Introduction to the Spring REST Services

What is Spring?

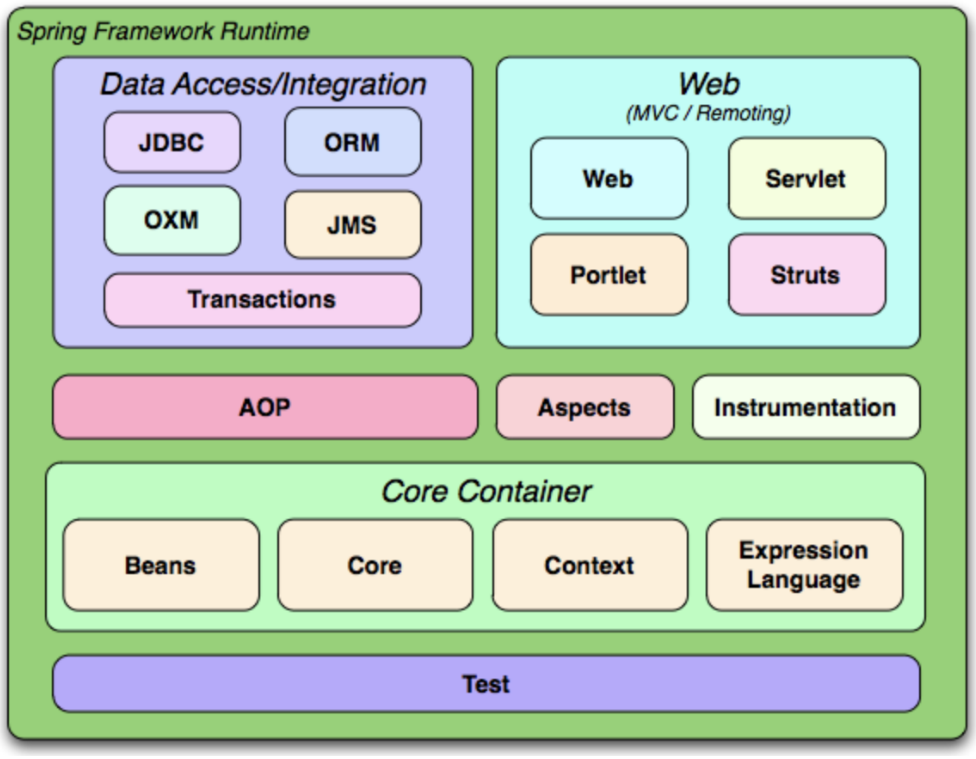
It is hard to choose a backend web framework for an enterprise-grade application. The choice must be many things: mature, stable, well-documented, secure, robust, and maintainable. Fortunately, there exists a good choice on the market: the Spring framework.

Spring is the world's most popular Java framework, providing the underlying infrastructure for Java applications. This makes it an excellent choice for the framework of a Java-based enterprise application that provides REST services.

Spring Modules

The Java platforms have many development functionality but it lacks standardization, especially for an application's basic building blocks. The Spring framework provides these building blocks along with its design patterns and best practices so that projects become formalized.

The Spring Framework consists of about 20 modules, grouped into the diagram as shown below:



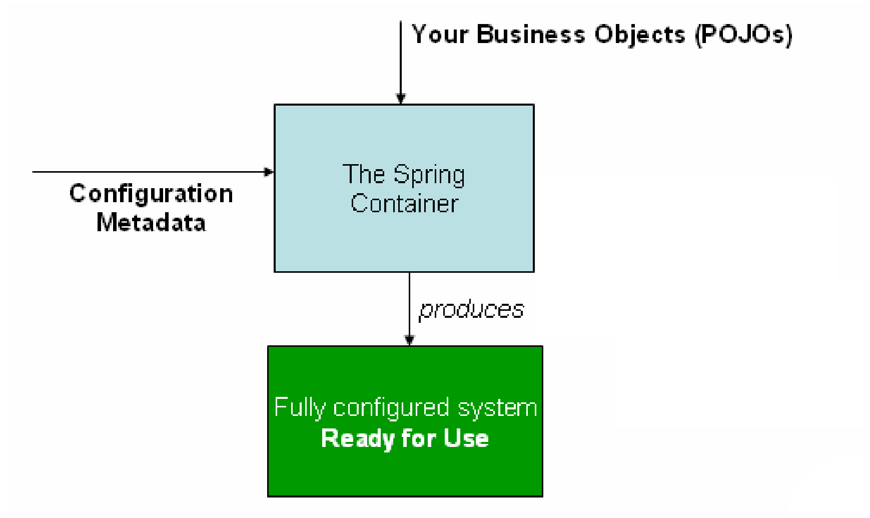
For this article, we will mainly be focusing on the MVC web aspects of the framework as well as the core container.

