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

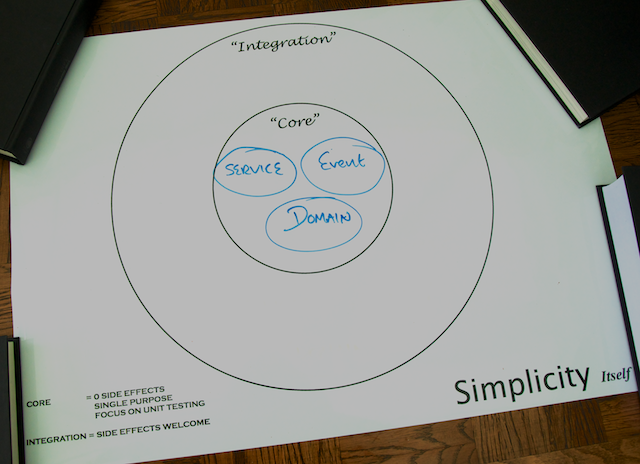
## Spring and the Web

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.

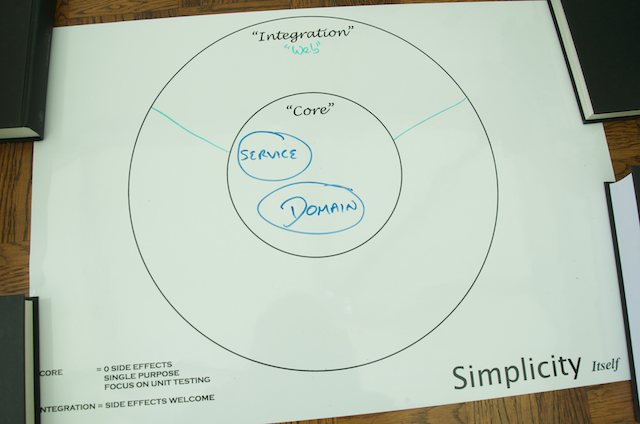
## Yummy Noodle Bar application architecture and the Core domain

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)

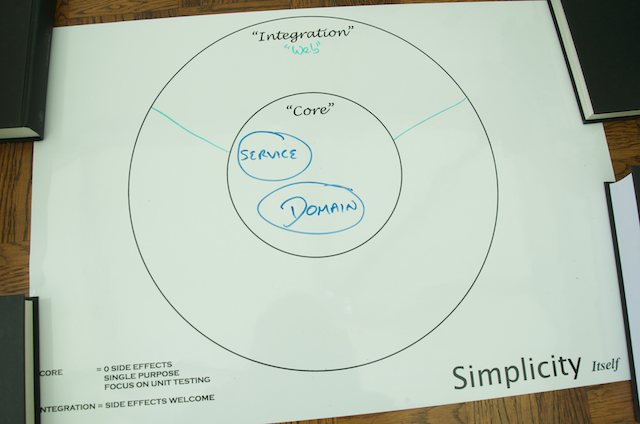
Model View Controller (MVC) is an architecture design that is popularly used in user interface development, whether desktop applications or for the web. It defines three major responsibilities in a UI and specifies how they should interact with each other.

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

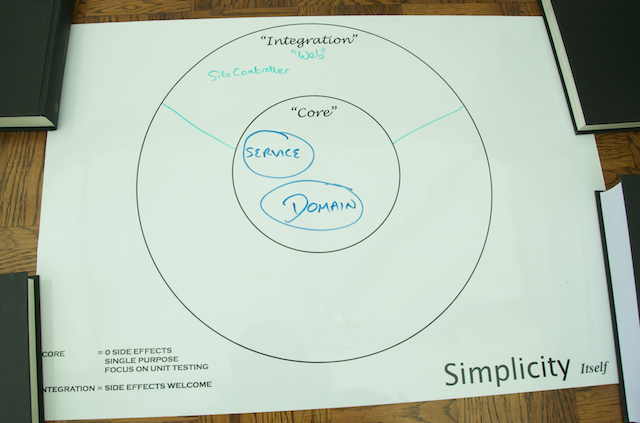
You are building an interactive, HTML website for a user to use, however the first thing you need to do is get a basic controller returning some text to the user.

