 Estilo de arquitectura para desarrollar “SERVICIOS REST”

Integrantes:

Ordoñez Zurita, Adriel

Calle Castillo, Jesús

Arroyo Llacsahuanga, Arturo

Lipa Cueva, Alonso

Asignación:

* Programación II

Docente:

* Coronel Castillo, Erick Gustavo.

Centro de Estudios:

* Universidad de Ciencias y Humanidades



Índice

1. Titulo --------------------------------------------------------------------------- pág. 3

2. Resumen ---------------------------------------------------------------------- pág. 4

3. Antecedentes ----------------------------------------------------------------- pág. 5

4. Desarrollo del Tema

4.1 ¿Qué es REST? ------------------------------------------ pág. 6-9

4.2 Verbos disponibles en REST ------------------------ pág. 9-11

4.2.1 GET (Recuperar)

4.2.2 POST (Crear)

4.2.3 PUT (Actualizar)

4.2.4 DELETE (Borrado)

4.2.5 PATCH (Actualizaciones parciales)

4.2.6 HEAD (Solicitud de cabeceras)

4.3 Creación de una RESTFUL API o API REST---------- pág.11

5. Casos -------------------------------------------------------------------------- pág.12-16

5. Conclusiones ------------------------------------------------------------------ pág. 17

6. Recomendaciones------------------------------------------------------------ pág. 18

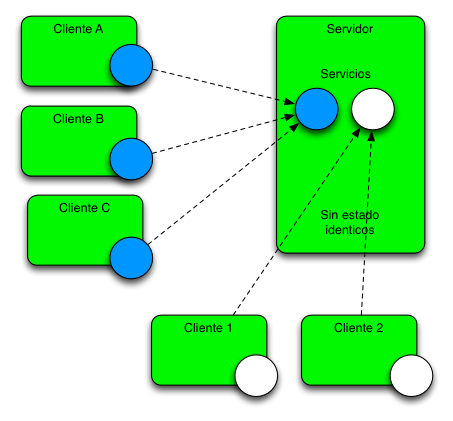
7. Bibliografía --------------------------------------------------------------------------- pág. 19

8. Anexos -------------------------------------------------------------------------------- pág. 20

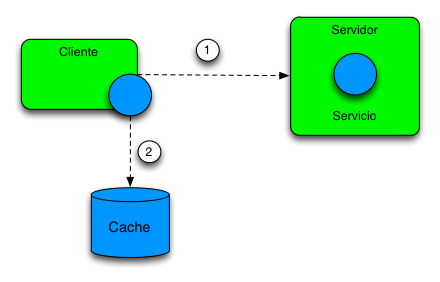
**“Servicios Rest”**



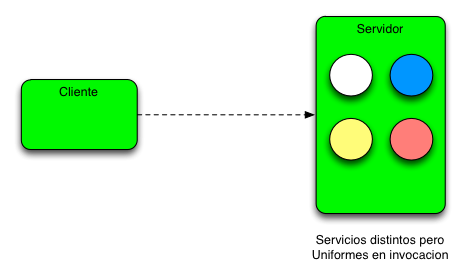
**Servidor/ Cliente**



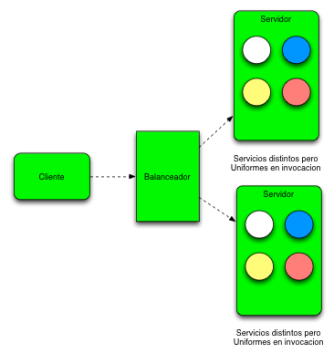
**Sin Estado**

****

**Cache**

****

**Servicios Uniforme**

****

**Arquitectura de Capas**

**Resumen.**

REST al igual que otras tecnologías, como Git y CSS, requiere un poco de tiempo para su comprensión. En el presente trabajo de investigación nos centraremos en la filosofía que está detrás de REST, y en los aspectos prácticos. ¿Cómo podemos implementar REST hoy? Y en una segunda parte hablaremos sobre los Slim framework.

REST (Representational State Transfer) es un estilo de arquitectura para desarrollar servicios. Los servicios web que siguen este estilo deben cumplir con algunas premisas.

**Cliente/Servidor:** Como servicios web son cliente servidor y definen un interface de comunicación entre ambos separando completamente las responsabilidades entre ambas partes.

