**Designing and Implementing a Web Application with Spring**

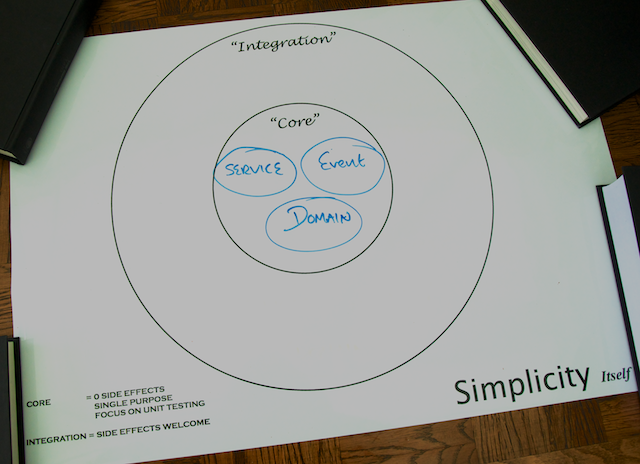
## Spring and the Web

The web has become a core part of our lives, from shopping to finding the closest ATM. Web applications, server software sending HTML over HTTP, implement the web.

Spring helps you build web applications that scale from a small internal application to those serving millions of users and thousands of concurrent requests.

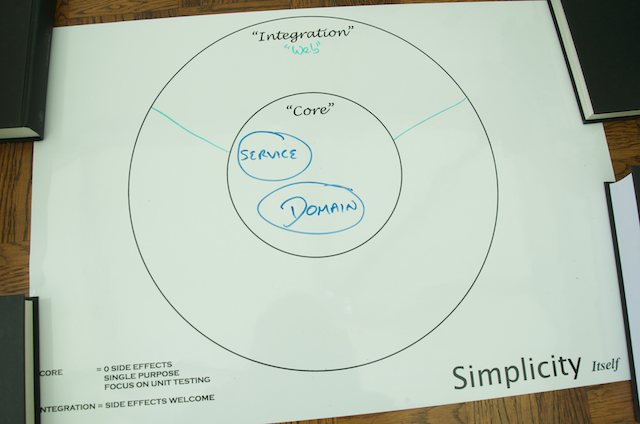
## Yummy Noodle Bar application architecture and the Core domain

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



## Web domain

A web front end integrates your application with web browsers and their users. As such, the web front end lives in its own integration domain on the periphery of your application's core, as show 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:

* The user experience (UX) is your focus; the core application structure should not influence the design of the web front end.
* The components that make up your Web domain need to evolve at a rate that is appropriate for the many consumers that rely on your services.
* Your Web 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 Web Domains

For the first version of your new Yummy Noodle Bar Web front end, the ability to view the Menu and create and monitor 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 Web domain and would lead to the Web front end being driven by the internal application structure, and so becoming coupled to that internal structure.

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 Web domain and would lead to the Web front end being driven by the internal application structure, and so becoming coupled to that internal structure.

To manage this friction you need to create concepts and components in the Web domain that are unique to, and can evolve at the rate needed by, the Web 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 Web domain the concepts are captured as they are used purely for the purpose of exposing the public Web front end.

### 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:

* **Customer**. A username, address and name that an Order will be delivered to.
* **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.

This tutorial focuses on the Order domain classes, 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 out 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 that you will use for the Web include:

* **CreateOrderEvent** and **OrderCreatedEvent**. Corresponding events to request the creation of a new Order, and a confirmation that the new Order has been created..
* **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.

### Model your Users interactions

When you are building a web application, the users you build it for are humans. While this may seem obvious, it has massive implications for the design and model of your Web domain.

Most importantly :

* Users expect to be able to visit any URL they see again. You should expect URLs to be copy and pasted.
* Users expect to move around a website arbitrarily.
* Users expect to use the back and forward buttons at will.
* The users experience of HTTP GET and POST (from HTML forms) is dramatically different. A POST should only be used for submitting information, and never for navigation.

Given the above:

* Your URLs should be standalone and the server should be able to construct the entire page from the URL.
* You should provide links between the related pages on your site and not attempt to constrain users into a particular flow.

For the Yummy Noodle Bar, Users need to:

* View the Menu
* Add and remove items from an order Basket
* Send the Order to the kitchen.
* See the progress of the Order.

### Design your URLs

The following URLs will give that functionality in a way that the user can easily use and return to:

|  |  |
| --- | --- |
| Action | URL |
| Show menu list | GET "/" |
| Add a Menu Item to the current basket and redirect to / | POST "/addToBasket?menuId={menuId}" |
| Remove a Menu Item from Basket and redirect to /showBasket | POST "/removeFromBasket?menuId={menuId}" |
| Show current Basket | GET "/showBasket" |
| Form to gather customer information, which posts to /doCheckout | GET "/checkout" |
| Take the current basket and create an order from it, redirect to "/order/{id}" | POST "/doCheckout" |
| View the status of a given order | GET "/order/{id}" |

Note that every POST URL immediately redirects to another. This allows the user to manually refresh the page at will after the POST has occurred without causing a double submission.