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:

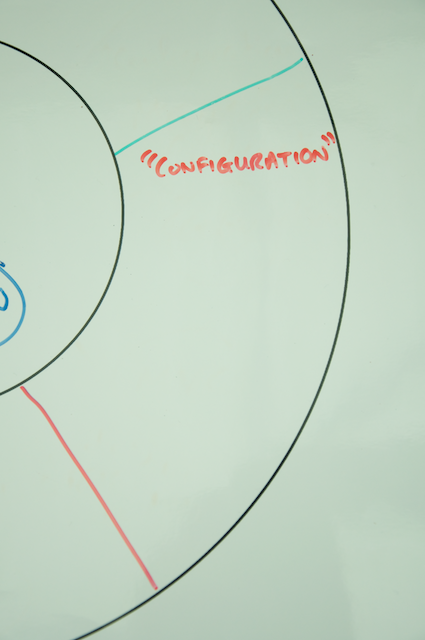


## Step 3: Configuring a basic application

At this point you are ready to:

* Configure the core of your application
* Configure your Web components
* Initialize your infrastructure to create a working WAR file.
* Run your Web application in a web container

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



Create a configuration for your application's Core and Persistence domains using Spring JavaConfig

The Yummy Noodle Bar application contains a core set of components that include domain classes and services. It also contains an in memory persistence stored integrated with the core.

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 and Persistence configurations

### Implement your Core domain configuration

The Core domain configuration for the Yummy Noodle Bar application only contains two services. It relies on the Persistence domain being configured to provide dependencies.

The core event handler will dispatch events to the persistence domain, which will perform the actual persistence. This currently uses a set of HashMaps wrapped by in memory 'repositories'.

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

Create a configuration for your Web components

Configuring your new set of controllers is very straightforward as you have used @Controller on each of the controller classes.

To initialize your Web 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 Web domain configuration

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

### Test your Web domain configuration

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

Initialize your Web 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 single line of XML.

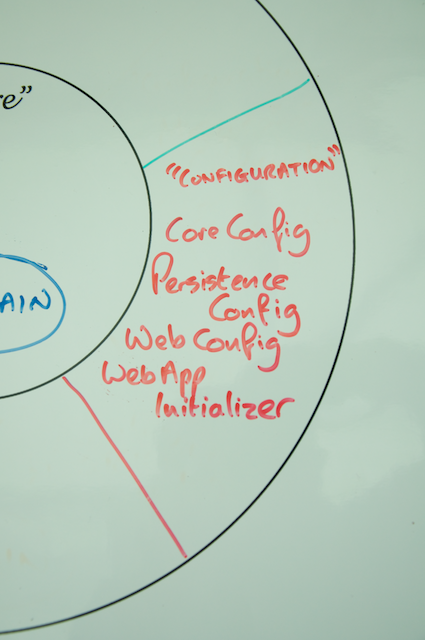
Here you're going to use the WebApplicationInitializer to set up your application's web application context parameters to bootstrap your application's web infrastructure as shown in the following code.

Running your Web service in a Web Container

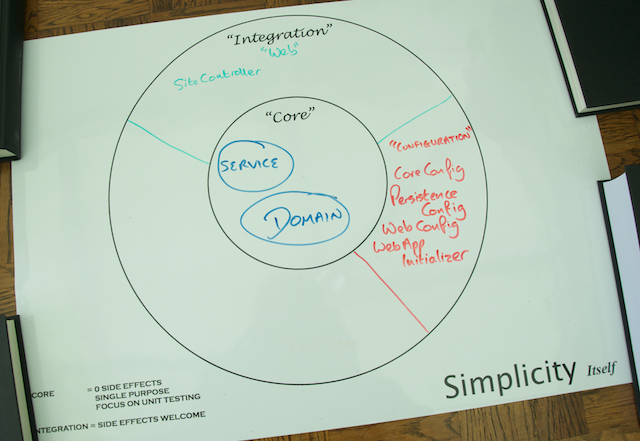
Summary

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

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



Your full Life Preserver should now look like the following:



Your web front end isn't very pretty yet, or even functional. You will add both a pretty face and expand the number of URLs your application responds to, in the next section of this tutorial

## Step 4: Creating rich HTML views using Thymeleaf

Now that you have [configured and started your application](http://spring.io/guides/tutorials/web/2/), which appears in the new Configuration Domain on your life preserver, its time to make the application usable by adding a basket for users to add menu items, and also a view layer to show HTML.

Your application is now ready to:

* Create a basket for the user to keep the items they want in
* Add views to generate HTML.
* Add view fragments to keep common HTML in.

Creating a basket

In the Yummy Noodle Bar website, you are going to add the ability for users to add the food items from the menu that they want to a 'basket'. This will be a list of items, that the user can then choose to convert into an Order. The order process is something you'll do on the next step; for now you need to create a basket.

Up to now, you have created Spring Components (like @Controller) that are shared between all users the system. The default scope [link] of Spring Components is Singleton, so a single instance of the class is shared, everywhere it is used. A basket can't be like this, instead we need an instance of the Basket per user.

While there are several different ways to achieve this, you will use the Scope feature of the Spring Application Context to create a Session scoped bean and inject it into your Controllers as normal.

### Start with a test

As with the other changes you have made, you must start with a test describing the change in behaviour you wish to make.

### Create the basket

Now that you have a test, you can start implementation.

Create a new Basket in the web domain to represent this new concept.

Introducing views

A View is a component that generates HTML that can be sent to the users browser for them to interact with.

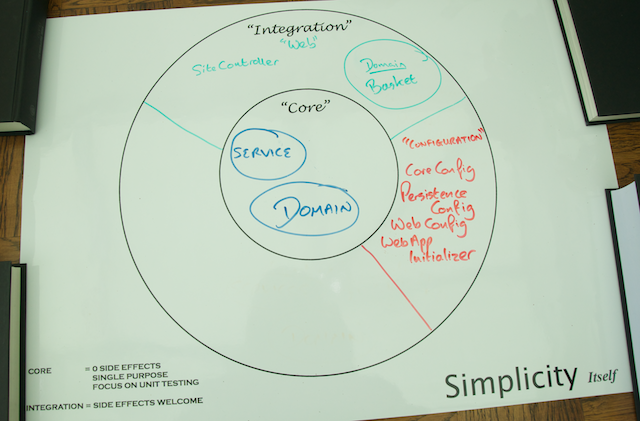
You need to create a new View for the SiteController to render.

You will be using the Thymeleaf templating engine. This is a rich and powerful templating engine that provides all of its functionality as attributes on standard HTML.

Summary

You have extended the application to show the menu in HTML and allow a user to select the items they are interested in and put them into a session backed Basket object, that is only present in the Web Domain.

See the current state of your application in the following Life Preserver



## Step 5: Accepting user submitted data

For Yummy Noodle Bar to accept the orders that a user is making, it needs to know where to send it. Your users will need to give their name and address.

To do this, you must:

* Add a checkout URL - "/checkout"
* Show an HTML form on GET
* Process the form information on POST
* Convert the Basket into an Order and send it to the core.

### Create the Checkout Controller

The Basket you created in the last section will contain all the items that a user wants to order. When they want to place their Order, you need to also collect sufficient information from the customer to deliver their Order.

To do this you will create a new Controller, and have that Controller accept a Command Object.

A command Object is a bean that is used to model an HTTP request. It does this by automatically mapping request parameters onto the properties of the bean. These properties can then be tested using Validation.

Java has a standard Validation API, which you will need to include in the dependencies section of build.gradle.

Since the Validation specification only defines an API, you need to include an implementation of that API as well. Above, you have included Hibernate Validator.

Start with a test

As you should expect, you will first write a test to describe the features you want to implement. Once those tests are ready, you can then safely implement the features themselves.

Storing Customer Information, enter the Command Object

The first thing you need to do is introduce the new concepts you need into the system. The first is the command object.

When you submit a POST request to /checkout, it will contain a set of POST variables in the request that will include information about a user.

You could parse these variables yourself and check their contents according to whatever rules you want to apply. This happens so often, however, that Spring supplies a lot of functionality to ease your implementation.