**Sin estado:** Son servicios web que no mantienen  estado asociado al cliente .Cada petición que se realiza a ellos es completamente independiente de la siguiente. Todas las llamadas al mismo servicio serán idénticas.

**Cache:** El contenido de los servicios web REST se puede cachear de tal forma que una vez realizada la primera petición al servicio el resto puedan apoyarse en la cache si fuera necesario.

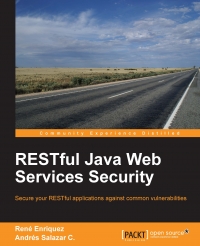
**Servicios Uniformes:** Todos los servicios REST compartirán una forma de invocación y métodos uniforme utilizando los métodos GET,POST,PUT ,DELETE

**Arquitectura en Capas: Todos** los servicios REST están orientados hacia la escalabilidad y un cliente REST no será capaz de distinguir entre sí está realizando una petición directamente al servidor, o se lo está devolviendo un sistema de caches intermedio o por ejemplo existe un balanceador que se encarga de redirigirlo a otro servidor.

**Antecedentes**

Hemos notado que cada día necesitamos más usar servicios web REST porque estos servicios se diferencian de una forma importante de los servicios web SOAP con los que se ha trabajado últimamente y nos pareció magnífico hacer una investigación de este tema y que a la vez el mundo conozca más sobre los grandes beneficios de SERVICIOS REST.

La Transferencia de Estado Representacional (Representational State Transfer) o REST es un estilo de arquitectura software para sistemas [Hipermedia](https://es.wikipedia.org/wiki/Hipermedia) distribuidos como la **[World Wide Web](https://es.wikipedia.org/wiki/World_Wide_Web" \o "World Wide Web)**. El término se originó en el año [2000](https://es.wikipedia.org/wiki/2000), en una tesis doctoral sobre la web escrita por [Roy Fielding](https://es.wikipedia.org/wiki/Roy_Fielding), uno de los principales autores de la especificación del protocolo [HTTP](https://es.wikipedia.org/wiki/HTTP) y ha pasado a ser ampliamente utilizado por la comunidad de desarrollo. Nacido en [1965](https://es.wikipedia.org/wiki/1965) es uno de los autores principales de la especificación [HTTP](https://es.wikipedia.org/wiki/HTTP) y una autoridad citada frecuentemente en la materia de Arquitectura de Redes. Fielding nació en [Laguna Beach](https://es.wikipedia.org/wiki/Laguna_Beach), [California](https://es.wikipedia.org/wiki/California), [Estados Unidos](https://es.wikipedia.org/wiki/Estados_Unidos), y obtuvo el doctorado de la Universidad de California en [2000](https://es.wikipedia.org/wiki/2000).

Si bien el término *REST* se refería originalmente a un conjunto de principios de arquitectura (descritos más abajo), en la actualidad se usa en el sentido más amplio para describir cualquier interfaz entre sistemas que utilice directamente [HTTP](https://es.wikipedia.org/wiki/HTTP) para obtener datos o indicar la ejecución de operaciones sobre los datos, en cualquier formato ([XML](https://es.wikipedia.org/wiki/XML), [JSON](https://es.wikipedia.org/wiki/JSON), etc) sin las abstracciones adicionales de los protocolos basados en patrones de intercambio de mensajes, como por ejemplo [SOAP](https://es.wikipedia.org/wiki/SOAP). Es posible diseñar sistemas de servicios web de acuerdo con el estilo arquitectural REST de Fielding y también es posible diseñar interfaces [XMLHTTP](https://es.wikipedia.org/wiki/XMLHttpRequest) de acuerdo con el estilo de [llamada a procedimiento remoto](https://es.wikipedia.org/wiki/Remote_Procedure_Call) (RPC), pero sin usar SOAP. Estos dos usos diferentes del término *REST* causan cierta confusión en las discusiones técnicas, aunque [RPC](https://es.wikipedia.org/wiki/Remote_Procedure_Call) no es un ejemplo de REST. Los sistemas que siguen los principios REST se llaman con frecuencia *RESTful*

**MARCO TEÓRICO.**

**¿Qué es Rest?**

REST son las siglas de Representacional State Transfer. Fue definido hace una década por Roy Fielding en su tesis doctoral, y proporciona una forma sencilla de interacción entre sistemas, la mayor parte de las veces a través de un navegador web y HTTP. Esta cohesión con HTTP viene también de que Roy es uno de los principales autores de HTTP.

