

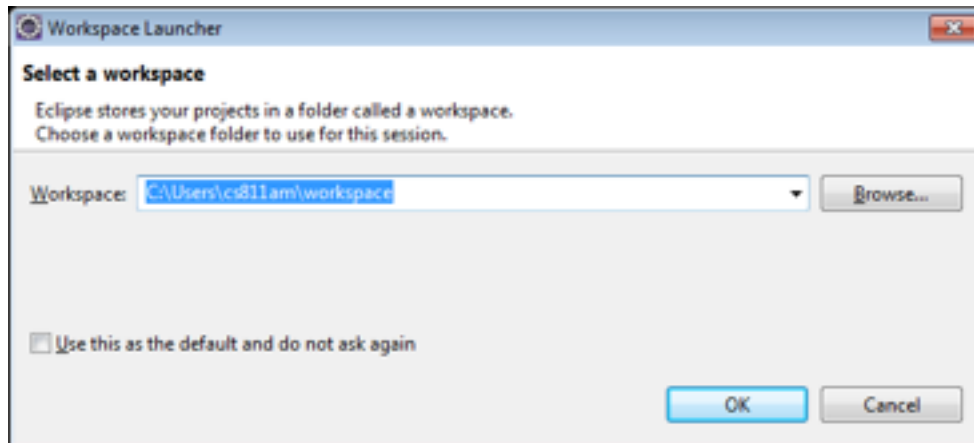
## Creating a Web Service in JAVA

### 1. Create a Maven Project

#### 1. Load Eclipse

Find where you installed Eclipse and run eclipse.exe.

When loading it will ask you for a workspace location. Click OK.

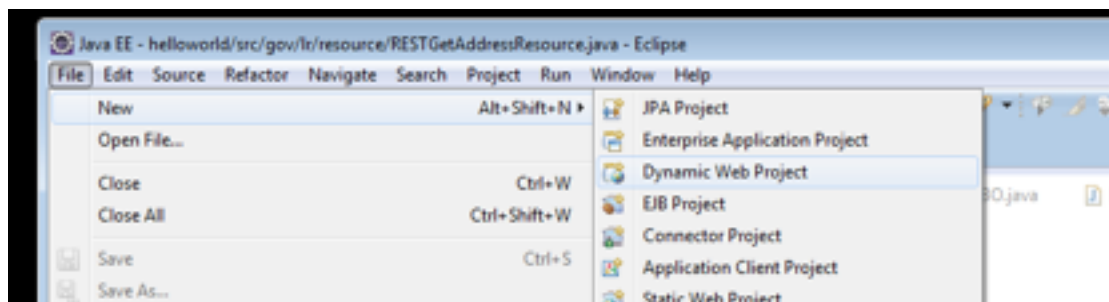


#### What is a workspace?

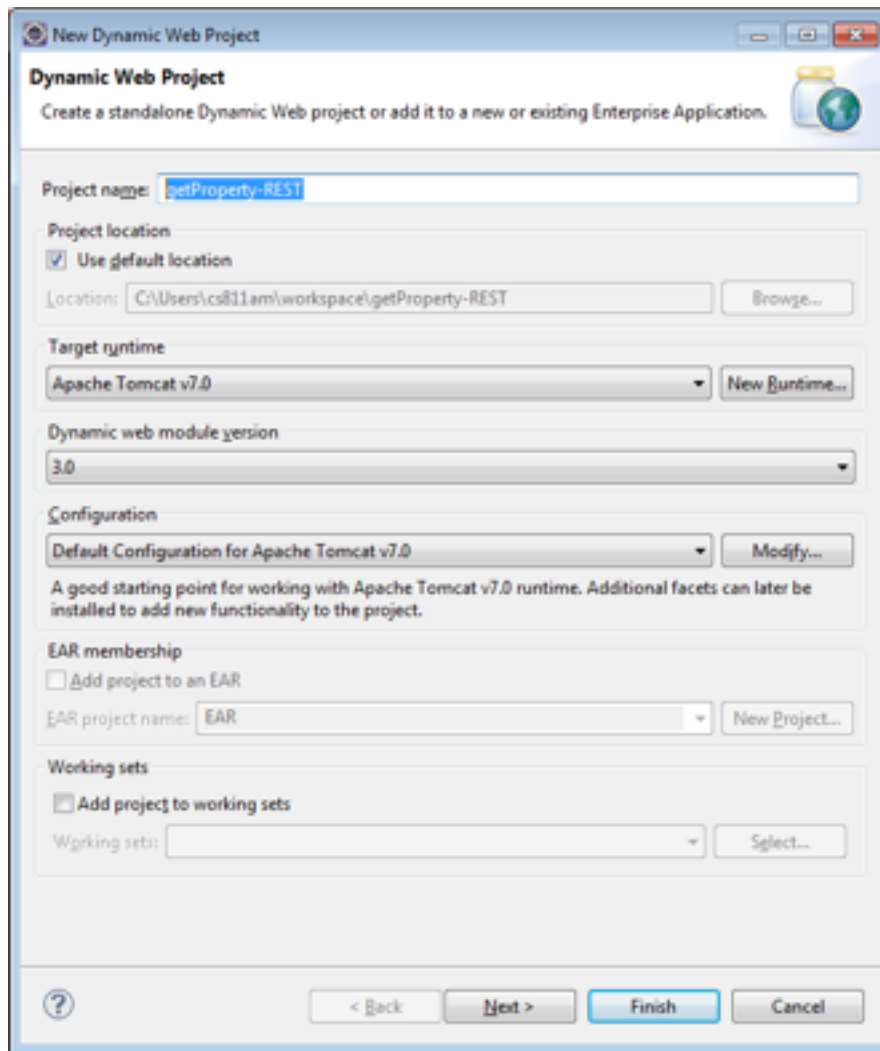
A workspace is a logical collection of projects. A workspace is a directory on your hard drive where Eclipse stores the projects that you define to it.

### 2. Create a Dynamic Web Project

Go to **File -> New** and then click **Dynamic Web Project**.

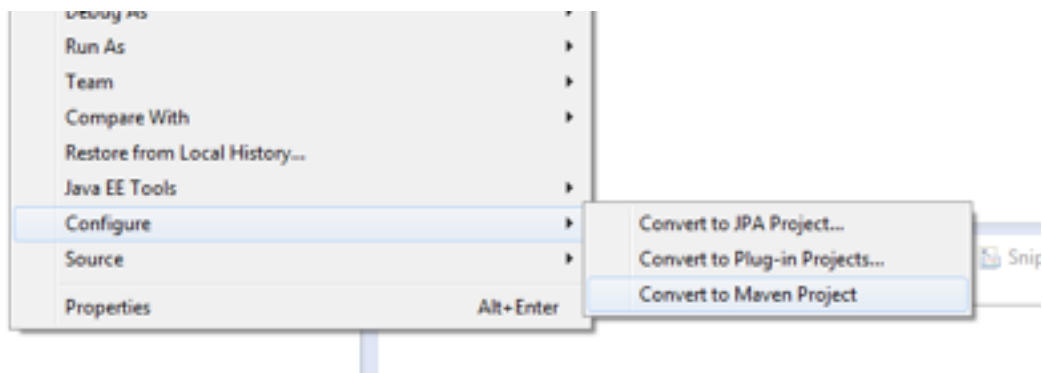


Put in a project name of **getProperty-REST** and click **Finish**.

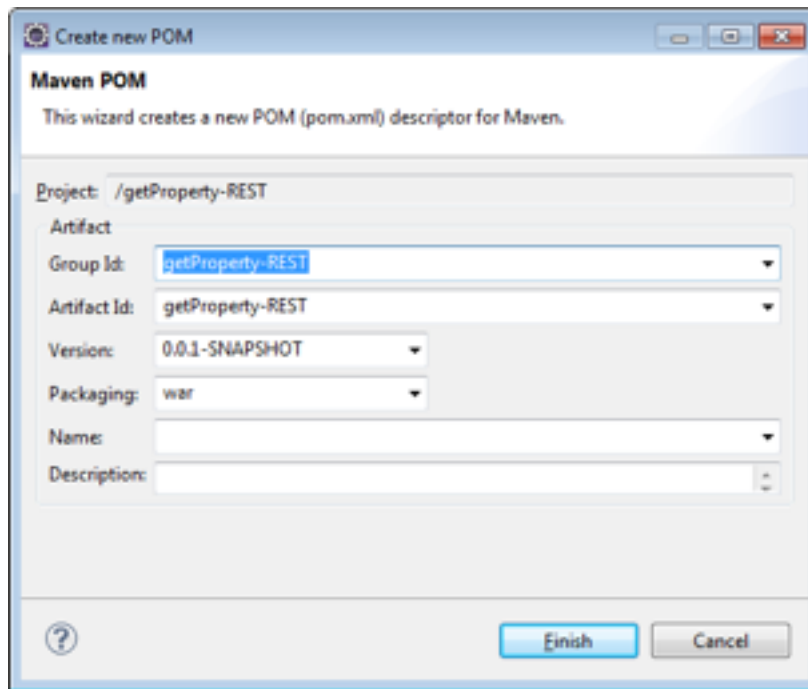


### 3. Convert the project to Maven

Right click on **getProperty-REST** and select **Configure**, and then within that sub menu, select **Convert to Maven Project**.



On the **Create new POM** screen, click **Finish**.



### What is a Maven?

Maven's primary goal is to allow a developer to comprehend the complete state of a development effort in the shortest period of time. In order to attain this goal there are several areas of concern that Maven attempts to deal with:

- Making the build process easy
- Providing a uniform build system
- Providing quality project information
- Providing guidelines for best practices development
- Allowing transparent migration to new features

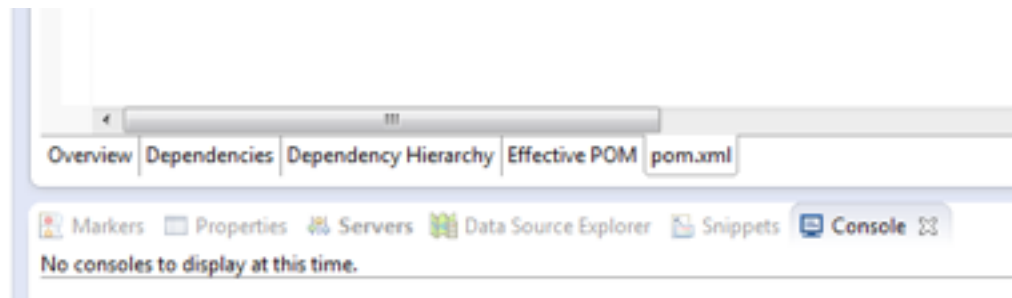
### What is a POM?

POM stands for "Project Object Model". It is an XML representation of a Maven project held in a file named pom.xml. It is a one-stop-shop for all things concerning the project.

## 2. Adding Dependencies to the POM

### 2.1 Modify the pom.xml

Click on **pom.xml** tab



Amend your pom.xml so it looks like this (the bit in red is new):

```

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/
2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>getProperty-REST</groupId>
  <artifactId>getProperty-REST</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>war</packaging>
  <build>
    <sourceDirectory>src</sourceDirectory>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.1</version>
        <configuration>
          <source>1.7</source>
          <target>1.7</target>
        </configuration>
      </plugin>
      <plugin>
        <artifactId>maven-war-plugin</artifactId>
        <version>2.3</version>
        <configuration>
          <warSourceDirectory>WebContent</warSourceDirectory>
          <failOnMissingWebXml>>false</failOnMissingWebXml>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <dependencies>
    <dependency>
      <groupId>javax.ws.rs</groupId>
      <artifactId>javax.ws.rs-api</artifactId>
      <version>${jaxrs.api.version}</version>
    </dependency>
    <dependency>
      <groupId>javax.annotation</groupId>
      <artifactId>javax.annotation-api</artifactId>
      <version>${javax.annotation.version}</version>
    </dependency>
    <dependency>
      <groupId>org.glassfish.jersey.containers</groupId>
      <artifactId>jersey-container-servlet</artifactId>
      <version>${jersey2-version}</version>
    </dependency>
    <dependency>
      <groupId>org.glassfish.jersey.core</groupId>
      <artifactId>jersey-client</artifactId>
      <version>${jersey2-version}</version>
    </dependency>
    <dependency>
      <groupId>org.glassfish.jersey.media</groupId>
      <artifactId>jersey-media-multipart</artifactId>
      <version>${jersey2-version}</version>
    </dependency>
    <dependency>
      <groupId>org.glassfish.jersey.media</groupId>
      <artifactId>jersey-media-moxy</artifactId>
      <version>${jersey2-version}</version>
    </dependency>
    <dependency>
      <groupId>org.glassfish.jersey.media</groupId>
      <artifactId>jersey-media-json-jackson</artifactId>

```

Rather than typing this out, It might be best to copy this from the POM in SVN.

### What have I just done?

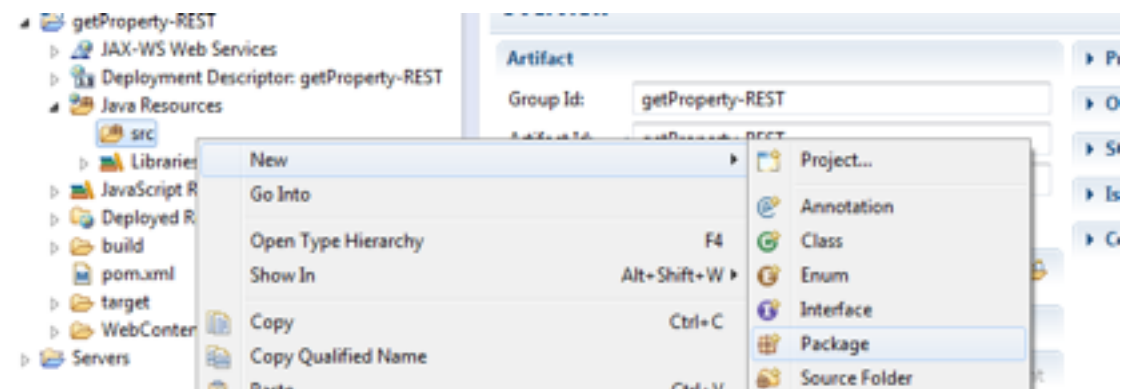
You have just modified the POM.xml file to add dependency libraries which will be required for to build a web service (and also connect to mongodb). These files will be downloaded and compiled with your java web service when you go to build it.

## 3. Creating a Web Service

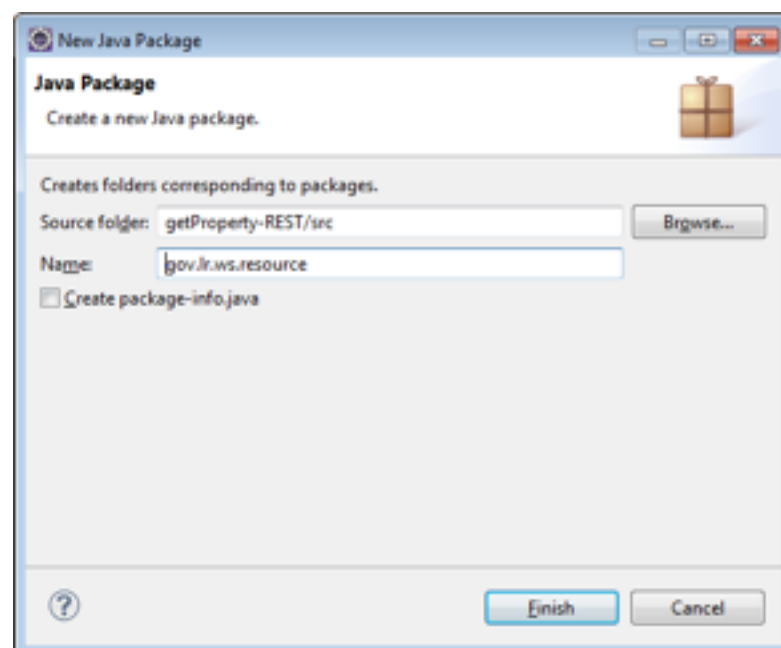
### 3.1 Creating a Resource Class

Expand out **Java Resources**.

Right click on **src** and select **New** and then **Package**.



Give the package a name of **gov.lr.ws.resource** and click **Finish**



### What is a Package?

A package is a namespace that organizes a set of related classes and interfaces. Conceptually you can think of packages as being similar to different folders on your computer. Because software written in the Java programming language can be composed of hundreds or *thousands* of individual classes, it makes sense to keep things organized by placing related classes and interfaces into packages.

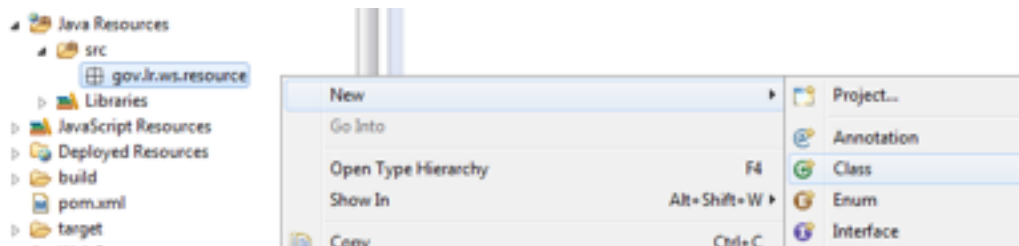
### Why gov.lr.ws.resource?

Package naming convention:

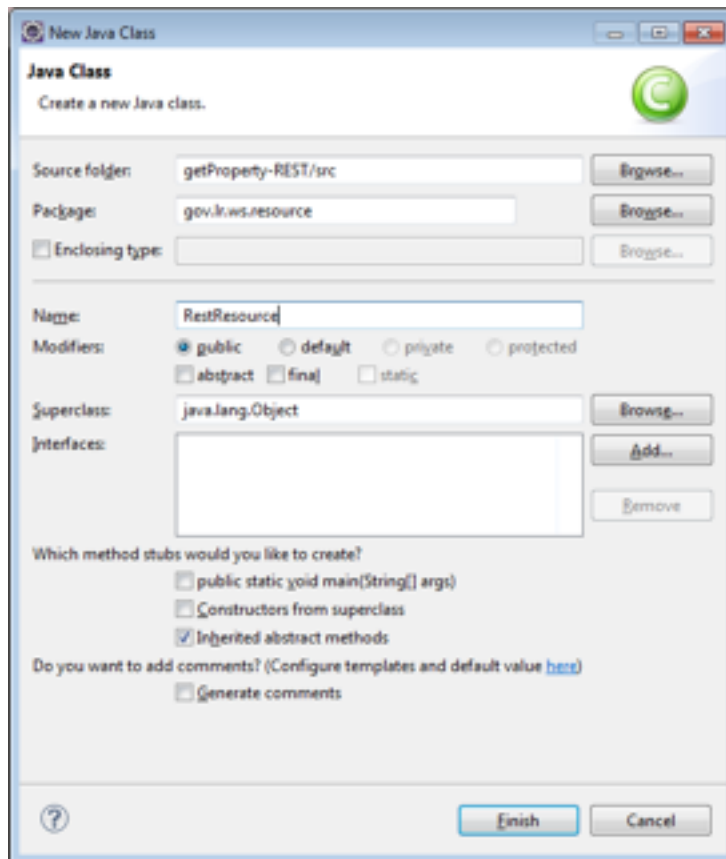
- Package names are written in all lower case to avoid conflict with the names of classes or interfaces.
- Companies use their reversed Internet domain name to begin their package names—for example, com.example.mypackage for a package named mypackage created by a programmer at example.com.

The package is named resource because this is a way of describing the client.

Select **gov.lr.ws.resource** and right click. Select **New** and then **Class**.



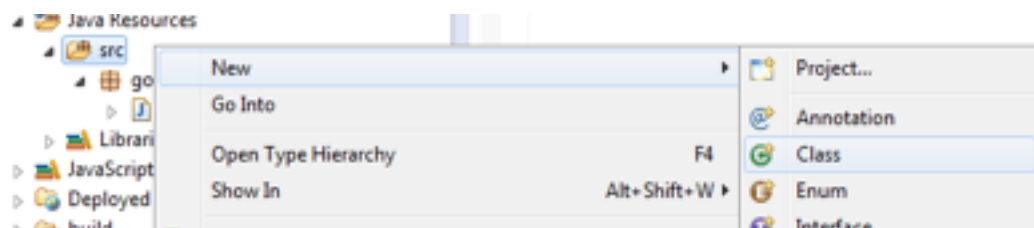
Give the class name of **RestResource** and click **Finish**.



### 3.2 Creating an Application Class

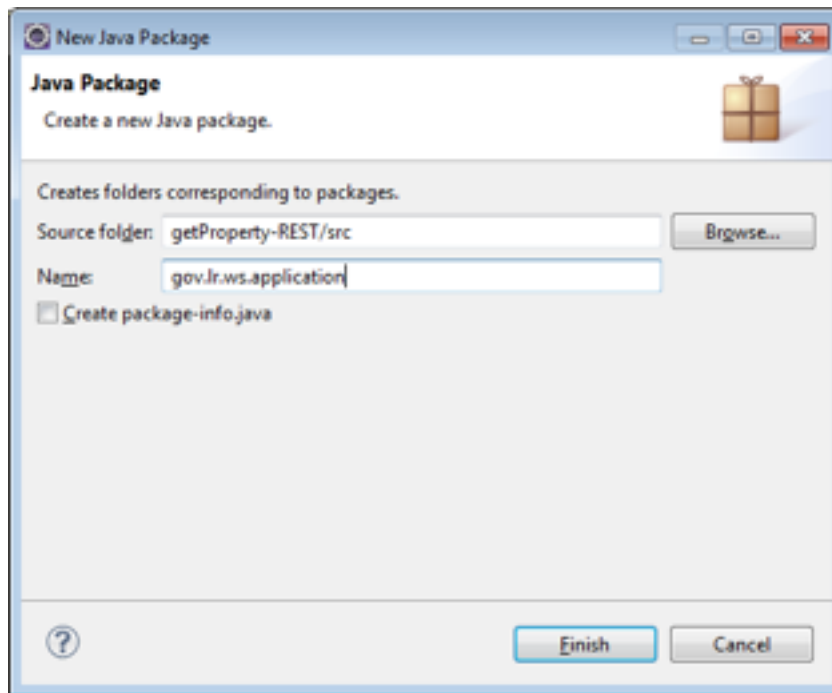
Expand out **Java Resources**.

Right click on **src** and select **New** and then **Package**.