#### URI templates

Each of the above URIs are expressed as templates; they contain blocks demarcated with {} in the URI.

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:

### Model View Controller (MVC)

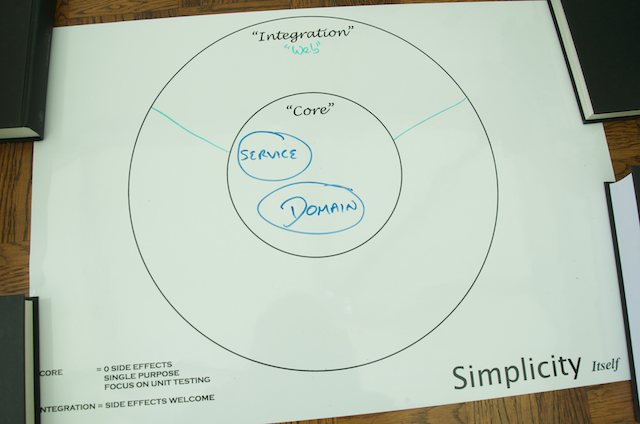
Model View Controller (MVC) is an architecture design that is popularly used in user interface development, whether desktop applications or for the web. It defines three major responsibilities in a UI and specifies how they should interact with each other.

This separation allows an application to be built in a more scalable and testable way.

* **Controller** - Controllers are responsible for accepting user inputs, generating a data Model and then selecting a View to render the Model. In Spring MVC, this is a class annotated with @Controller, with each method annotated with @RequestMapping handling a particular a user input.
* **Model** - The Model is provided to the view, and contains all the information it needs to render itself to show to the user. This is represented in Spring MVC by the Model class. Sometimes this class is not visible, and is generated from other information returned by the Controller method.
* **View** - The View is responsible for presenting information to the user, in the web, the View components will generate HTML and may contain JavaScript. Views in Spring MVC can take many forms, however in this tutorial, all views will be coded using Thymeleaf.

### Summary

Congratulations! You've determined the URLs and links between them that you are going to show to your users and captured those components in the following Life Preserver :



## Step 2: Implementing URLs and returning data

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 interactions that you designed in [Step 1](http://spring.io/guides/tutorials/web/1/).

These interactions can be split into three categories:

* Requests that read, or query, the Menu
* Requests that update the basket
* Requests that create an Order

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.

### Testing at the right level

When writing code, you have a choice of what tests to write, and how much to isolate the code you are testing. The amount of isolation you apply defines the type of test you are writing. There are three major levels, unit, integration and functional.

As spoken about by Mike Cohn and Martin Fowler, these form a [Testing Pyramid](http://martinfowler.com/bliki/TestPyramid.html).

You should write as many tests as is practical at the lower levels of the pyramid, at the unit level, fewer in integration and a minimal amount of tests at the functional level.

### Testing with Spring MockMvc

Spring provides comprehensive support for writing tests at all levels of the pyramid.

You need to write a Spring MVC controller, which will contain a significant number of annotations to define its behaviour. That behaviour needs to be tested so you can be sure that it works along with the raw Java implementation of the controller

Spring provides MockMVC as the solution to this testing, and allows you to write what Martin Fowler calls a [Subcutaneous Test](http://martinfowler.com/bliki/SubcutaneousTest.html), driving the controller in the same way that a full web container would.

Define the behaviour, create a test

The first URL you will implement is "/". This is the root of the Yummy Noodle bar web site, and will be referred to as the 'site' url. It will contain a list of the menu items available, allow users to add the menu items to a basket, and provide a mechanism to place the order

The first thing to do, is make the site url available.

First step, you need to create a new, empty class

com.yummynoodlebar.web.controller.SiteController This is where you will implement the controller.

Before you can implement that controller though, create a new test

com.yummynoodlebar.web.controller.SiteIntegrationTest

Create a controller and mapping

You are building an interactive, HTML website for a user to use, however the first thing you need to do is get a basic controller returning some text to the user.

As we have described in the test above, you will build a controller that will query for menu items. For now, the test expects the menu items to be populated into a plain text file, comma delimited.

Summary

Congratulations! You've created a controller that implements a portion of your Website. You've tested that controller using 'MockMVC' outside of a container to confirm that the handler mappings work.

Your Life Preserver now contains a new component, the SiteController, in the Web domain:

