

Creación de APIs REST con Spring Boot

17 de diciembre de 2014

<http://ryanjbaxter.com/2014/12/17/building-rest-apis-with-spring-boot/>

Sobre la base de mi [anterior entrada del blog](#) en la primavera de arranque, ahora me gustaría hablar de la construcción de las API REST con la primavera de arranque. Durante los últimos 3 años, no he creado una aplicación web que no tenga API REST. REST se ha convertido en el camino a seguir para la construcción de servicios web en las aplicaciones actuales. No voy a gastar tiempo discutiendo lo REST es o por qué es tan popular por lo que si usted no está familiarizado con el descanso y los beneficios de la construcción de las API REST hay un montón de buenos [recursos](#) que hay en discusiones sobre estos temas que sugiero su lectura en primer lugar. En esta entrada de blog me gustaría describir lo fácil que es construir una API REST que persiste los datos en una base de datos Mongo. (No hay nada específico sobre la creación de APIs REST con Mongo DB en Spring Boot. Puede reemplazar fácilmente Mongo con MySQL, PostgreSQL, Couch DB o cualquier otra tecnología DB que le gustaría usar.)

En primer lugar, permite el andamiaje de nuestro proyecto. La cabeza a start.spring.io y rellene el formulario de la siguiente y haga clic en Generar proyecto.

Project metadata		Project dependencies	
Group	<input type="text" value="org.test"/>		
Artifact	<input type="text" value="mongo-demo"/>		
Name	<input type="text" value="Mongo Demo"/>		
Description	<input type="text" value="Demo project for Spring Boot"/>		
Package Name	<input type="text" value="demo"/>		
Type	<input type="text" value="Maven Project"/>		
Packaging	<input type="text" value="Jar"/>		
Java Version	<input type="text" value="1.7"/>		
Language	<input type="text" value="Java"/>		
Spring Boot Version	<input type="text" value="1.2.0"/>		
		Core	Data
		<input type="checkbox"/> Security	<input type="checkbox"/> JDBC
		<input type="checkbox"/> AOP	<input type="checkbox"/> JPA
			<input checked="" type="checkbox"/> MongoDB
			<input type="checkbox"/> Redis
			<input type="checkbox"/> Gemfire
			<input type="checkbox"/> Solr
			<input type="checkbox"/> Elasticsearch
		I/O	Web
		<input type="checkbox"/> Batch	<input checked="" type="checkbox"/> Web
		<input type="checkbox"/> Integration	<input type="checkbox"/> Websocket
		<input type="checkbox"/> JMS	<input type="checkbox"/> WS
		<input type="checkbox"/> AMQP	<input type="checkbox"/> Rest Repositories
			<input type="checkbox"/> Mobile
		Template Engines	Social
		<input type="checkbox"/> Freemarker	<input type="checkbox"/> Facebook
		<input type="checkbox"/> Velocity	<input type="checkbox"/> LinkedIn
		<input type="checkbox"/> Groovy Templates	<input type="checkbox"/> Twitter
		<input type="checkbox"/> Thymeleaf	
			Ops
			<input type="checkbox"/> Actuator
			<input type="checkbox"/> Remote Shell
		<input type="button" value="Generate Project"/>	

Extraiga la cremallera creada por la herramienta e importe el proyecto Maven en su Eclipse IDE. Si miras el archivo POM incluido en el proyecto te darás cuenta de que tenemos dos dependencias, una para el arrancador web y otra para el arrancador Mongo, eso es todo lo que necesitaremos :).

Necesitaremos un objeto de dominio para crear nuestra API, en este ejemplo crearemos una API alrededor de contactos, algo así como una libreta de direcciones. Aquí está el código que utilicé para mi objeto Java de contacto.

```
Import org.springframework.data.annotation.Id;
```

```
@Documento
Public class Contacto {
```

```
@Carné de identidad
```

```

Private String id;
Private String firstName;
Private String lastName;
Privado Dirección de cadena;
Privado String phoneNumber;
Correo electrónico privado de la secuencia;
Cadena privada twitterHandle;
Privado String facebookProfile;
Private String linkedInProfile;
Private String googlePlusProfile;

Public String getId () {
    Return id;
}

Public void setId (String id) {
    This.id = id;
}

Public String getFirstName () {
    Return firstName;
}

Public void setFirstName (String firstName) {
    This.firstName = firstName;
}

Public String getLastName () {
    Devuelve lastName;
}

Public void setLastName (String lastName) {
    This.lastName = lastName;
}

Public String getAddress () {
    dirección del remitente;
}

Public void setAddress (String address) {
    This.address = dirección;
}

Public String getPhoneNumber () {
    Return phoneNumber;
}

Public void setPhoneNumber (String phoneNumber) {
    This.phoneNumber = phoneNumber;
}

Public String getEmail () {
    Devolver correo electrónico;
}

Public void setEmail (String email) {
    This.email = correo electrónico;
}