Give the package a name of **gov.lr.ws.application** and click **Finish**

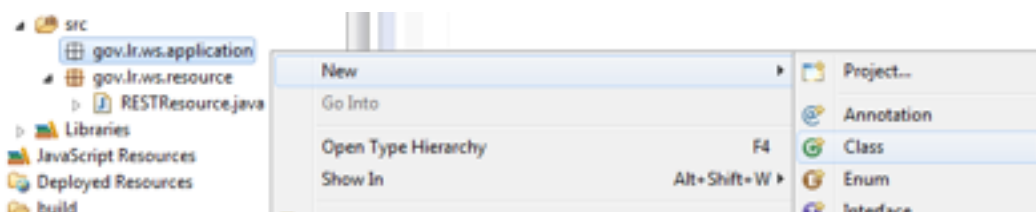




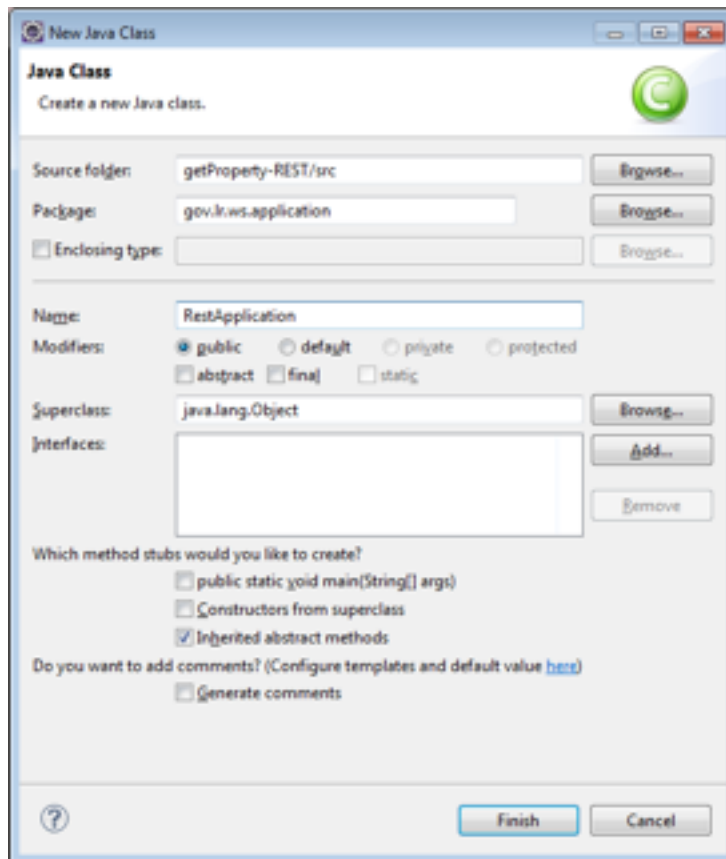
### Why gov.lr.ws.application?

Application is basically a wrapper of the web service. This is the code which will get executed first.

Select **gov.lr.application** and right click. Select **New** and then **Class**.



Give the class name of **RestApplication** and click **Finish**.



### 3.3 Adding code to our Application Class

Eclipse has generated us a template for both the classes we just created, we now need to add some code to make our web service.

Open **RESTApplication.java**

Currently your code looks like:

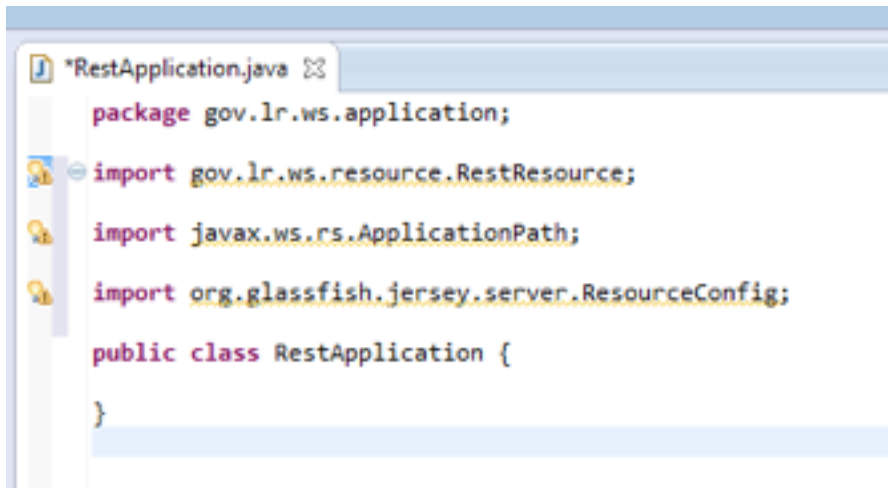


We now need to add our libraries

Below **package gov.lr.ws.application;** add the following:

```
import gov.lr.ws.resource.RestResource;  
  
import javax.ws.rs.ApplicationPath;  
  
import org.glassfish.jersey.server.ResourceConfig;
```

Your code should now look like:



```
*RestApplication.java  
package gov.lr.ws.application;  
  
import gov.lr.ws.resource.RestResource;  
  
import javax.ws.rs.ApplicationPath;  
  
import org.glassfish.jersey.server.ResourceConfig;  
  
public class RestApplication {  
  
}
```

### What have I just done?

You have added function libraries to the to your java code. Importing the function libraries allows your class to use the new functions.

Now we need to change our RestApplication so it uses the Jersey library which will allow it to become a web service.

Change the line:

```
public class RestApplication {
```

To:

```
public class RestApplication extends ResourceConfig {
```

### What have I just done?

In Java, when we wish to extend the usefulness of a class, we can create a new class that inherits the attributes and methods of another. We don't need a copy of the original source code (as is the case with many other languages) to extend the usefulness of a library. We simply need a compiled '.class' file, from which we can create a new enhancement.

Your code should now look like:

```
*RESTPropertyApplication.java
package gov.lr.application;

import gov.lr.resource.RESTPropertyResource;
import javax.ws.rs.ApplicationPath;
import org.glassfish.jersey.server.ResourceConfig;

public class RESTPropertyApplication extends ResourceConfig {
}
```

Now we need to our **RestApplication** to do create a new instance of the **RESTResource** class, which will generate the content which our web service will display.

Within **RESTApplication** method, add:

```
public RestApplication() {
    packages(RestResource.class.getPackage().getName());
}
```

Your code should now looks like:

```
*RestApplication.java RestApplication.java
package gov.lr.ws.application;

import gov.lr.ws.resource.RestResource;
import javax.ws.rs.ApplicationPath;
import org.glassfish.jersey.server.ResourceConfig;

public class RestApplication extends ResourceConfig {
    public RestApplication() {
        packages(RestResource.class.getPackage().getName());
    }
}
```

### What have I just done?

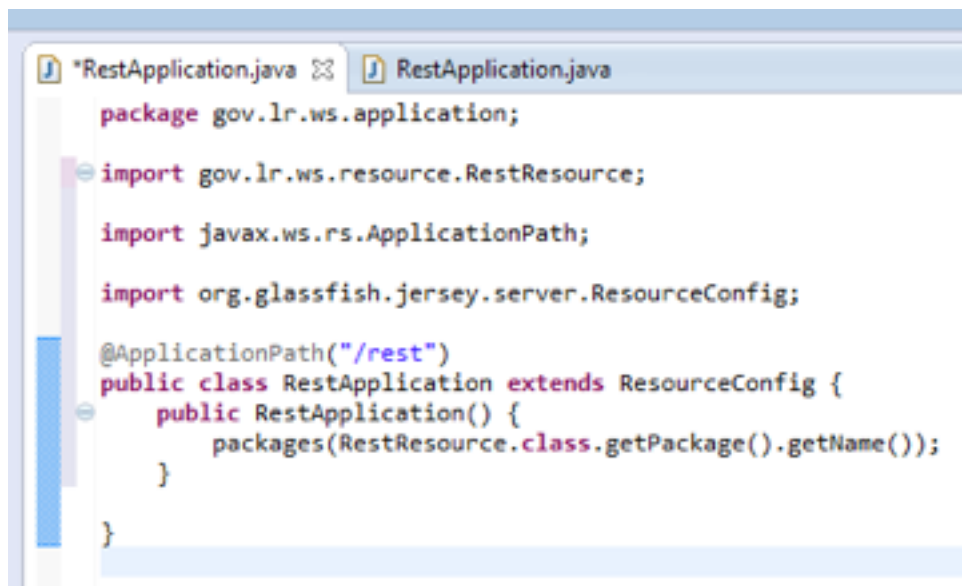
When **RESTApplication** is called, it will look for a function of the same name. In the code above, an instance of **RestApplication** class will be created, and this will then call the function of the same name, this will then initiate the code in the resource file, creating our web service.,

And now we need to add the REST path (what will be displayed in the url) to our application:

Above **public class RestApplication extends ResourceConfig** { Add the following line:

```
@ApplicationPath("/rest")
```

Your code should now looks like:



```
package gov.lr.ws.application;

import gov.lr.ws.resource.RestResource;
import javax.ws.rs.ApplicationPath;
import org.glassfish.jersey.server.ResourceConfig;

@ApplicationPath("/rest")
public class RestApplication extends ResourceConfig {
    public RestApplication() {
        packages(RestResource.class.getPackage().getName());
    }
}
```

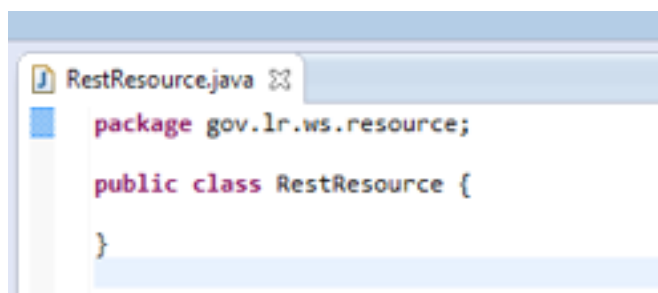
This will mean, our web service currently has a url structure of:

**http://localhost:8080/getProperty-REST/rest**

### 3.4 Adding code to our Resource Class

Open **RESTResource** (In gov.lr.ws.resource).

Currently your code looks like:



```
package gov.lr.ws.resource;

public class RestResource {
}
```

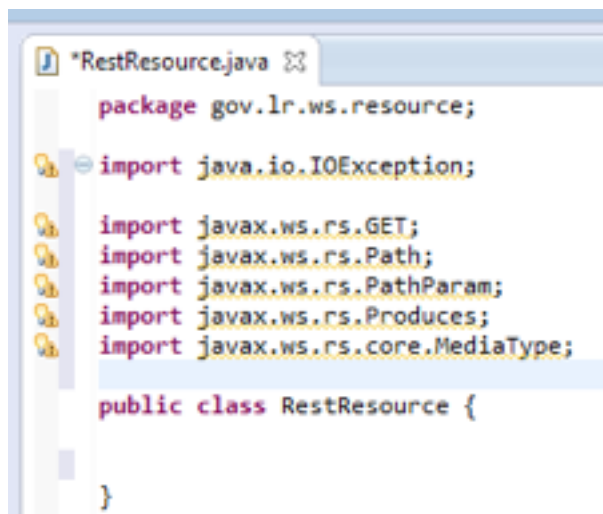
We now need to add our libraries

Below **package gov.lr.ws.resource**; add the following:

```
import java.io.IOException;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
```

Your code should now looks like:

A screenshot of an IDE window titled "RestResource.java". The code is as follows:

```
package gov.lr.ws.resource;

import java.io.IOException;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

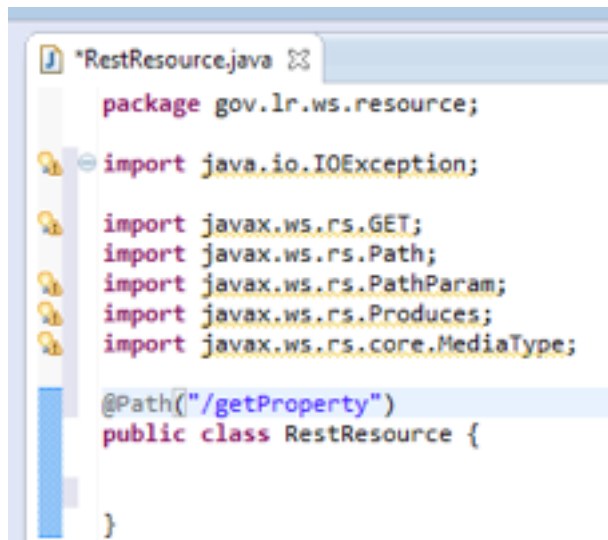
public class RestResource {

}
```

Above **public class RESTPropertyResource {** Add the following line:

```
@Path("/getProperty")
```

Your code should now looks like:



```

*RestResource.java
package gov.lr.ws.resource;

import java.io.IOException;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/getProperty")
public class RestResource {
}

```

This will mean, our web service currently has a url structure of:

**http://localhost:8080/getProperty-REST/rest/getProperty**

We now want to create a simple hello world response. We want to create a GET function which has a URL path of **getmessage**. We want this function to just display a plain text response of "Hello World!"

