes it possible to do this testing where you can be sure that for a given URI, a given rendered content in the response will be returned, all executed in a unit test from within your IDE or Continuous Integration environment.

#### Test DELETE HTTP method HTTP requests

Take a look at a test implemented in exactly the same fashion, but performing the job of canceling an Order by sending a HTTP request with a DELETE HTTP method to the Order's URI:

The main difference with this test is that no content is returned from the mock HTTP Request performed using mockMvc. Instead you are using Mockito's verify behavior to ensure that your controller is making the appropriate deleteOrder call to the mock orderService in order for the test to pass.

#### Test POST HTTP method HTTP requests for creating resources

Take a look at how to test HTTP requests that contain POST as the HTTP method. Specifically, a POST creates a new resource and generates a new URI for that new resource, and so this URI generation also needs to be part of the test.

The focus here is the andExpect condition at the end of the perform call to mockMvc. Here you're testing that the response of the post has resulted in a new Location HTTP Header and that it contains a URI.

There are other test implementations in the section 2 complete project, so you can see how the rest of the tests for your RESTful interface is implemented. Of course at this point those tests will fail as you haven't created any corresponding controllers.

## Make the tests pass: Implement the controllers

You now have a collection of test classes that will fail, given that no controllers exist yet to respond to the mocked HTTP requests. It's time to focus on making the ViewOrderIntegrationTest, CancelOrderIntegrationTest and CreateNewOrderIntegrationTest tests pass.

### Implement the OrderQueriesController

Start by implementing the controller that is responsible for handling requests that simply read the current state of the Order resources. This controller will make the ViewOrderIntegrationTest tests pass.

Notice how a controller handler method implementation is kept very clean as all interactions with the underlying system occur via firing events into the core domain. It is a reasonable design goal to avoid business logic in your controllers and delegate that responsibility to a collaborating component.

As you can see from the @RequestMapping annotation, the viewOrder handler method is mapped to a URI that is constructed from a combination of the controller's default URI, **/aggregator/orders**, combined with the template parameter of {id} to make the complete mapping for this method **/aggregator/orders/{id}**.

The {id} parameter is then mapped as a String into the viewOrder method. Since this is a read-only query request, the @RequestMapping specifies that this method can only be called for HTTP GET requests.

Finally the handler method needs to return the content that the client requests. Spring's REST support provides an easy way to not have to implement this yourself. You use @ResponseBody so that objects returned from the method are marshaled directly into the expected content.

Those are all the handler methods you need to directly request information about the current state of your service's Orders as specified by the ViewOrderIntegrationTest. The next step is to support that state being manipulated by requests that instigate commands.

### Implement the OrderCommandsController

Next you need to implement a method that handles an HTTP request that carried a DELETE HTTP method, targeting a specific Order resource. The following code snippet shows that handler method

The cancelOrder method needs to deal with conditions in addition to a simple call to see a representation of an Order. For example, there's the possibility that there is no Order with the indicated ID.

To vary the response code to a handler method, you use the ResponseEntity class. In the example above, the ResponseEntity objects afford you the opportunity to return an HTTP Status code of 403 (Forbidden) if an attempt is made to cancel an Order that does not exist.

The OrderCommandsController also needs to deal with the case where a new Order resource is being created using an HTTP request that contains an HTTP POST method.

The major difference here from the previous controller method implementation is that you're using the ResponseEntity return object to also set the HTTP Headers. This is necessary as you need to return the newly generated URI for the newly created Order resource.

## Where did the JSON representations come from?

Now when you run the tests in the example project you'll find that they all pass. But wait a second; how did those tests pass when they look for JSON content and you haven't specified how that is being rendered?

In traditional Spring MVC there would be a ViewResolver and a specific View to render content as HTML in a browser. With a RESTful service, it is much more common to render the returned object from a handler method *as the content itself* and so a view is rarely needed.

The secret of how things are working here is in looking at the dependencies that the project itself has. If you look in the build.gradle file in the project's root directory you should see the following entries in the project dependencies:

runtime 'com.fasterxml.jackson.core:jackson-core:2.2.2'

runtime 'com.fasterxml.jackson.core:jackson-databind:2.2.2'

Now when you run the tests in the example project you'll find that they all pass. But wait a second; how did those tests pass when they look for JSON content and you haven't specified how that is being rendered?

In traditional Spring MVC there would be a ViewResolver and a specific View to render content as HTML in a browser. With a RESTful service, it is much more common to render the returned object from a handler method *as the content itself* and so a view is rarely needed.

The secret of how things are working here is in looking at the dependencies that the project itself has. If you look in the build.gradle file in the project's root directory you should see the following entries in the project dependencies:

build.gradle

runtime 'com.fasterxml.jackson.core:jackson-core:2.2.2'

runtime 'com.fasterxml.jackson.core:jackson-databind:2.2.2'

These two dependencies are enough for Spring MVC to take classes defined in your RESTful domain and render those objects as JSON according to the content type requested by the client.

This scenario might be new to someone coming from traditional web application development. It is normal in traditional web application development for a browser to send a plethora of possible options as part of the content type negotiation on a given HTTP request. The server will then decide what content to return from that large set of possibilities.

With a RESTful service it is much more typical for a client to ask for a specific content type for a returned representation. Rather than your controller declaring a specific view to render, the Spring MVC content negotiation is invoked according to what content type is requested by the client.

In this test environment, JSON is being requested. But what if another content type is requested? For example, perhaps the client prefers XML?

### Use JAXB to marshall objects into content

The first thing to notice in the tests is that the mockMvc object is being set up to support both XML and JSON. This works only if the appropriate jar files are on the classpath. A glance in build.gradle shows the following dependencies to support JAXB2 rendering of XML representations:

runtime 'javax.xml.bind:jaxb-api:2.2.9'

All good so far, but XML marshalling from Java objects is a little more involved than JSON. Here you're using JAXB2, and so in addition you'll need to annotate your REST domain classes so that the additional metadata to marshall the right XML is supplied. Open the Order class in com.yummynoodlebar.rest.domain and add the @XmlRootElement annotation, as below:

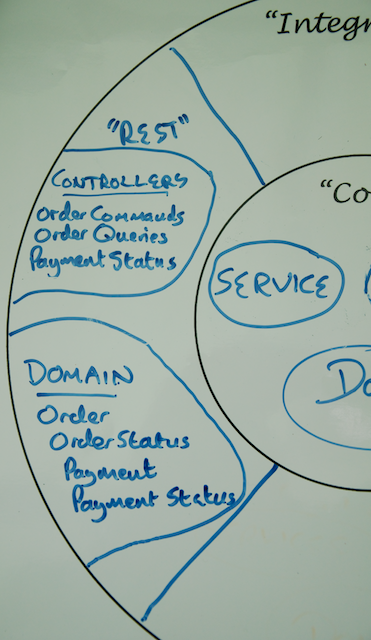
@XmlRootElement

public class Order implements Serializable {

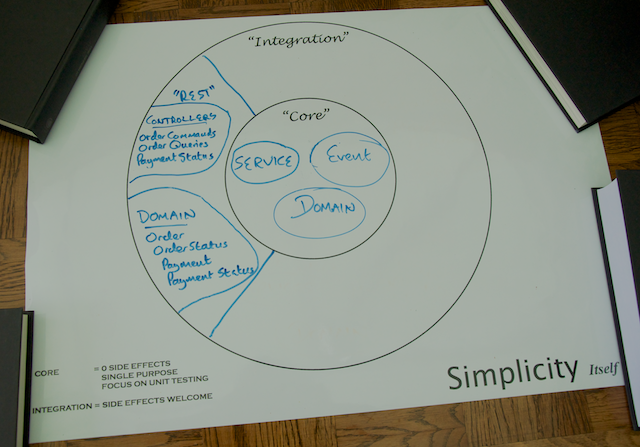
## Summary

Congratulations! You've created controllers that can implement your RESTful service's API. You've tested those controllers using 'MockMVC' outside of a container to confirm that the handler mappings work. And you verified that your controller will react to the right forms of HTTP requests with the right types of content.