```

```

    Public String getTwitterHandle () {
        Return twitterHandle;
    }

    Public void setTwitterHandle (String twitterHandle) {
        This.twitterHandle = twitterHandle;
    }

    Public String getFacebookProfile () {
        Return facebookProfile;
    }

    Public void setFacebookProfile (String facebookProfile) {
        This.facebookProfile = facebookProfile;
    }

    Public String getLinkedInProfile () {
        Return linkedInProfile;
    }

    Public void setLinkedInProfile (String linkedInProfile) {
        This.linkedInProfile = linkedInProfile;
    }

    Public String getGooglePlusProfile () {
        Return googlePlusProfile;
    }

    Public void setGooglePlusProfile (String googlePlusProfile) {
        This.googlePlusProfile = googlePlusProfile;
    }
}

```

Como se puede ver esto es más o menos su clase de Java estándar, nada realmente especial. Lo único que no es normal es la anotación `@Id`. Las dependencias de Spring Mongo utilizan esta anotación para vincular este campo al id del objeto creado en la base de datos Mongo.

Ahora que tenemos nuestro objeto de dominio, sería bueno si tuviéramos una clase Java que nos permitiera la interfaz con Mongo DB y proporcionara métodos para manipular los datos dentro de la base de datos. Esto es increíblemente fácil con las dependencias iniciales de Spring Mongo, todo lo que tenemos que hacer es crear una interfaz que extienda `MongoRepository`. Aquí está lo que parece

```

    Import org.springframework.data.mongodb.repository.MongoRepository;
    Public interface ContactRepository extiende MongoRepository <Contacto,
String> {}

```

Eso es ... no realmente eso es todo lo que tenemos que hacer! Usted probablemente está diciendo, "Ryan, esta es una interfaz que al menos necesitamos una aplicación de la derecha?". No, no necesitamos hacer nada, Spring lo cuidará para nosotros. Todo lo que necesitamos hacer es asegurarnos de especificar nuestro objeto de dominio (Contacto) y el tipo de ID de nuestro objeto de dominio (String) en los parámetros de `MongoRepository`. Incluso puede definir métodos de consulta personalizados sin necesidad de implementar nada. El repositorio incluso toma de serialización y deserialización de Java a JSON para nosotros. Cómo funciona todo esto vale la pena otra entrada de blog en sí mismo. Si usted está interesado en algunos de los detalles más finos se puede leer la [documentación](#) .

Finalmente necesitamos definir nuestras API REST. Para ello podemos utilizar la anotación `@RestController` para anotar una clase que abarcará todas nuestras API REST. También usaremos la anotación `@RequestMapping` para definir la ruta URL de nuestra API, así como los parámetros del método y la ruta HTTP para nuestras API. Aquí está el trozo de nuestro controlador REST.

```
import java.util.List;
import java.util.ArrayList;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping ( "/ contactos")
Public class ContactRestController {

    @RequestMapping (method = RequestMethod.GET)
    Public List <Contacto> getAll () {
        Return nuevo ArrayList <Contacto> ();
    }

    @RequestMapping (method = RequestMethod.POST)
    Public Contact create (@RequestBody Contacto de contacto) {
        Return null;
    }

    @RequestMapping (method = RequestMethod.DELETE, value = "{id}")
    Public void delete (@PathVariable String id) {

    }

    @RequestMapping (method = RequestMethod.PUT, value = "{id}")
    Public Contact update (@PathVariable String id, @RequestBody Contacto
de contacto) {
        Return null;
    }

}
```

Como puede ver, utilizamos la anotación `@RequestMapping` tanto en el nivel de clase como en el nivel de método. En el nivel de clase lo usamos para definir la ruta de URL en la que vivirán nuestras API de CRUD, en este caso está en `/ contactos`. También podríamos haber definido esto en la anotación `@RequestMapping` al nivel del método, pero esto nos ahorra un poco de mecanografía repetitiva.

En el nivel de método especificamos el método HTTP que queremos utilizar para cada método Java (GET, POST, PUT, DELETE) y cualquier parámetro de ruta adicional que podamos usar. Algunos ejemplos suelen hacer las cosas más concretas.

Si realiza una solicitud GET a `/ contactos` que hará que se llame al método `getAll`. Mientras que si usted hace una petición DELETE a `/ contactos / 123` que hará que el método `delete` sea llamado y 123 será pasado como el parámetro al método `delete`. La anotación `@PathVariable` le permite asignar parámetros Java a un parámetro path (cualquier cosa rodeada por paréntesis en el parámetro value en `@RequestMapping`).

También usamos la anotación `@RequestBody` para tomar el cuerpo de la solicitud y asignarlo a un parámetro en un método Java. Spring nos ayuda aquí al serializar el JSON en la solicitud HTTP a un objeto Java (en este caso, `Contact`).

Permite llenar esta clase con la implementación de estos métodos

```
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@ RestController
@RequestMapping ( "/" contactos")
Public class ContactRestController {

    autowired
    Privado ContactRepository repo;


    @RequestMapping (method = RequestMethod.GET)
    Public List <Contacto> getAll () {
        Return repo.findAll ();
    }

    @RequestMapping (method = RequestMethod.POST)
    Public Contact create (@RequestBody Contacto de contacto) {
        Return repo.save (contacto);
    }

    @RequestMapping (method = RequestMethod.DELETE, value = "{id}")
    Public void delete (@PathVariable String id) {
        Repo.delete (id);
    }

    @RequestMapping (method = RequestMethod.PUT, value = "{id}")
    Public Contact update (@PathVariable String id, @RequestBody Contacto
de contacto) {
        Contacto update = repo.findOne (id);
        Update.setAddress (contact.getAddress ());
        Update.setEmail (contact.getEmail ());
        Update.setFacebookProfile (contact.getFacebookProfile ());
        Update.setFirstName (contact.getFirstName ());
        Update.setGooglePlusProfile (contact.getGooglePlusProfile ());
        Update.setLastName (contact.getLastName ());
        Update.setLinkedInProfile (contact.getLinkedInProfile ());
        Update.setPhoneNumber (contact.getPhoneNumber ());
        Update.setTwitterHandle (contact.getTwitterHandle ());
        Return repo.save (actualización);
    }
}
```

Ahora debería ser capaz de ver por qué la interfaz MongoRepository que creamos antes hace nuestra vida

más fácil  La única parte de este código que puede tener preguntas es la anotación @Autowired. Esta anotación indica al framework de inyección de dependencia de Spring que inyecta una instancia de ContactRepository en este controlador REST cuando se crea el controlador REST. (Spring Boot creará automáticamente una instancia de nuestro ContactRepository para nosotros ... parte de la magia de configuración automática de Spring Boot!)

Antes de poder probar la aplicación, necesitará un servidor Mongo DB instalado. Esto no es difícil de hacer si desea tener una instalada localmente en su máquina. Puede seguir las Instrucciones en la [página de descarga Mongo DB](#) para el sistema operativo que esté utilizando. La otra opción es utilizar un servicio alojado Mongo DB nube como [Mongo laboratorios](#) . Mongo Labs tiene un plan gratuito que es perfecto para propósitos de desarrollo.

De forma predeterminada Spring asumirá que su servidor Mongo DB se ejecuta en localhost: 27017 y no requiere que se autentique. Si ese no es el caso de lo que necesita para decirle a Spring cómo conectarse a su servidor Mongo DB. Para cambiar el Mongo DB URL Spring utiliza podemos usar un archivo de propiedades. Si creó el proyecto Spring Boot utilizando start.spring.io, debe haber un archivo llamado application.properties en src / main / resources. Si el archivo no está allí, adelante y lo creó. Dentro del archivo, añada la siguiente línea

```
Spring.data.mongodb.uri = mongodb://nombre de usuario: password @ host: port
```

Reemplazando el nombre de usuario y la contraseña con el nombre de usuario y la contraseña a su servidor Mongo DB. Si no necesita un nombre de usuario o contraseña, simplemente elimine esa parte. A continuación, reemplace el host y el puerto con el host y el puerto de su servidor Mongo DB.

Ahora que su servidor Mongo DB está en funcionamiento, puede iniciar su aplicación de inicio de Spring. Para ello se puede abrir el archivo Application.java y ejecutarlo como una aplicación Java desde dentro de Eclipse o si desea utilizar Maven desde la línea de comando de marcha *mvn resorte de arranque: ejecutar* desde el directorio que contiene el archivo de POM. Tomcat debe iniciar en localhost: 8080.

Hay varias opciones para probar tu API, te sugiero que utilices CURL si te gusta la línea de comandos, o un cliente REST si no lo eres. Hay varios buenos clientes de REST por ahí. Me gusta el [plugin de Firefox RESTO cliente](#) . Si usted es un usuario de Mac, [la pata](#) es un cliente nuevo que he estado utilizando últimamente, parece muy prometedor.

Lo primero que podemos probar es hacer una solicitud GET para obtener todos los contactos de la base de datos. Debe devolver un array vacío ya que no hemos añadido ningún contacto todavía. Desde su cliente REST haga una solicitud GET a http: // localhost: 8080 / contacts. Debería obtener una matriz JSON de entry.

No permite agregar un nuevo contacto. Para ello, necesitaremos construir un objeto JSON para un contacto. Aquí hay algunos ejemplos de JSON que puede usar.

```
{
  "FirstName": "Ryan",
  "LastName": "Baxter",
  "Dirección": "1600 Pennsylvania Avenue NW Washington, DC 20500",
  "PhoneNumber": "202-456-1111",
  "Email": "ryan@example.com",
  "TwitterHandle": "@ryanjbaxter",
  "FacebookProfile": "https://www.facebook.com/ryan.baxter.7545",
  "LinkedInProfile": "http://www.linkedin.com/in/ryanjbaxter/",
}
```

```
}    "GooglePlusProfile": "https://plus.google.com/+RyanBaxterJ"
```

Para crear un contacto necesitamos emitir una solicitud POST. Copie el JSON anterior en el campo del cuerpo de la solicitud de su cliente REST y realice un POST a `http://localhost:8080/contacts`. También puede tener que establecer el encabezado `Content-Type` de la solicitud en `application/json` si su cliente REST no lo hace automáticamente. Debe obtener 200 de vuelta después de hacer la solicitud y si mira en el cuerpo de respuesta que debería ver más o menos el mismo JSON que publicó en el servidor, excepto que este objeto JSON tiene un campo de identificación. El ID fue agregado cuando el objeto fue almacenado en el DB Mongo, ese ID es el ID exclusivo para este objeto.

Si vuelves a hacer una solicitud GET a `http://localhost:8080/contacts`, deberías recuperar una matriz JSON que contiene el contacto que acabamos de crear.

Ahora vamos a modificar el objeto de contacto que creamos arriba cambiando el número de teléfono. Una vez más necesitaremos un JSON que contenga los campos modificados, así que aquí hay una muestra

```
{  "FirstName": "Ryan",  "LastName": "Baxter",  "Dirección": "1600 Pennsylvania Avenue NW Washington, DC 20500",  "PhoneNumber": "202-456-2222",  "Email": "ryan@example.com",  "TwitterHandle": "@ryanjbaxter",  "FacebookProfile": "https://www.facebook.com/ryan.baxter.7545",  "LinkedInProfile": "http://www.linkedin.com/in/ryanjbaxter/",  "GooglePlusProfile": "https://plus.google.com/+RyanBaxterJ"
```

Observe la diferencia en el número de teléfono. En su cliente REST, agregue el JSON anterior al cuerpo de la solicitud y realice una solicitud PUT en `http://localhost:8080/contacts/{id}`. Reemplazar `{id}` con el ID que obtuvo cuando realizó la solicitud POST anterior. Si la solicitud tiene éxito, volverá a recuperar un objeto JSON con el número de teléfono actualizado.

Finalmente permite probar el método DELETE haciendo una solicitud DELETE a `http://localhost:8080/contacts/{id}`. Reemplazar `{id}` con la ID del objeto de contacto. Debería obtener una respuesta con un código de estado HTTP de 204, lo que significa que la solicitud se completó correctamente pero no devolvió ningún contenido.

¡Eso es todo lo que hay para crear una API REST con Spring Boot con Mongo DB!

Hay mucho más que hacer y explorar sobre este tema. La interfaz `MongoRepository` es muy potente y le permite crear consultas personalizadas, por lo que sugiero que lea más en la [documentación](#). Lo creas o no, en realidad podemos construir las mismas API REST con menos código. La primavera tiene un proyecto llamado [Primavera de datos REST](#) que nos permitirá eliminar por completo la clase `ContactRestController` escribimos. Se puede ver un ejemplo del uso de este proyecto [aquí](#).

-