REST es un estilo arquitectónico, un conjunto de convenciones para aplicaciones web y servicios web, que se centra principalmente en la manipulación de recursos a través de especificaciones HTTP. Podemos decir que REST es una interfaz web estándar y simple que nos permite interactuar con servicios web de una manera muy cómoda.

Gracias a REST la web ha disfrutado de escalabilidad como resultado de una serie de diseños fundamentales clave:

Un protocolo cliente/servidor sin estado: cada mensaje HTTP contiene toda la información necesaria para comprender la petición. Como resultado, ni el cliente ni el servidor necesitan recordar ningún estado de las comunicaciones entre mensajes. Sin embargo, en la práctica, muchas aplicaciones basadas en HTTP utilizan cookies y otros mecanismos para mantener el estado de la sesión (algunas de estas prácticas, como la reescritura de URLs, no son permitidas por REST).

Un conjunto de operaciones bien definidas que se aplican a todos los recursos de información: HTTP en sí define un conjunto pequeño de operaciones, las más importantes son POST, GET, PUT y DELETE. Con frecuencia estas operaciones se equiparan a las operaciones CRUD que se requieren para la persistencia de datos, aunque POST no encaja exactamente en este esquema.

Una sintaxis universal para identificar los recursos. En un sistema REST, cada recurso es direccionable únicamente a través de su URI.

Vamos a ver primero lo que son las URIs. Una URI es esencialmente un identificador de un recurso. Veamos el siguiente ejemplo:

GET /amigos

Podríamos llamar a esto un recurso. Cuando esta ruta es llamada, siguiendo los patrones REST, se obtendrán todos los amigos (generalmente de una base de datos), y se mostrarán en pantalla o se devolverán en un formato determinado a quien lo solicite.

Pero, ¿cómo haremos si queremos especificar un amigo en particular?

GET /amigos/marta

Como se puede ver es fácilmente comprensible. Esa es una de las claves de la arquitectura RESTful. Permite el uso de URIs que son fácilmente comprensibles por los humanos y las máquinas.

Piensa en un recurso como un nombre en plural. Contactos, estados, usuarios, fotos --- todos éstos serían elecciones perfectas.

Hasta ahora, hemos visto cómo identificar a una colección y a elementos individuales en esa colección:

GET /amigos

GET /amigos/marta

De hecho, encontrarás que estos dos segmentos son todo lo que tendrías que haber necesitado siempre. Pero podemos profundizar un poco más en la potencia de HTTP para indicar cómo queremos que el servidor responda a esas peticiones. Veamos:

Cada petición HTTP especifica un método, o un verbo, en sus encabezados. Generalmente te sonarán un par de ellos como GET y POST.

Por defecto el verbo utilizado cuando accedemos o vemos una página web es GET.

Para cualquier URL dada, podemos referenciar hasta 4 tipos diferentes de métodos: GET, POST, PUT, PATCH y DELETE.

GET /amigos

POST /amigos

PUT /amigos

DELETE /amigos

Esencialmente, estos verbos HTTP indican al servidor que hacer con los datos especificados en la URI. Una forma fácil de asociar estos verbos a las acciones realizadas, es comparándolo con CRUD (Create-Read-Update-Delete).

GET => READ

POST => CREATE

PUT => UPDATE

DELETE => DELETE

Anteriormente hemos dicho que GET es el método utilizado por defecto, pero también te debería sonar POST. Cuando enviamos datos desde un formulario al servidor, solemos utilizar el método POST. Por ejemplo si quisiéramos añadir nuevos Tweets a nuestra base de datos, el formulario debería hacer un POST de los tweets POST /tweets, en lugar de hacer /tweets/añadirNuevoTweet.php.