Your Life Preserver now contains a whole new set of components, your controllers, in the RESTful domain:

**

The full view of your current Life Preserver should look like the following:

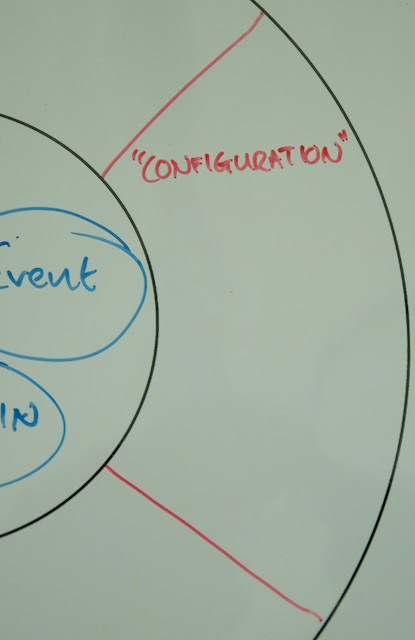
**

# Step 3: Configuring and deploying your service

At this point you are ready to:

* Configure the core of your application
* Configure your REST components
* Initialize your RESTful Service's web infrastructure
* Run your RESTful service in a web container

To complete these tasks, you'll need a new domain, the Configuration domain.

**

As with core and rest, the Configuration domain will have its own code package com.yummynoodlebar.config.

## Create a configuration for your application's Core domain using Spring JavaConfig

The Yummy Noodle Bar application contains a core set of components that include domain classes and services.

You could just create a configuration for these components; however, as in the previous step, you'll apply the Test Driven Development approach to your configuration.

### Test your Core configuration

First, create an empty placeholder class com.yummynoodlebar.config.CoreConfig

This integration test constructs an ApplicationContext using JavaConfig as specified on the @ContextConfiguration annotation. The Core domain's configuration will be created using Spring JavaConfig in a class called CoreConfig.

With the ApplicationContext constructed, the test can have its OrderService test entry point autowired, ready for the test methods.

Finally you have one test method that asserts that an orderService dependency has been provided and appears to work correctly.

Next, create the Core domain configuration.

### Implement your Core domain configuration

The Core domain configuration for the Yummy Noodle Bar application only contains one service and one dependency that needs to be configured for that service.

Spring JavaConfig will detect each @Bean annotated method as a method that generates configured Spring Beans.

Spring will create the OrdersRepository bean first, and then use that as the single parameter into the createService method to create the OrderService bean.

Running the CoreDomainIntegrationTest in the com.yummynoodlebar.config test package will verify that your Core Domain configuration is good to go.

## Create a configuration for your REST components

Configuring your new set of controllers is very straightforward as you have used @Controller on each of the controller classes. To initialize your RESTful domain's components, all you need to do is turn on component scanning so that Spring can find and initialize these Spring beans.

### Implement your RESTful domain configuration

You can create the following Spring JavaConfig to execute component scanning for the components in your application's RESTful domain:

The @ComponentScan attribute in JavaConfig specifies that your components should be found underneath the base Java package of

com.yummynoodlebar.rest.controllers

It's always a good idea to be as specific as possible when defining the place where component scanning should occur so that you don't accidentally initialize components you didn't expect!

### Test your RESTful domain configuration

No configuration should be trusted without an accompanying test. The following test asserts that the output of the RESTful configuration is as it should be:

You've already asserted the correctness of the collaboration between your controllers and the underlying service components in the Core Domain.

This test ensures that once everything is wired together, the wiring in the MVCConfig is correct and the appropriate controllers are in attendance.

The test validates the MVCConfig by mocking requests which exercise the handler mappings. The full responses are also confirmed to be correct. More testing could be done, but you've already asserted that your controllers should work appropriately in the previous steps. This test is simply there to show you that now you are configuring those components using Spring JavaConfig properly.

## Initialize your RESTful service web infrastructure

As of Spring 3.2, if you're using a web container that supports the Servlet 3 specification such as Tomcat 7+, it's possible to initialize the underlying web infrastructure for your application without writing a line of XML.

Here you're going to use the WebApplicationInitializer to set up your application's web application context parameters to bootstrap your application's web infrastructure as shown in the following code.

First you create a new piece of configuration as a class inside com.yummynoodlebar.config called WebAppInitializer that extends the WebApplicationInitializer from Spring as shown below

The LOG attribute shows that you can even log messages as your web infrastructure is initialised, despite having no XML settings.

Next you override the onStartup method which in turn sets up your root Spring Application Context by calling createRootContext and then finally request the configuration of SpringMvc by calling configureSpringMvc.

The root Spring Application Context will contain the majority of your components including your Core Domain. In the createRootContext method an instance of the AnnotationConfigWebApplicationContext class is constructed and then configured by calling register indicating the JavaConfig classes to be applied. In your case the root context can be initialised simply with the CoreConfig class.

Now with a root Application Context already to hand, in the configureSpringMvc method you can configure the REST Domain components in a new AnnotationConfigWebApplicationContext application context, connecting this new context to the root application context so that your REST Domain components can see and be dependency-injected with components from the root application context

Finally, using the servletContext you can dynamically initialise the Spring MVC DispatcherServlet, in this case mapping the DispatcherServlet to the root of the newly registered application.

The DispatcherServlet is a 'front controller' servlet that receives all incoming requests that should be considered for the various controllers registered. The DispatcherServlet then is the overall orchestrator of how each incoming request is channelled to the appropriate handler method on the available controllers.

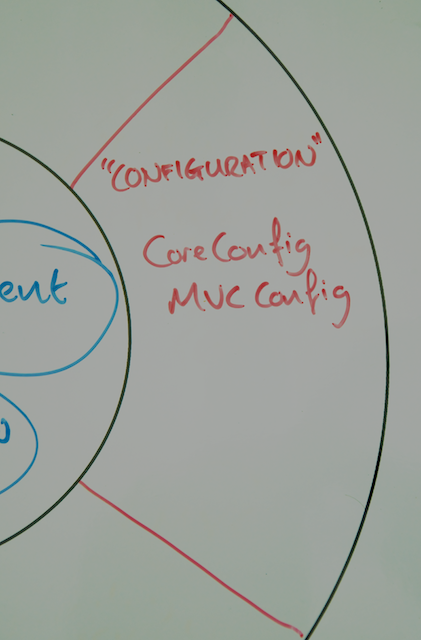
## Running your RESTful service in a Web Container

It's the moment of truth: can you execute your new RESTful service?

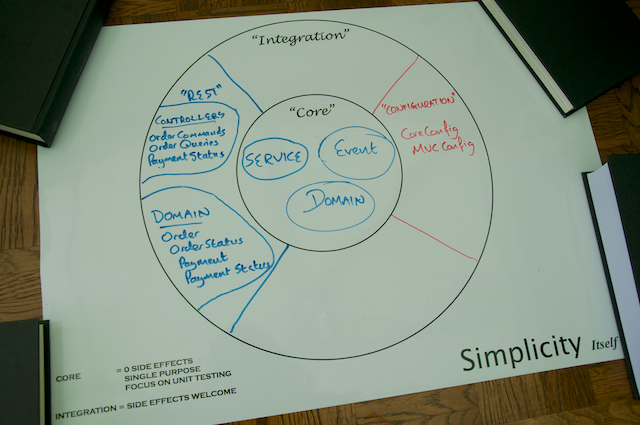
## Summary

You've come a long way! You've now got a fully configured RESTful web service that is running in Tomcat and can be packaged for distribution in a WAR file.

You've added two new components to your Configuration domain, CoreConfig and MVCConfig as shown in the updated life preserver below.



Your full Life Preserver should now look like the following:



But how do you really know that when you've deployed your service it really works? That's the job of functional testing, and that's your task in the next section of this tutorial.

# Step 4: Testing your Service with Spring's RestTemplate

Full-stack functional tests provide the last check-in-the-box proving your service is fully integrated and functional.

There should be as few tests as possible to prove that the full stack works. Ideally you would have just one: testing your 'happy path', or the code path that **must work** for you to make money

You'll use Spring's RestTemplate to create functional tests. Following the above rule, you test one code path, placing orders, which is the core part of the REST interface. If this doesn't work, Yummy Noodle Bar will go out of business.

## Implement a functional test with RestTemplate

Here you're setting up an initial Spring RestTemplate. You've created HTTP Headers that set the content type and accept header to JSON in anticipation of receiving JSON in the response.

Next you prepare an HTTPEntity to contain the request that you'll submit to your service:

This HttpEntity uses a test fixture (RestDataFixture) to generate some JSON data that will form the content of your POST request to your service.

## Summary

The Spring RestTemplate is a powerful means of functionally interacting with your RESTful services, regardless of your testing framework. Here you've implemented a minimal set of functional tests, looking specifically at ensuring that new Order resources can be created. Your REST service is moving closer to production use by all those high-profile aggregators.

There's just one hitch. How do you ensure that only legitimate aggregators can submit Orders to your system? It's time to secure your RESTful service