Spring’s Inversion of Control

IoC is also known as dependency injection (DI). It is a process whereby objects define their dependencies (that is, the other objects they work with) only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method.

In a design without Inversion of Control, one would have to initialize and manage an object's dependencies when the object itself is created. For example, a Car object would have to create an Engine object and manage the Engine.

With Spring’s Inversion of Control, an object can define dependencies without creating them. This object delegates the job of constructing such dependencies to an IoC container. In this way, managing many dependencies is easier and coupling is loosened. One can think of the concept of the inversion of control as letting the IoC container have the control of managing the object’s dependencies.



As the preceding diagram shows, the Spring IoC container consumes a form of configuration metadata that allows the Spring container to instantiate, configure, and assemble the objects in the Spring application. XML-based configuration metadata was traditionally used for defining configuration metadata. However, starting with Spring 3.0, developers can use Java-based configurations instead, which is what this article will focus on.

Constructor-based Dependency Injection

Lets see an example of Beans and dependency injection. Consider a Car and Engine class:

```java
@Component
public class Car {

	private static int counter = 0;

	private String id;
	private Engine engine;

	@Autowired
	public Car(String id, Engine engine) {
		this.id = id;
		this.engine = engine;
	}

	public String getID() {
		return id;
	}

	public Engine getEngine() {
		return engine;
	}

}
```

```java
public class Engine {

	private String type;

	public Engine(String type) {
		this.type = type;
	}

	public String getType() {
		return type;
	}

}
```

```java
@Configuration
@ComponentScan(basePackageClasses = Car.class)
public class CarConfig {

	@Bean
	public Engine engine() {
		return new Engine(1);
	}

}
```

```java
ApplicationContext context = new AnnotationConfigApplicationContext(CarConfig.class);
Car car = context.getBean(Car.class);
```

The above shows an example of constructor-based dependency injection done in the Spring framework.

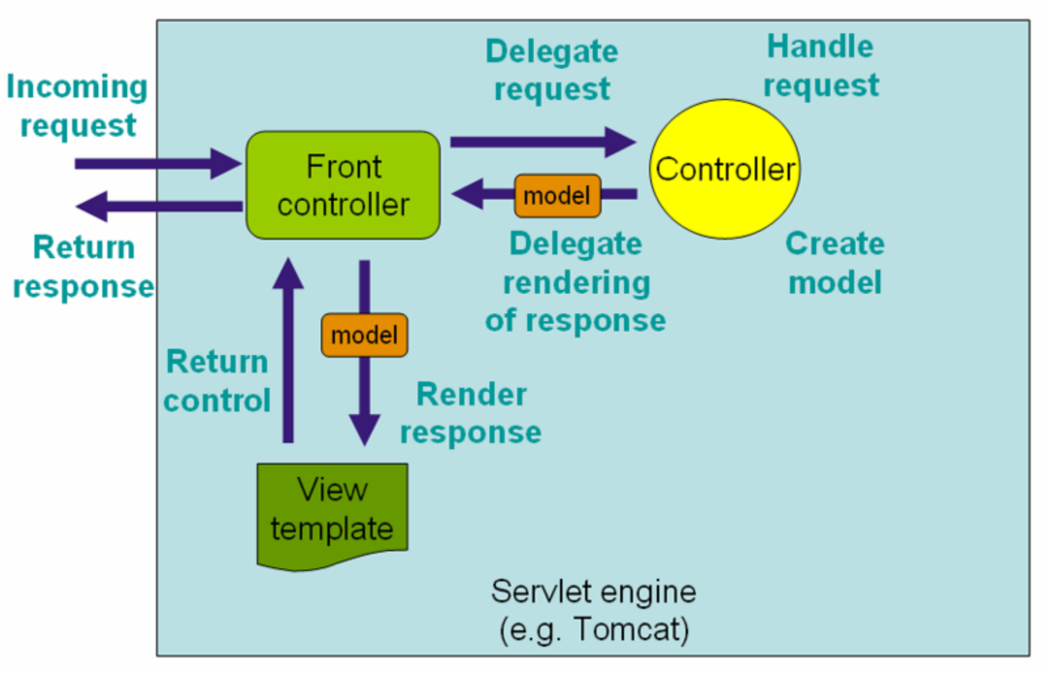
The Car and Engine class are what one would expect in a regular Java setup where a Car object is created by passing in a reference of an Engine object through the constructor. However, we see there are additional annotations in the Car class. The @Component annotation indicates to Spring that this class defines a Spring bean and will automatically detect custom bean dependencies (in this case, the Engine). The @Autowire annotation marks either a constructor, field, setter method, or config method and allows the IoC container to autowire relationships and inject collaborating beans.