Ejemplos de URIs que son no RESTful y que no se recomienda utilizar:

/tweets/añadirNuevoTweet.php

/amigos/borrarAmigoPorNombre.php

/contactos/actualizarContacto.php

Ejemplos de URIs que son RESTful y que serían un buen ejemplo:

GET /tickets/12/messages - Devuelve una lista de mensajes para el ticket #12

GET /tickets/12/messages/5 - Devuelve el mensaje #5 para el ticket #12

POST /tickets/12/messages - Crea un nuevo mensaje en el ticket #12

PUT /tickets/12/messages/5 – Actualiza el mensaje #5 para el ticket #12

PATCH /tickets/12/messages/5 - Actualiza parcialmente el mensaje #5 para el ticket #12

DELETE /tickets/12/messages/5 - Borra el mensaje #5 para el ticket #12

¿Pero entonces, cuáles serían las URIs correctas para presentar un formulario al usuario, con el objetivo de añadir o editar un recurso?

En situaciones como esta, tiene más sentido añadir URIs como:

GET /amigos/nuevo

GET /amigos/marta/editar

La primera parte de la trayectoria /amigos/nuevo, debería presentar un formulario al usuario para añadir un amigo nuevo. Inmediatamente después de enviar el formulario, debería usarse una solicitud POST, ya que estamos añadiendo un nuevo amigo.

Para el segundo caso /amigos/marta/editar, este formulario debería editar un usuario existente en la base de datos. Cuando actualizamos los datos de un recurso, se debería utilizar una solicitud PUT.

**Verbos disponibles en REST**

Antes de seguir adelante con ejemplos concretos, vamos a revisar un poco más los verbos utilizados en las peticiones a una API REST.

**GET (Recuperar)**

GET es el método HTTP utilizado por defecto en las peticiones web. Una advertencia a tener en cuenta es que deberíamos utilizar GET, para hacer peticiones sólo de lectura, y deberíamos obtener siempre el mismo tipo de resultado, independientemente de las veces que sea llamado ese método.

Como programador puedes hacer lo que quieras cuando se hace una llamada a las rutas en la URI, pero una buena práctica es seguir las reglas generales para diseñar una API REST correctamente.

**POST (Crear)**

El segundo método que te resultará familiar es POST. Se utilizará para indicar que vamos a crear un subconjunto del recurso especificado, o también si estamos actualizando uno o más subconjuntos del recurso especificado.

Por ejemplo vamos a crear un recurso nuevo por ejemplo enviando un nuevo usuario para darlo de alta, entonces lo haremos a la URL /amigos

POST /amigos

**PUT (Actualiza)**

Utiliza PUT cuando quieras actualizar un recurso específico a través de su localizador en la URL.

Por ejemplo si un artículo está en la URL http://miweb.local/api/articulos/1333, podemos actualizar ese recurso enviando todos los campos a actualizar desde un formulario (método POST):

PUT /api/articulos/1333

Si no conocemos la dirección del recurso actual, por ejemplo para añadir un nuevo artículo, entonces utilizaríamos la acción POST. Por ejemplo.

POST /api/artículos

**DELETE (Borrado)**

Por último DELETE debería ser usado cuando queremos borrar el recurso especificado en la URI. Por ejemplo si ya no somos más amigos de macarena, siguiendo los principios de REST, podríamos borrarla usando una petición delete a la URI:

DELETE /amigos/macarena

**PATCH (Actualizaciones parciales)**

Una solicitud de tipo PATCH se utiliza para realizar una actualización parcial de un recurso es decir para actualizar ciertos campos del recurso y no el recurso al completo. Los campos a actualizar se enviarían desde un formulario por POST y el tipo de petición es PATCH.

PATCH /api/articulos/1333

**HEAD (Solicitud de cabeceras)**

Una solicitud de tipo HEAD es como una solicitud GET, con la salvedad que solamente se devuelven las cabeceras HTTP y el código de respuesta, no el documento en sí mismo.

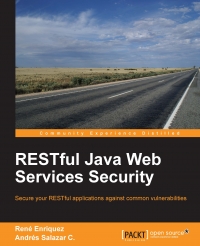
Con este método el navegador puede comprobar si un documento ha sido modificado, por temas de caché por ejemplo. También puede comprobar si un documento existe o no.

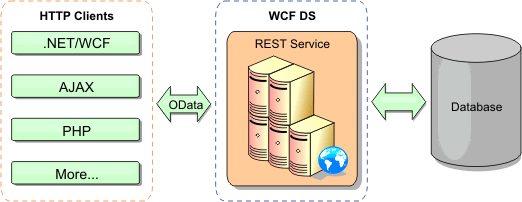
Por ejemplo, si tienes un montón de enlaces en tu web, periódicamente podrías comprobar mediante peticiones HEAD si los hiperenlaces son correctos o están rotos. Éste tipo de comprobación es muchísimo más rápido que usar GET.

**Creación de una RESTFUL API o API REST**

Para crear una API REST o RESTFUL API (en inglés) en php podremos hacerlo utilizando un fichero .htaccess dónde programamos todos los tipos de URI's que gestionaremos en la API o bien utilizando un framework que nos facilite dicha programación.

Veremos en la siguiente parte del “Trabajo de Investigación” acerca del Slim micro Framework, que es bastante sencillo y nos facilita muchísimo este tipo de creación, con lo que podremos crear una API REST para cualquier aplicación que ya tengamos programada de forma muy sencilla.

****

****

**CASOS.**

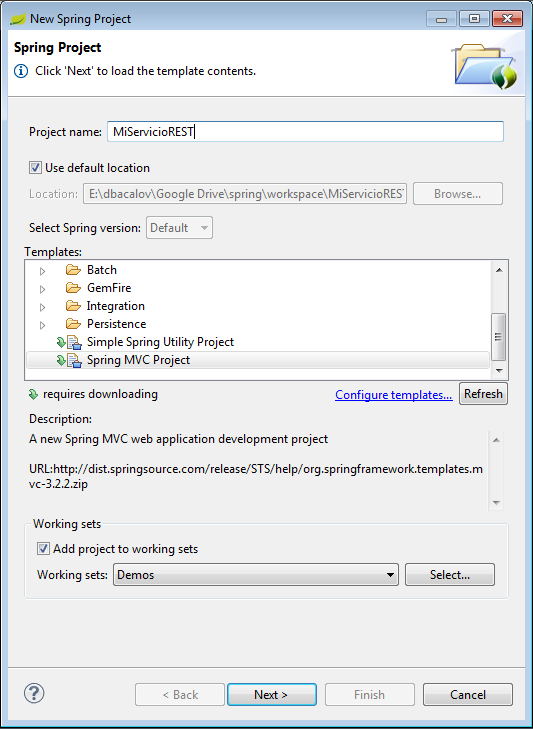
Les mostraremos detalladamente lo sencillo que es crear un Servicio Web REST que responda en formato JSON con SpringSource Tool Suite (STS).

En la “Arquitectura de aplicaciones corporativas para nube” que propone VMware, la Capa de Negocio ofrece un Servicio REST que responde en formato JSON listo para ser utilizado por el Javascript de aplicación de Front End.

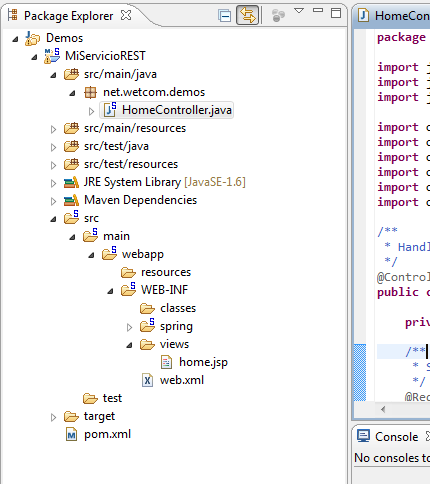
Veamos que tanto esfuerzo supone para crear nuestro primer Servicio REST JSON.

**Paso 1: crear el nuevo proyecto**

Abrimos SpringSource Tool Suite, seleccionamos New -> Spring Project -> Spring MVC Project y simplemente completamos el nombre del proyecto. Si es la primera vez que lo usamos, STS bajará el template automáticamente de Internet.

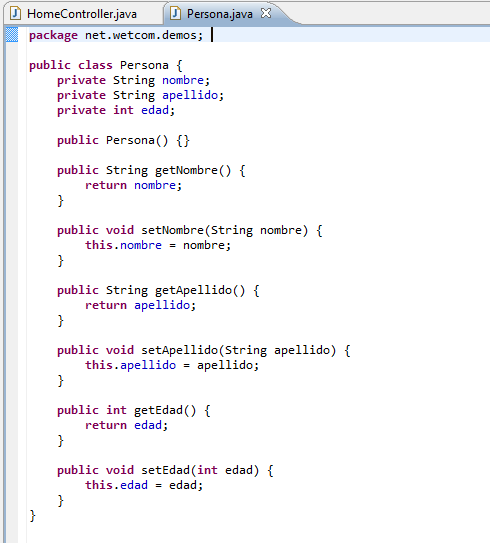


Una vez finalizado el proceso veremos el conocido proyecto MVC con el “HomeController.java”, la vista “home.jsp” y el pom.xml.