Within **RESTResource** method, add:

```

@GET
@Path("getmessage")
@Produces(MediaType.TEXT_PLAIN)
public String getmessage() throws IOException {
    return "Hello World!";
}

```

Your code should now looks like:

```
*RestResource.java
package gov.lr.ws.resource;

import java.io.IOException;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/getProperty")
public class RestResource {

    @GET
    @Path("getMessage")
    @Produces(MediaType.TEXT_PLAIN)
    public String getMessage() throws IOException {
        return "Hello World!";
    }
}
```

### What have I just done?

You have written code which has going to respond to a certain URL. It defines the web service “getMessage” as being a GET service and will just respond with text (as apposed to xml or json). The web service will just display on the page the text “Hello World!”.

To call the hello world web service, the URL will be:

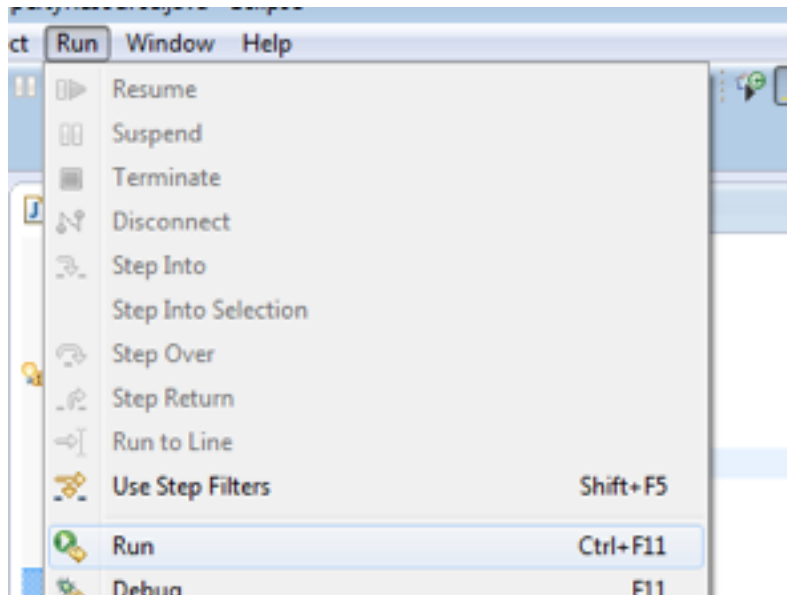
**<http://localhost:8080/getProperty-REST/rest/getProperty/getMessage>**

## 4. Running our web service

### 4.1 Build and Deploy

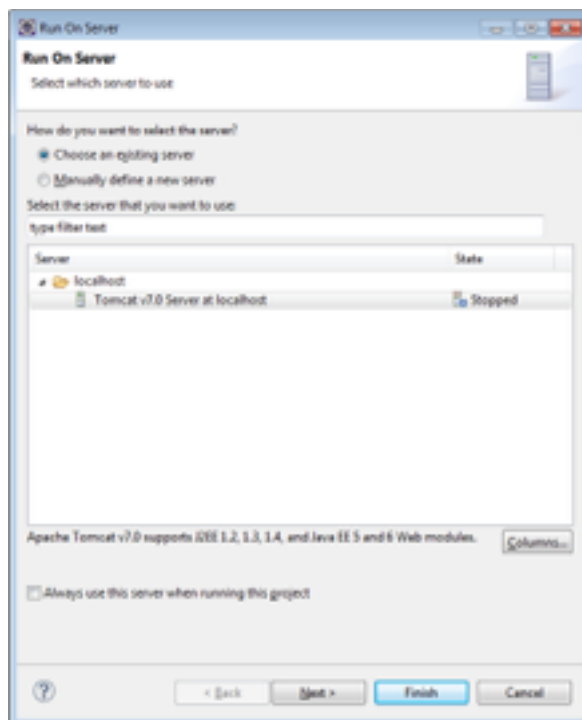
Click on the Run menu, and choose the Run option from the list.



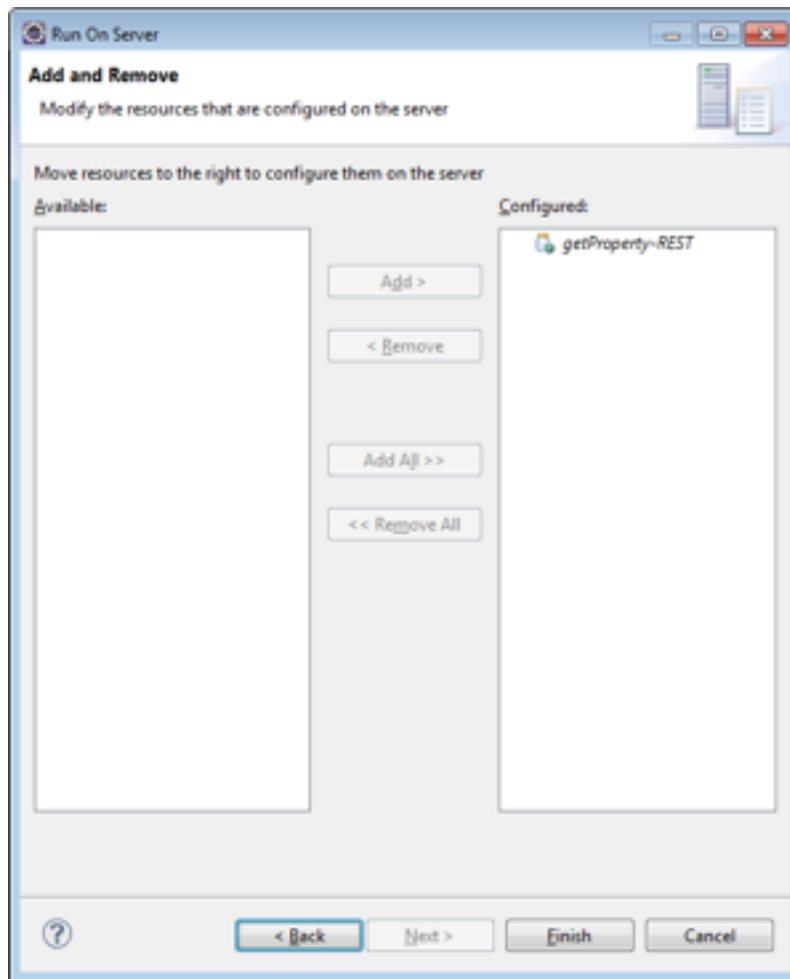


It will ask you to save any unsaved work, click Yes.

On the **Run on Server** screen click **Next**.

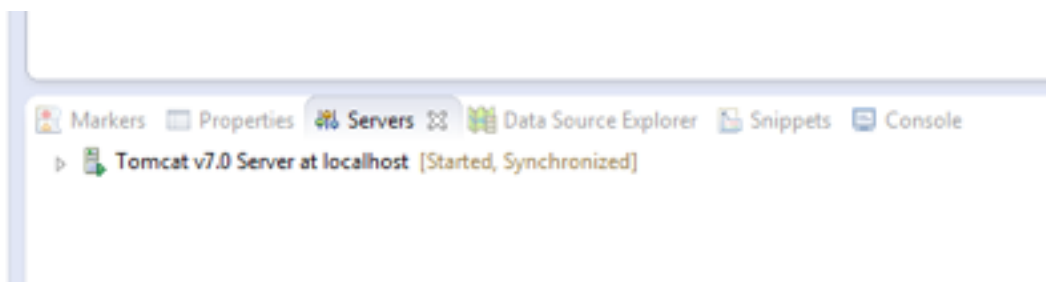


If your application appears on the left menu, select it on the left side and click add.



Click Finish.

In the bottom window, it should say that the Tomcat v7 server has Started and is Synchronized.



### **What have I just done?**

You have compiled your code, started the tomcat web server and deployed your code onto it.

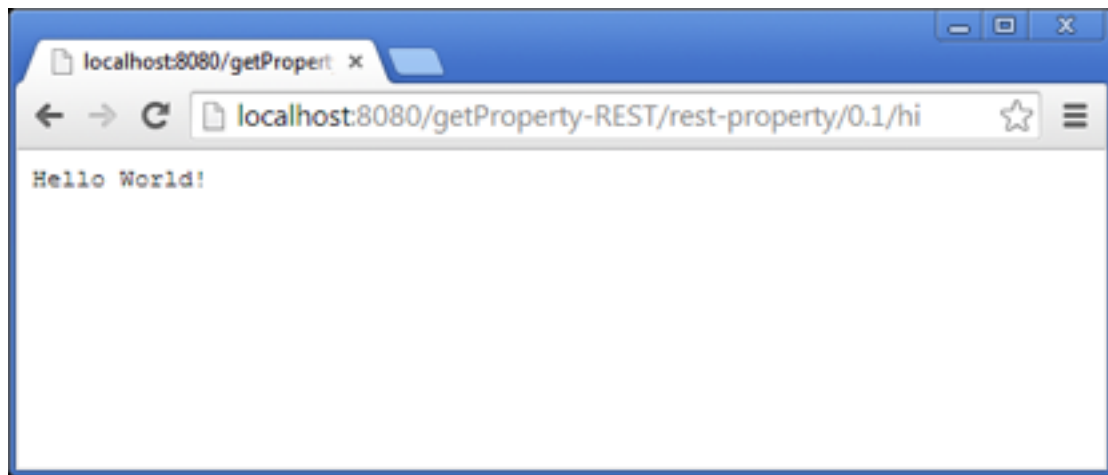
### What is Tomcat?

Tomcat is a web server where you deploy your Java web application to. It provides a "pure Java" HTTP web server environment for Java code to run in.

## 4.2 Viewing our Web service

Open up a web browser and navigate to:

**`http://localhost:8080/getProperty-REST/rest/getProperty/getmessage`**



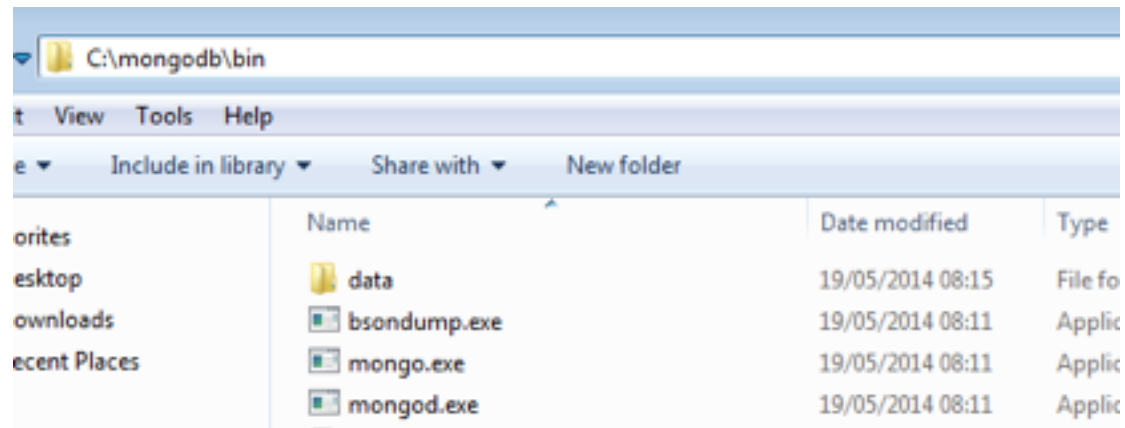
You have successfully created a web service.

## 5. MongoDB

### 5.1 Running MongoDB

Navigate to where you have installed MongoDB. (e.g. C:\mongodb).

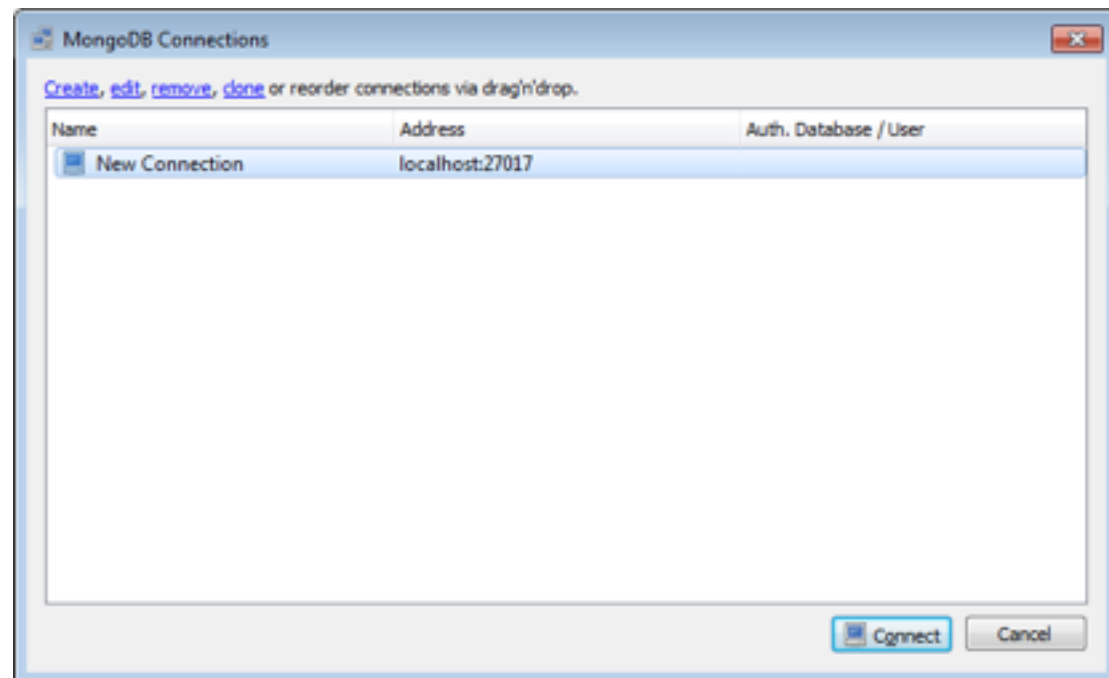
Within the Bin folder, run the mongod.exe file.



This will load a command prompt window, if you close this window you will close your MongoDB server. It is best to minimize this window. If you do accidentally close it, just start mongod.exe again.

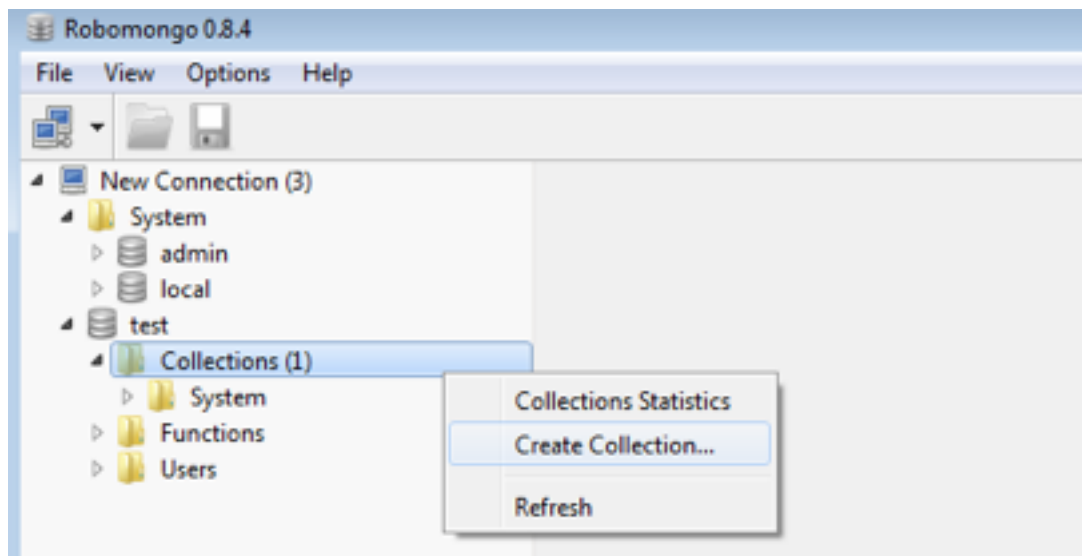
### 5.2 Creating a Collection (aka Table)

Open Robomongo.

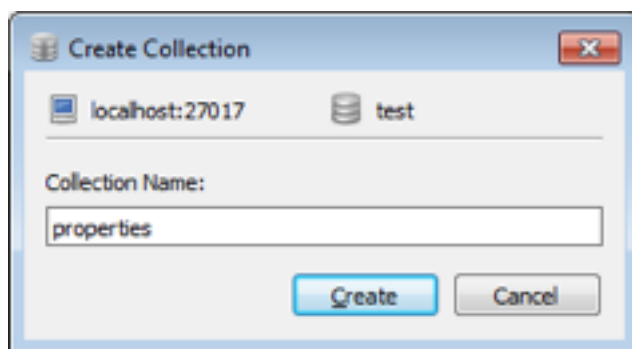


If you don't have any connections listed. Create one for localhost.

Expand test on the right side. Right click on Collections and choose **Create Collection**.

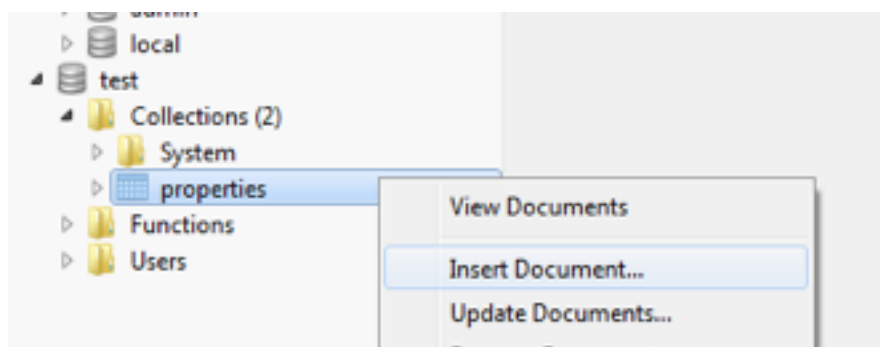


Give the collection a name of **properties**. Click Create



## 5.2 Creating a Document (aka Row)

Right click on properties and click **Insert Document**.

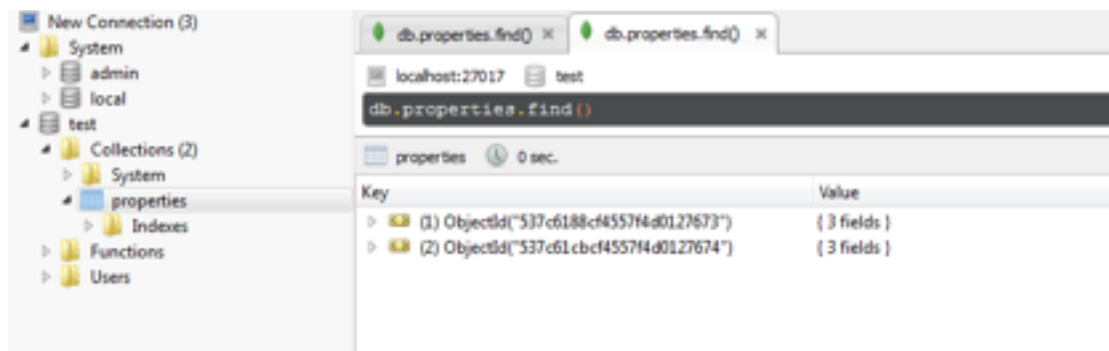


MongoDB uses JSON syntax. We will create a new document which contains two items of data, a name and a title number. Click Save.



Repeat this now to create additional data.

Now double click on properties.



If you expand out the objects you can see the data that was inserted:

Key	Value	Type
(1) ObjectId("537c6188cf4557f4d0127673")	[ 3 fields ]	Object
_id	ObjectId("537c6188cf4557f4d0127673")	ObjectId
name	Andrew	String
title_no	DN100	String
(2) ObjectId("537c61cbcf4557f4d0127674")	[ 3 fields ]	Object

### What have I just done?

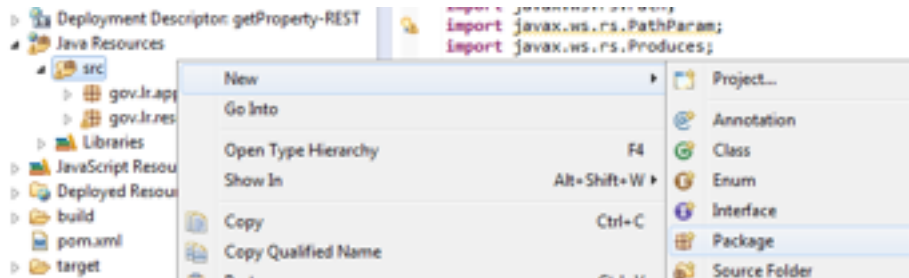
You have a mongodb database running which now has a collection (table) called properties, you have inserted several documents (rows) and you have browsed the data in the collection.

## 6. Connecting Your Java Web Service to MongoDB

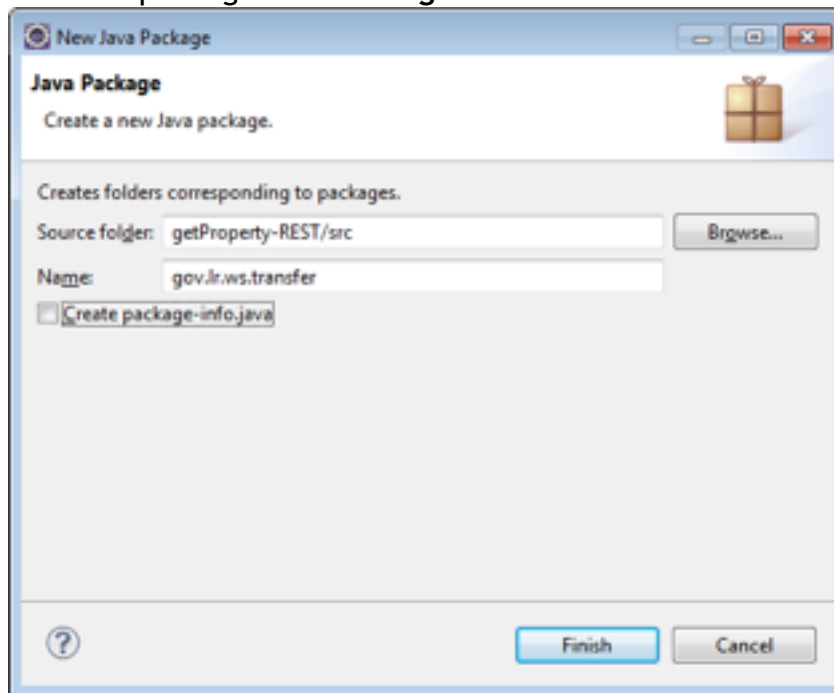
### 6.1. Creating a Transfer Class

Expand out **Java Resources**.

Right click on **src** and select **New** and then **Package**.



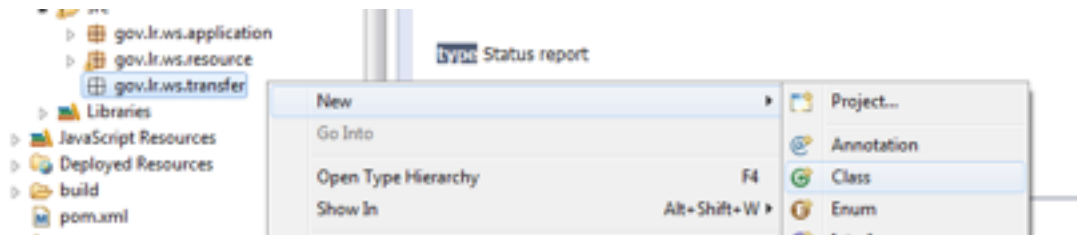
Give the package a name of **gov.lr.ws.transfer** and click **Finish**



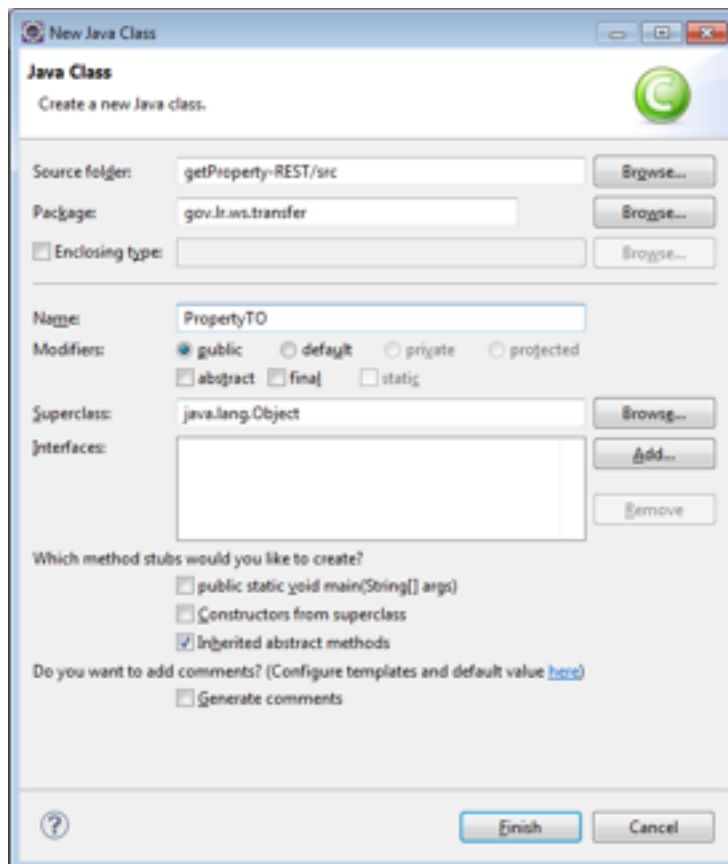
#### Why gov.lr.ws.transfer?

This is used to transfer data as objects. You would define your data structures as classes and use there to pass data as objects between other classes.

Select **gov.lr.ws.transfer** and right click. Select **New** and then **Class**.



Give the class name of **PropertyTO** and click **Finish**.



## 2. Define the structure of your transfer object

We created a mongodb with documents that contain two fields, name and title\_no. We now need to map that structure to the java application.

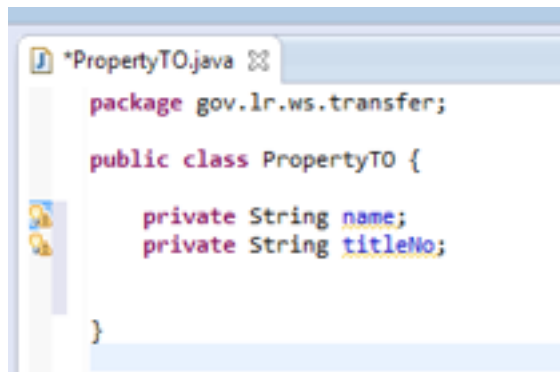
We will define these fields as private variables in this object.

Below **public class PropertyTO {** add

```
private String name;
private String titleNo;
```

Your code should now looks like:





As these are private variables, we won't be able to interact with these, instead we need to create public functions that can interact with these private variables.

### Public vs Private variables

Public variables, are variables that are visible to all classes.

Private variables, are variables that are visible only to the class to which they belong.

Deciding when to use private, protected, or public variables is sometimes tricky. You need to think whether or not an external object (or program), actually needs direct access to the information. If you do want other objects to access internal data, but wish to control it, you would make it either private, but provide functions which can manipulate the data in a controlled way (get() and set() functions).

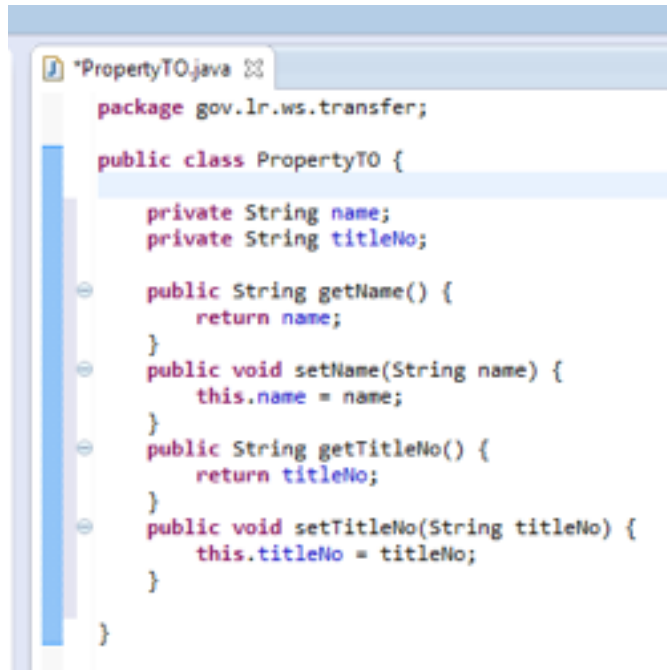
The way to do this is to create get() and set() functions that will update and return these values.

Below **private String titleNo;** add:

```
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
public String getTitleNo() {
    return titleNo;
}
public void setTitleNo(String titleNo) {
    this.titleNo = titleNo;
}
```

Four functions were added. A get and set for name, and a get and set for title number.

Your code should now look like:



```
PropertyTO.java
package gov.lr.ws.transfer;

public class PropertyTO {
    private String name;
    private String titleNo;

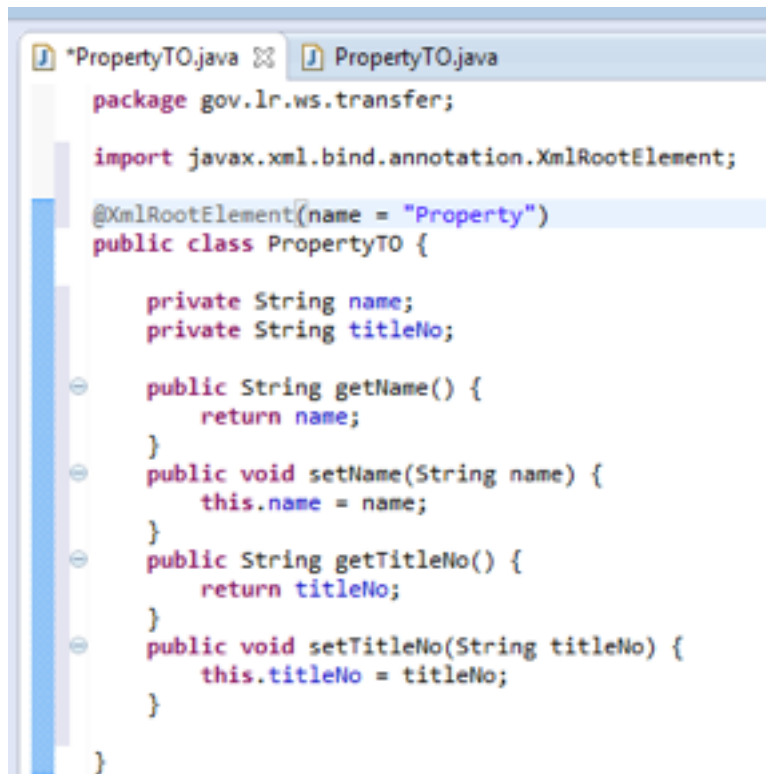
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getTitleNo() {
        return titleNo;
    }
    public void setTitleNo(String titleNo) {
        this.titleNo = titleNo;
    }
}
```

We also want to be able to present this data as an xml object, to do this, below `package gov.lr.ws.transfer;` add:

```
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "Property")
```

Your code should now looks like:



```
package gov.lr.ws.transfer;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "Property")
public class PropertyTO {

    private String name;
    private String titleNo;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getTitleNo() {
        return titleNo;
    }

    public void setTitleNo(String titleNo) {
        this.titleNo = titleNo;
    }

}
```

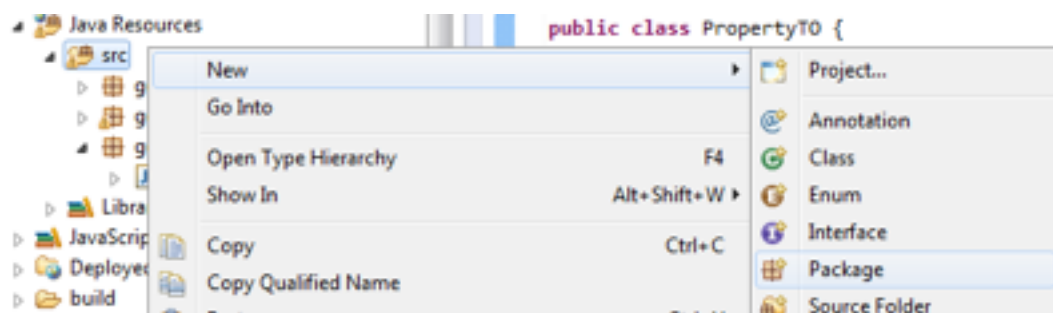
### What have I just done?

You have created an object in java that has a property of name and title\_no. You have created functions that will update those values.

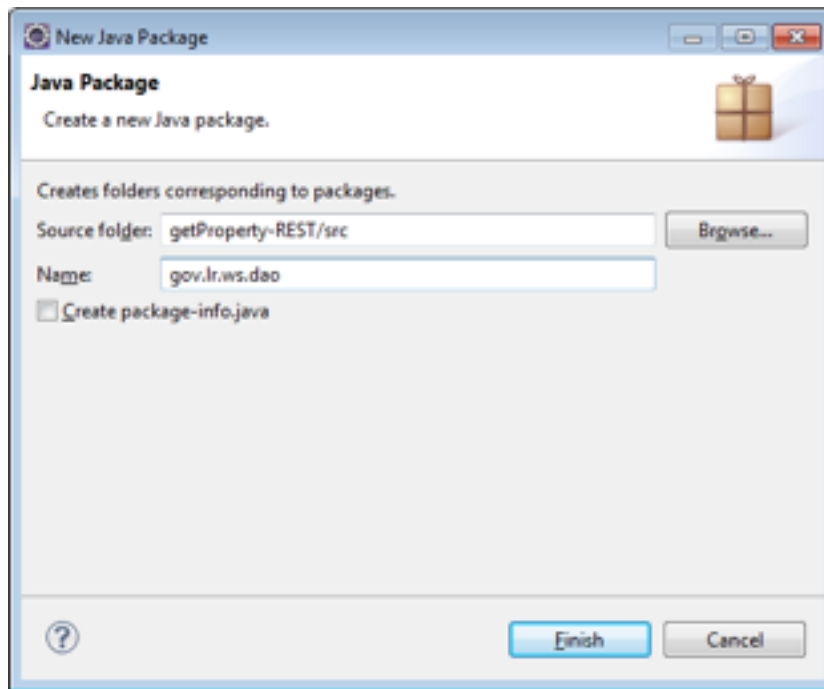
### 3. Create a Data Access Object Class

Expand out **Java Resources**.

Right click on **src** and select **New** and then **Package**.



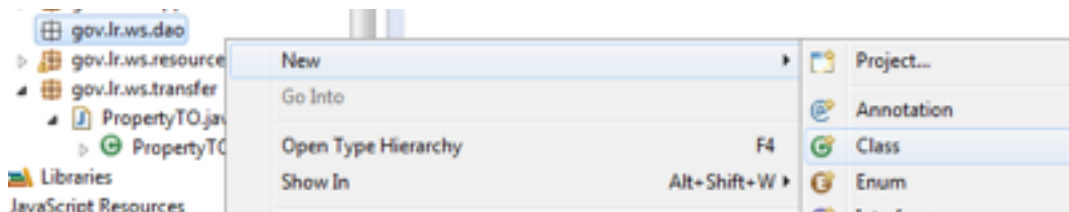
Give the package a name of **gov.lr.ws.dao** and click **Finish**



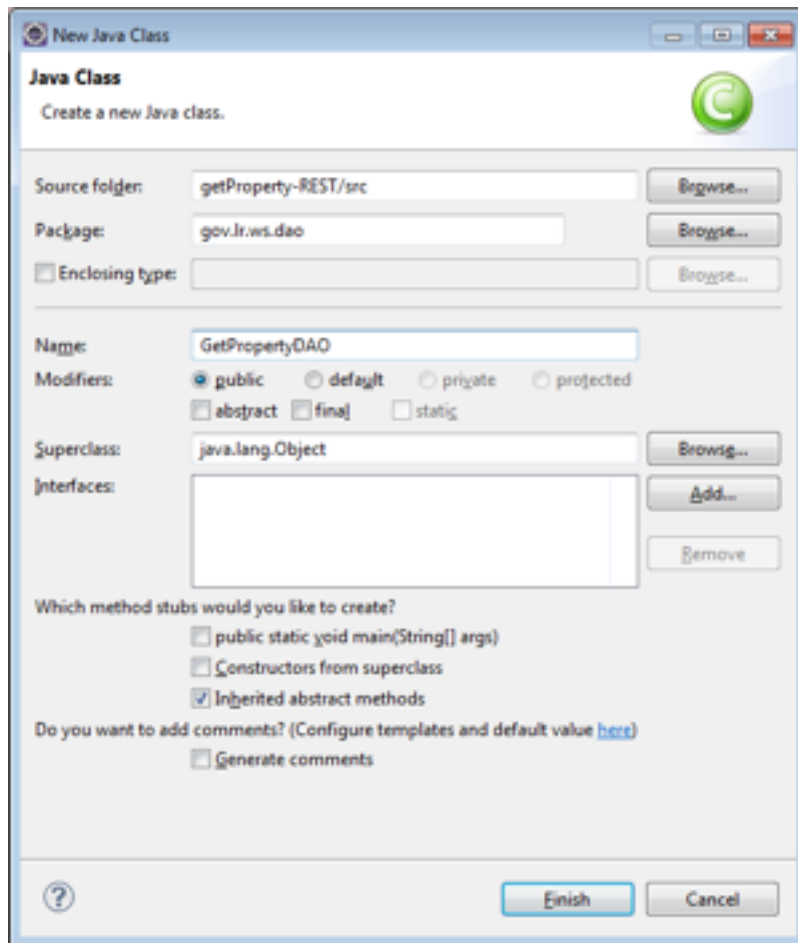
### Why **gov.lr.ws.dao**?

This is the Data Access Layer of your project. This contains the configuration and performs the actions of retrieving data from your database.

Select **gov.lr.ws.dao** and right click. Select **New** and then **Class**.



Give the class name of **GetPropertyInfo** and click **Finish**.



#### 4. Connecting to the MongoDB

We need to make a connection to the MongoDB, and bring back all the data which matches a specific name. As we could return multiple results.

Add the following code below **package gov.lr.ws.dao;**

```
import gov.lr.ws.transfer.PropertyTO;  
  
import java.net.UnknownHostException;  
import java.util.ArrayList;  
  
import com.mongodb.BasicDBObject;  
import com.mongodb.DB;  
import com.mongodb.DBCollection;  
import com.mongodb.DBCursor;  
import com.mongodb.MongoClient;  
import com.mongodb.MongoException;
```

Includes the Property  
Object you just created

Your code should now looks like:

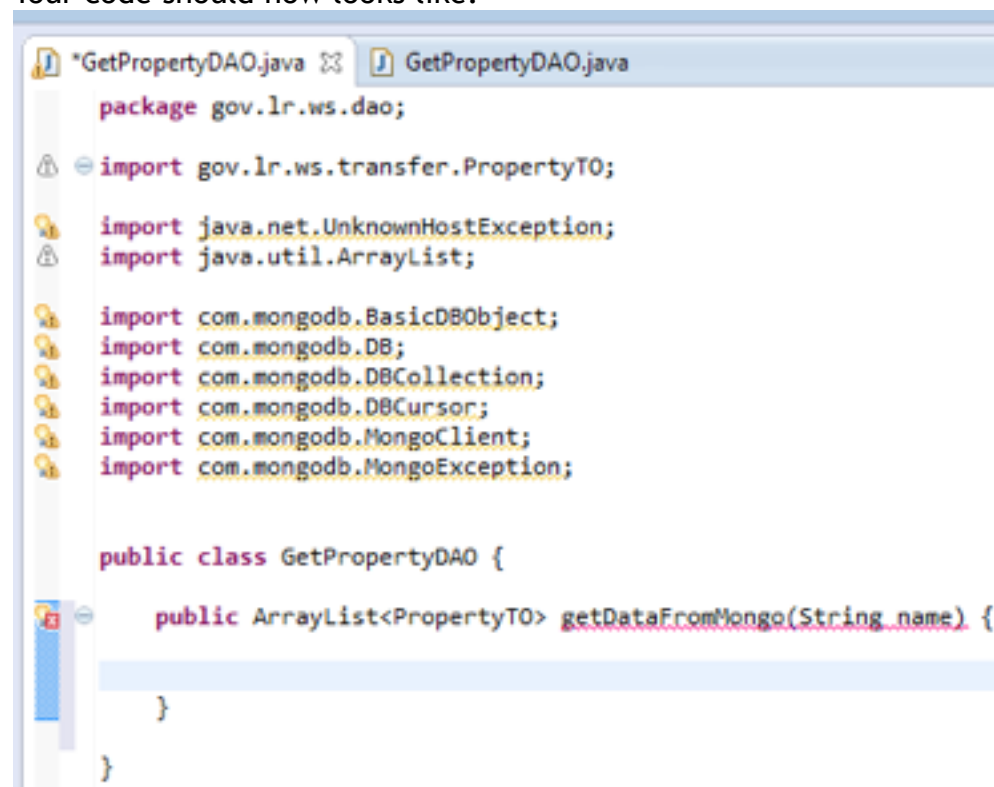
We now need to add a method to the `GetPropertyInfo` class. As our database could contain rows with the same name, we need to code our method to support this. We will create an array of `PropertyTO` (the object class we created earlier which has the properties of name and title\_no).

Below `public class GetPropertyDAO {`, add the following:

```
public ArrayList<PropertyTO> getDataFromMongo(String name) {  
  
}
```

The `ArrayList<PropertyTO>` means this function will return an array of the `PropertyTO` instances.

Your code should now look like:



```
*GetPropertyDAO.java GetPropertyDAO.java  
package gov.lr.ws.dao;  
  
import gov.lr.ws.transfer.PropertyTO;  
import java.net.UnknownHostException;  
import java.util.ArrayList;  
  
import com.mongodb.BasicDBObject;  
import com.mongodb.DB;  
import com.mongodb.DBCollection;  
import com.mongodb.DBCursor;  
import com.mongodb.MongoClient;  
import com.mongodb.MongoException;  
  
public class GetPropertyDAO {  
  
    public ArrayList<PropertyTO> getDataFromMongo(String name) {  
  
    }  
  
}
```

We now want to create a variable called `list`, this variable will be an Array of `PropertyTO`. We also need to return this array as an output of the function.

After `public ArrayList<PropertyTO> getDataFromMongo(String name) {`, add the following:

```

        ArrayList<PropertyTO> list = new ArrayList<PropertyTO>();

        // Comment: MongoDB code here

        return list;

```

Your code should now looks like:

```

package gov.lr.ws.dao;

import gov.lr.ws.transfer.PropertyTO;
import java.net.UnknownHostException;
import java.util.ArrayList;

import com.mongodb.BasicDBObject;
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.DBCursor;
import com.mongodb.MongoClient;
import com.mongodb.MongoException;

public class GetPropertyDAO {

    public ArrayList<PropertyTO> getDataFromMongo(String name) {

        ArrayList<PropertyTO> list = new ArrayList<PropertyTO>();

        // Comment: MongoDB code here

        return list;

    }

}

```

We now want to connect to mongoDB and perform a query on the name field.

Replace the comment **// Comment: MongoDB code here** with:

Connect to MongoDB

```

MongoClient mongo = new MongoClient("localhost", 27017);

DB db = mongo.getDB("test");

DBCollection table = db.getCollection("properties");

BasicDBObject searchQuery = new BasicDBObject();
searchQuery.put("name", name);
DBCursor cursor = table.find(searchQuery);

// Comment: MongoDB Loop

```

Your code should now look like (Part of the header has been skipped):

```
import java.util.ArrayList;

import com.mongodb.BasicDBObject;
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.DBCursor;
import com.mongodb.MongoClient;
import com.mongodb.MongoException;

public class GetPropertyDAO {

    public ArrayList<PropertyTO> getDataFromMongo(String name) {

        ArrayList<PropertyTO> list = new ArrayList<PropertyTO>();

        MongoClient mongo = new MongoClient("localhost", 27017);

        DB db = mongo.getDB("test");

        DBCollection table = db.getCollection("properties");

        BasicDBObject searchQuery = new BasicDBObject();
        searchQuery.put("name", name);
        DBCursor cursor = table.find(searchQuery);

        // Comment: MongoDB Loop

        return list;

    }

}
```

Replace the comment **// Comment: MongoDB Loop** with:

Loop through the documents returned

```
while (cursor.hasNext()) {

    PropertyTO data = new PropertyTO();

    cursor.next();

    data.setName(cursor.curr().get("name").toString());
    data.setTitleNo(cursor.curr().get("title_no").toString());

    list.add(data);

}
```



Your code should now look like (Part of the header has been skipped):

```
public class GetPropertyDAO {  
    public ArrayList<PropertyTO> getDataFromMongo(String name) {  
        ArrayList<PropertyTO> list = new ArrayList<PropertyTO>();  
        MongoClient mongo = new MongoClient("localhost", 27017);  
        DB db = mongo.getDB("test");  
        DBCollection table = db.getCollection("properties");  
        BasicDBObject searchQuery = new BasicDBObject();  
        searchQuery.put("name", name);  
        DBCursor cursor = table.find(searchQuery);  
        while (cursor.hasNext()) {  
            PropertyTO data = new PropertyTO();  
            cursor.next();  
            data.setName(cursor.curr().get("name").toString());  
            data.setTitleNo(cursor.curr().get("title_no").toString());  
            list.add(data);  
        }  
        return list;  
    }  
}
```

Your code currently won't build because there is an error on the MongoClient line (the red x in the side menu shows this). You need to add exception handling around the mongo collection.

Select all the code as demonstrated below:

```

public class GetPropertyDAO {

    public ArrayList<PropertyTO> getDataFromMongo(String name) {

        ArrayList<PropertyTO> list = new ArrayList<PropertyTO>();

        MongoClient mongo = new MongoClient("localhost", 27017);

        DB db = mongo.getDB("test");

        DBCollection table = db.getCollection("properties");

        BasicDBObject searchQuery = new BasicDBObject();
        searchQuery.put("name", name);
        DBCursor cursor = table.find(searchQuery);

        while (cursor.hasNext()) {

            PropertyTO data = new PropertyTO();

            cursor.next();

            data.setName(cursor.curr().get("name").toString());
            data.setTitleNo(cursor.curr().get("title_no").toString());

            list.add(data);

        }

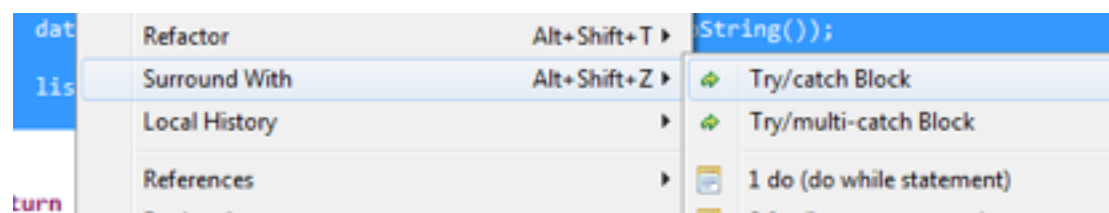
        return list;

    }

}

```

Right click; choose **Surround With** and then **Try/Catch Block**.



Your code will now appear as:

```
public class GetPropertyDAO {  
    public ArrayList<PropertyTO> getDataFromMongo(String name) {  
        ArrayList<PropertyTO> list = new ArrayList<PropertyTO>();  
        try {  
            MongoClient mongo = new MongoClient("localhost", 27017);  
            DB db = mongo.getDB("test");  
            DBCollection table = db.getCollection("properties");  
            BasicDBObject searchQuery = new BasicDBObject();  
            searchQuery.put("name", name);  
            DBCursor cursor = table.find(searchQuery);  
            while (cursor.hasNext()) {  
                PropertyTO data = new PropertyTO();  
                cursor.next();  
                data.setName(cursor.curr().get("name").toString());  
                data.setTitleNo(cursor.curr().get("title_no").toString());  
                list.add(data);  
            }  
        } catch (UnknownHostException e) {  
            // TODO Auto-generated catch block  
            e.printStackTrace();  
        }  
    }  
}
```

Start of the Try Catch Block

End of the Try Catch Block

### What have I just done?

You have created an object in java that will connect to MongoDB. A query will be performed to return all data matches the name entered. The values returned from MongoDB query are written to an instance of the PropertyTO object, this instance is then added to an array which is then returned from this function.

### What is a Try/Catch Block?

If an exception occurs within the try block, that exception is handled by an exception handler associated with it. To associate an exception handler with a try block, you must put a catch block after it. In the catch block, Exception handlers can do more than just print error messages or halt the program. They can do error recovery, prompt the user to make a decision, or propagate the error up to a higher-level handler.

## 5. Adding a Web Service for the MongoDB Query

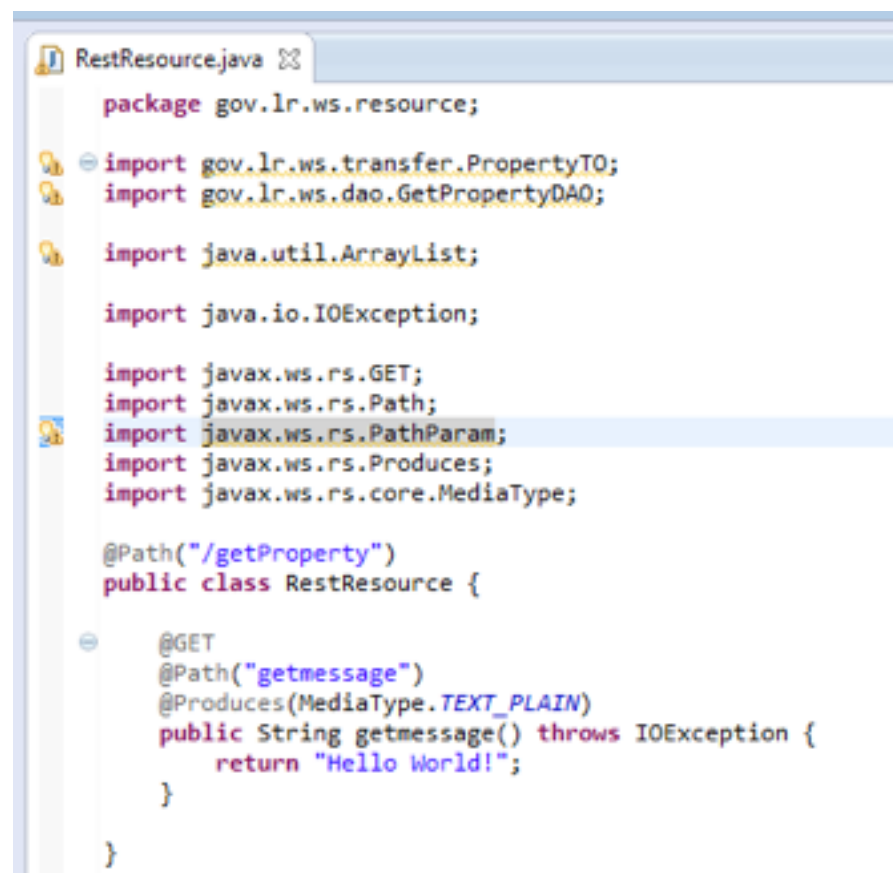
We have defined our object and made the query, but now we want to create a method in the web service to access this data.

Open the RestResource files (under gov.lr.ws.resource).

Below **package gov.lr.ws.resource;**, add the following code:

```
import gov.lr.ws.transfer.PropertyTO;  
import gov.lr.ws.dao.GetPropertyDAO;  
  
import java.util.ArrayList;
```

Your code will now appear as:



```
RestResource.java  
  
package gov.lr.ws.resource;  
  
import gov.lr.ws.transfer.PropertyTO;  
import gov.lr.ws.dao.GetPropertyDAO;  
  
import java.util.ArrayList;  
  
import java.io.IOException;  
  
import javax.ws.rs.GET;  
import javax.ws.rs.Path;  
import javax.ws.rs.PathParam;  
import javax.ws.rs.Produces;  
import javax.ws.rs.core.MediaType;  
  
@Path("/getProperty")  
public class RestResource {  
  
    @GET  
    @Path("getmessage")  
    @Produces(MediaType.TEXT_PLAIN)  
    public String getmessage() throws IOException {  
        return "Hello World!";  
    }  
  
}
```

This makes the RestResource code aware of the PropertyTO object, and the GetPropertyDAO function which generated the array of PropertyTO instances.

We now want to create a web service to display the array of PropertyTO instances. We want to create a GET function which has a URL path of `getpropertiesbyname` which allows for a value to be passed in to search on. We want this function to just display either an XML or JSON response. The function will display an array of PropertyTO instances.

```

@GET
@Path("/getpropertysbyname/{varName}")
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML}) //
APPLICATION_XML
public ArrayList<PropertyTO>
getPropertysFromMongoDB2(@PathParam("varName") String varName) {

}

```

This means the url of the web service will be:

**http://localhost:8080/getProperty-REST/rest/getProperty/getpropertysbyname /<name>**

With <name> being a variable.

Your code will now appear as:

```

import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/getProperty")
public class RestResource {

    @GET
    @Path("/getMessage")
    @Produces(MediaType.TEXT_PLAIN)
    public String getMessage() throws IOException {
        return "Hello World!";
    }

    @GET
    @Path("/getpropertysbyname/{varName}")
    @Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML}) //APPLICATION_XML
    public ArrayList<PropertyTO> getPropertysFromMongoDB2(@PathParam("varName") String varName) {

    }

}

```

Inside the getPropertysFromMongoDB2 function, add the following code:

```

GetPropertyDAO dao = new GetPropertyDAO();
ArrayList<PropertyTO> list = new ArrayList<PropertyTO>()

list = dao.getDataFromMongo(varName);

return list;

```

Creates an instance of the GetPropertyInfo class. Saves this instance as a variable

Returns the variable list as an output.

Your code will now appear as:

```

import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/getProperty")
public class RestResource {

    @GET
    @Path("/getMessage")
    @Produces(MediaType.TEXT_PLAIN)
    public String getMessage() throws IOException {
        return "Hello World!";
    }

    @GET
    @Path("/getPropertyByName/{varName}")
    @Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML}) //APPLICATION_XML
    public ArrayList<PropertyTO> getPropertyFromMongoDB2(@PathParam("varName") String varName) {
        GetPropertyDAO dao = new GetPropertyDAO();
        ArrayList<PropertyTO> list = new ArrayList<PropertyTO>();

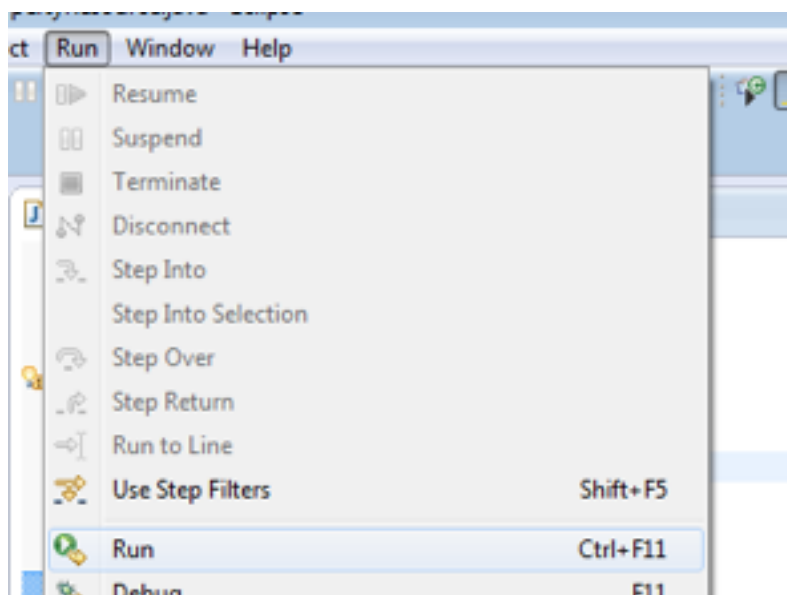
        list = dao.getDataFromMongo(varName);

        return list;
    }
}

```

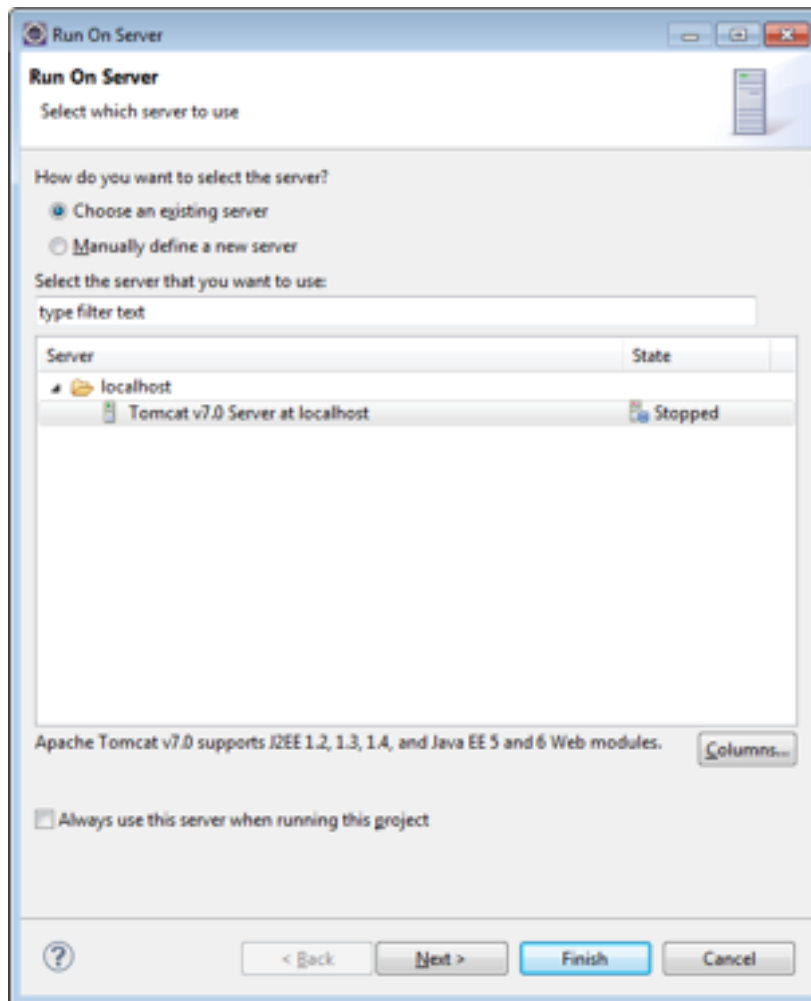
## 6. Build and Deploy

Click on the Run menu, and choose the Run option from the list.



It will ask you to save any unsaved work, click Yes.

On the Run on Server screen click Finish.



## 7. Viewing our Web service

Open up a web browser and navigate to:

<http://localhost:8080/getProperty-REST/rest/getProperty/getpropertybyname/Andrew>