The CarConfig class has the @Configuration annotation which indicates the class declares one or more @Bean methods that the Spring container may use to generate bean definitions and service requests at runtime. The @ComponentScan annotation will scan a package for annotated classes that are Spring beans; we will have to scan the package the Car class is located in to locate the class and inject the Engine dependency.

Finally, we create our IoC container by using Spring's built-in ApplicationContext class and pass into it the CarConfig.class as a parameter. We can then create a Car bean using the IoC container which will handle the creation of its dependencies for us.

DispatcherServlet

Spring's web MVC framework is request-driven and has a central servlet that dispatches requests to controllers. Spring relies on its DispatcherServlet to do just this. The servlet is completely integrated with the Spring IoC container and developers can use all of Spring's functionality with it. Each DispatcherServlet has its own WebApplicationContext, which inherits special beans for different web functionality. For this report, we will be focusing on the Controller bean type.



We can define a MVC style Controller using the @Controller annotation like below.

```
@RestController
@RequestMapping("cars")
public class ResponseController {

       private final List<Car> cars;

       public ResponseController() {
              cars = new ArrayList<Car>();
              cars.add(new Car("1", new Engine("v6")));
              cars.add(new Car("2", new Engine("v8")));
              cars.add(new Car("3", new Engine("v12")));
       }

       ...
}
```

URL Mapping

Spring provides many ways to make POST and GET requests, either through a parameter or a path variable. Spring's @PostMapping and @GetMapping annotations within a Controller provides an easy way to map URL endpoints. The @RequestMapping("cars") will map URLs starting with "cars/". As you will be able to see from the examples, Spring is very flexible with POST and GET parameters and responses.

To test our URL Mappings and see it in action, we can use Postman to make RESTful API requests and receive responses.

Example 1:

```
@GetMapping("/all")
public List<Car> getCarAll() {
       return cars;
}
```

The above will obtain the list of cars from the endpoint "/cars/all". Spring will automatically serialize Java objects into JSON format.

```
[
   {"id": "1", "engine": {"type": "v6"},
   {"id": "2", "engine": {"type": "v8"}
   {"id": "3", "engine": {"type": "v12"}
]
```

Example 2:

```
@GetMapping("/car")
public Car getCarRequestParam(@RequestParam(value = "id", defaultValue = "0") String id) {
        return cars.stream().filter(car -> id.equals(car.getID())).findFirst().orElse(null);
}
```

The above will obtain the list of cars from the endpoint “/cars/car” with parameter id. Spring will extract the parameter since we specified the @RequestParam annotation and gave it the appropriate value. For example, requesting the endpoint /cars/car?id=1 will return the Car in the list of cars with id = 1 in JSON format.

```
{
   "id": "1",
   "engine": {
       "type": "v6"
   }
}
```

Example 3

Alternatively, you can call the endpoint “car/car/1” from the following method to obtain the same Car as in example 2. This example uses the @PathVariable annotation for the parameters of the method.

```
@GetMapping("/car/{id}")
public Car getCarPathVar(@PathVariable(value = "id") String id) {
        return cars.stream().filter(car -> id.equals(car.getID())).findFirst().orElse(null);
}
```

Example 4

```
@PostMapping(value="/add")
public String addCar(@RequestBody JsonNode data) {
        String engineType = data.get("engine").get("type").asText();
        cars.add(new Car(Integer.toString(cars.size()+1), new Engine(engineType)));
        return "car added";
}
```

The above will receive a POST request that passes in a JSON object. Spring will automatically serialize the passed data into a JsonNode Java object that we specified in the parameter. We then create the new Car object from the data and add it to the list.

The JSON Object we passed in will be of the following form:

```
{
   "engine" : {
      "type": "v8"
   }
}
```

Example 5

We can also specify HttpServletRequest and HttpServletResponse parameters for our methods. This will allow us to receive and send more detailed requests and responses with additional meta information, such as the IP address of the client. We can also specify the errors in our response and edit Response headers.

```
@PostMapping(value="/servlet")
public ResponseEntity<String> callback(HttpServletRequest request) {
        HttpHeaders responseHeaders = new HttpHeaders();
        responseHeaders.set("header key", "header val");
        try {
                String data = "";
                data += "ip: " + request.getRemoteAddr() + "\n";
                data += "content: " + request.getReader().lines().collect(Collectors.joining());
                return ResponseEntity.ok().headers(responseHeaders).body(data);
        } catch(IOException e) {
                return ResponseEntity.status(HttpServletResponse.SC_INTERNAL_SERVER_ERROR)
                                .headers(responseHeaders)
                                .body("ERROR");
        }
}
```

This concludes a simple overview of the Spring MVC framework.