**Paso 2: construir la entidad que el servicio le entregará a la capa de presentación**

Para esta demo construimos una entidad “Persona” con tres campos (nombre, apellido y edad) y sus correspondientes “getters” y “setters”. De hecho, estos últimos no son realmente necesarios.

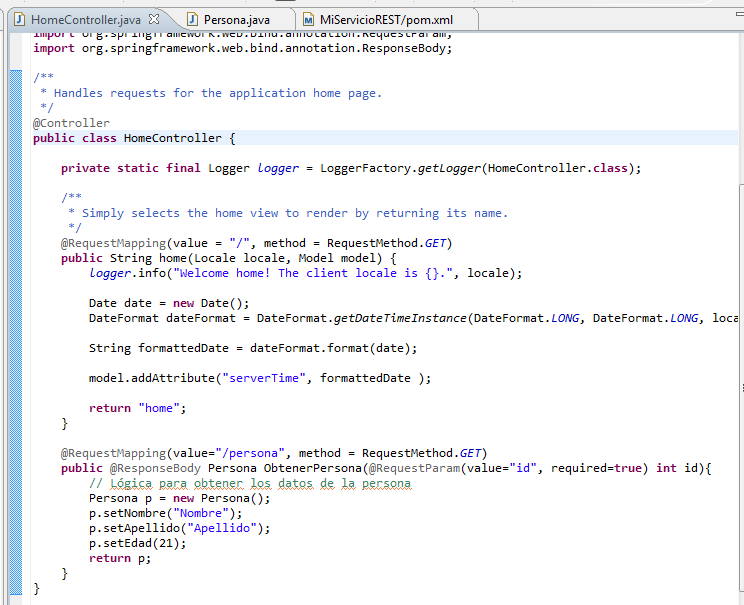


**Paso 3: modificando el “controller”**

Creamos un nuevo método en “HomeController.java” que responde a la URL “/persona” y que requiere un parámetro llamado “id”. Este parámetro podría ser opcional, pero lo hicimos así para mantener todo lo más simple posible.

Lo importante es que, en lugar de devolver un “String” con el nombre de la vista (home.jsp), este nuevo método devuelve directamente un objeto del tipo “Persona”. La etiqueta @ResponseBody le dice a Spring Framework que convierta el objeto en una respuesta http.

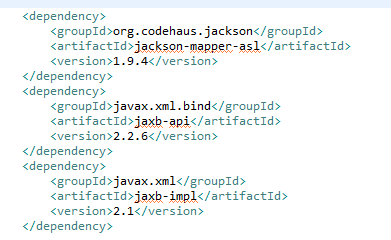
Por supuesto que debería haber una lógica para buscar los datos de la persona basándonos en el “id”, pero por ahora devolveremos siempre el mismo objeto.

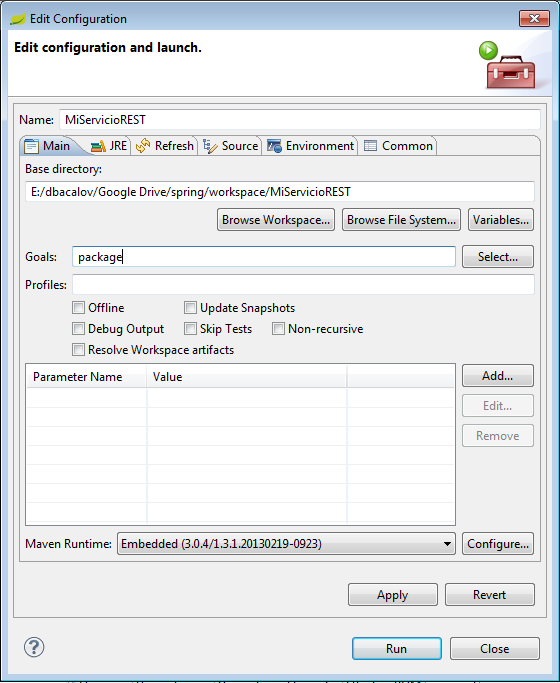
****

**Paso 4: agregando dependencias**

Para convertir nuestro objeto en una respuesta http válida, Spring Framework busca una librería que sepa cómo hacerlo. Si quisiéramos ejecutar nuestro proyecto en este momento nos daría un error diciendo que no encuentra ninguna que haga el trabajo.

Para solucionar esto, en el archivo pom.xml agregamos las siguientes dependencias (Podrían ser otras que hagan lo mismo).

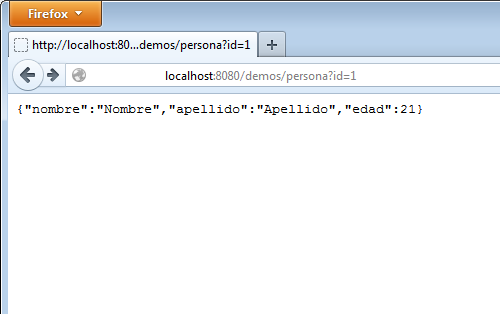


Luego, hacemos clic derecho y seleccionamos Run as -> Maven build y en “Goals” escribimos package. De esta forma descargaremos las nuevas librerías.

**Paso 5: ejecutar**

Finalmente hacemos clic derecho sobre el proyecto, Run as -> Run on server, seleccionamos el tcServer que instaló SpringSource Tool Suite y aparecerá la página inicial de nuestro proyecto web MVC.

Para ver nuestro trabajo, modificamos la URL y agregamos “/persona?id=1”. Veremos que la respuesta es JSON con los datos de la entidad Persona lista para ser utilizado por Javascript.



**CONCLUSIONES.**

-En conclusion REST no es una tecnología, ni siquiera una arquitectura, sino una familia de arquitecturas, ya que está basado en los siguientes estándares:

• HTTP

• URL

• Representación de los recursos: XML/HTML/GIF/JPEG/…

• Tipos MIME: text/xml, text/html.

-REST describe un estilo arquitectónico de sistemas en red como, por ejemplo, aplicaciones Web. Las limitaciones REST, aplicadas como un todo, generan una arquitectura simple, escalable, eficiente, segura, confiable y extensible.

**RECOMENDACIONES.**

En esta investigación hemos necesitados de información de varias páginas web confiables, las cuales nos han servido mucho para que logremos terminar esta investigación y de paso esto será muy beneficio para todos sus lectores, ya que podrán estar al tanto de los inicios y objetivos de Servicios Rest. Por otro lado el lector tendrá que manejar términos o palabras del campo de la Ingeniería sino tendrá un poco de problemas para entender lo que se ha redactado.

Más adelante profundizaremos más en este tema interesante que es Servicios Rest, llegaremos a ustedes con más casos cada vez un poco más complicados así que tendrá que estar al tanto de esta investigación primeriza porque si no será un poco complicado entender lo que se viene después.

**BIBLIOGRAFÍA.**

1. Arquitectura, J. (2015). Servicio Rest. Retrieved from http://www.arquitecturajava.com/servicios-rest/
2. Contreras, R. (2016). Intruccion Rest. Retrieved from http://vis.usal.es/rodrigo/documentos/soa/REST introduccion.pdf
3. Dosideas. (2013). Introduccion Servicios Rest. Retrieved from http://www.dosideas.com/noticias/java/314-introduccion-a-los-servicios-web-restful.html
4. Esteven, P. (2014). Introduccion y Bases teoricas Servicio Rest. Retrieved from http://www.adwe.es/general/colaboraciones/servicios-web-restful-con-http-parte-i-introduccion-y-bases-teoricas
5. Github. (2014). Cliente de un Servicio Rest. Retrieved from https://picodotdev.github.io/blog-bitix/2016/10/cliente-de-un-servicio-rest-autenticado-con-oauth-en-java/
6. Reeve, J. (2014). Servicio Rest Autenticado con Oauth. Retrieved from http://es.makemefeed.com/2016/10/02/blog-bitix-cliente-de-un-servicio-rest-autenticado-con-oauth-en-java-3840873.html
7. Rene, P. (2014). Desarrollo Servicio Rest. Retrieved from https://www.workana.com/es/job/desarrollo-de-servicios-rest-en-php-para-control-de-actividades