The Command Object is a class that Spring will map the POST variables onto, parsing them into the given types on the class. For example, if you have an int property on the Command Object, Spring will take the textual value supplied in the request and attempt to parse an int out of it. This process of automatic parsing and conversion is known as Binding, and you can find out more in the [reference documentation](https://docs.springframework.io/spring/docs/3.2.4.RELEASE/spring-framework-reference/html/)

Command Objects are also the ideal place for validation.

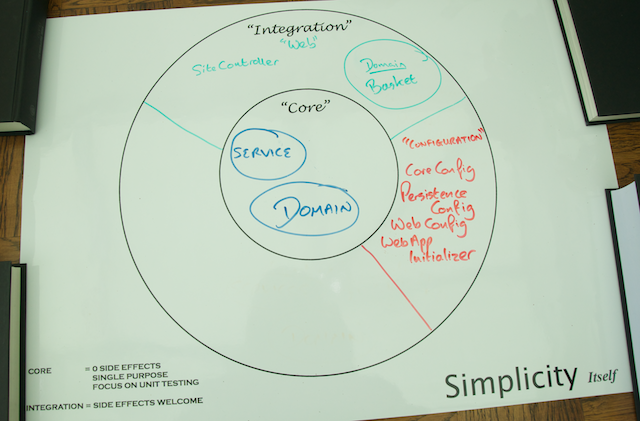
Implement the controller

Now that the Command Object is ready to represent the incoming POST request, you can implement the Controller.

Summary

You have successfully captured some user information in a form, mapped this onto a command object, validated it and combined it with the basket to create a fully functioning Order.

See the current state of your application in the following Life Preserver:



## Step 6: Securing the Web Application

Once again, all changes here are constrained to the Configuration domain:

### Authentication on the Web

To limit access to the Yummy Noodle Bar Web font end, you extend the initial web design as follows:

* URLs that need to have a known user will be protected, and if no user authentication is present, will issue a 302 to the login form.
* A security token will be placed in the HTTP Session in the web container.
* The session will be loaded on every request against the JSESSIONID cookie that is placed in the users browser by a response cookie.

Spring Security helps you perform these steps without you having to change so much as a single controller!

You add the Pivotal milestone repository so that you can use Spring Security 3.2.0.M2. This lets you use some dynamic configuration features of Spring Security, including setting up the web security through JavaConfig.

Now you need to secure your controllers. Until now you've been writing tests first before making any code changes, including configuration. So, instead of immediately adding your security configuration, you'll create a test so that you'll know when your security is being applied correctly.

Configure Spring Security

This configuration enables security using the @EnableWebSecurity annotation, and extends the WebSecurityConfigurerAdapter so that you can perform more detailed configuration of the web security you're applying.

The registerAuthentication method is overridden from WebSecurityConfigurerAdapter in order to configure an in-memory database of users, their passwords and associated roles.

The configure, overridden method from WebSecurityConfigurerAdapter, method provides a fine grained fluent API for controlling how the security system will be applied.

Here, you've configured URL level protection using the http.authorizeUrls() method. The http.authorizeUrls() method protects the /checkout and /order/\* urls, ensuring that only users with the USER role can access them.

This will force users to log in before checking out, and ensure that only logged in users can view orders. The formLogin() method call instructs Spring Security that users will login via an HTML form. We give no further information on how this will work, and so Spring Security will generate a new HTML form and URL for you available on /login.

### Configure the Spring Security filter chain

Spring Security relies on a Servlet filter to apply your security configuration. A filter is used so that security is applied before the Spring MVC Dispatcher Servlet gets involved in processing incoming requests. The Spring Security filter is referred to as the Spring Security filter chain, as it actually delegates to a chain of filters internally that each apply one aspect of the security responsibility.

You now need to configure this filter chain by updating the web application configuration you created earlier. In the previous section, you configured things in the WebAppInitializer class.

The first step is to simply add your new SecurityConfig JavaConfig class to the root context

Next, Spring Security needs to be inserted into the web context setup. This could be done in WebAppInitializer, however a better option in this case is to add a second web app initializer class specifically for the security setup.

## Summary

You have secured your application using Spring Security, and Yummy Noodle Bar is getting excited!

!

See the current state of your application in the following Life Preserver diagram:

