

Upload Image to database and Access Java Libraries

In this first part of the tutorial, we'll create a simple application to upload photos through a simple HTML form, and store them in the database. Then, render and display them on demand. The steps we'll go through are as follows:

Create a model. We'll use this model for the photographs, and we'll call it "Photo"

- **Create a migration.** Use a database migration to create a table in the database to hold the photographs
- **Handle the upload.** In the view, we'll create a template containing a HTML multi-part form and a corresponding action that will allow users to upload their photograph files
- **Render the image from the database.** Once you've got the images in the database, how do you pull them out and display them again? That's what I'll show you...

The second part of this tutorial will show you how to get started using Java libraries in a Rails application by doing simple image processing with the Java 2D API.

A primary advantage of developing with JRuby is that you have access to Java libraries from a Rails application. For example, say you might want to create an image database and a web application that allows processing of the images. You can use Rails to set up the database-backed web application and use the powerful Java 2DTM API for processing the images on the server-side.

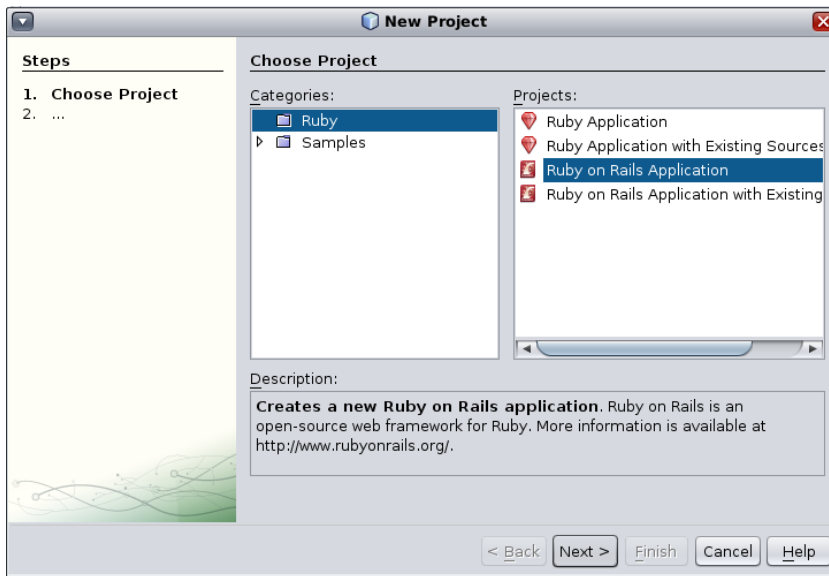
The steps we'll go through are as follows:

1. Giving your model access to Java libraries.
2. Creating constants to refer to Java classes.
3. Performing file input and output using the `java.io` and `javax.imageio` packages.
4. Assigning Java objects to Ruby objects.
5. Calling Java methods and using variables.
6. Converting arrays from Java language arrays to Ruby arrays.
7. Streaming files to the client.

Part 0 – Create a new Ruby on Rails project and MySQL Database:

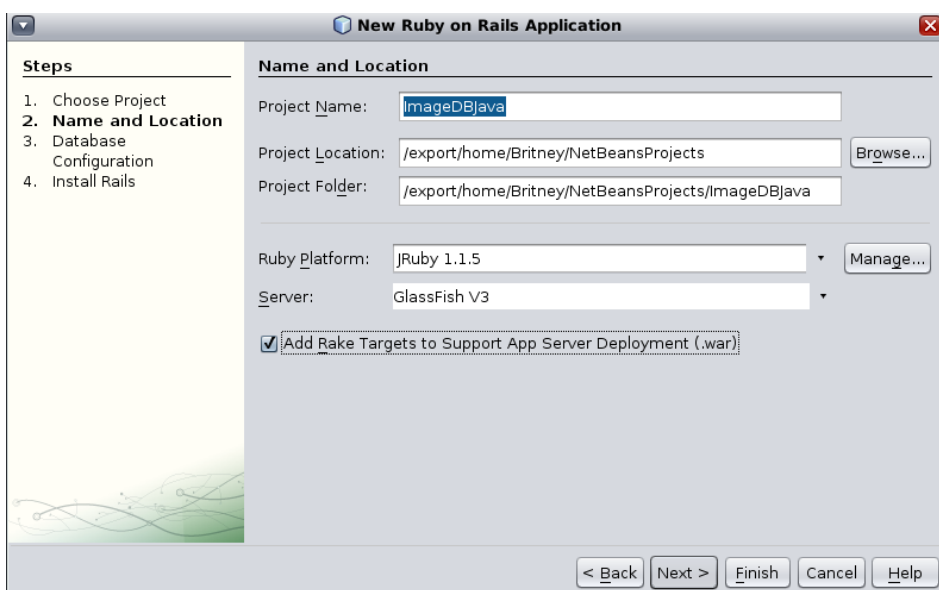
1. **Create a new project:** Go to File => New Project

Select Category: Ruby and Project: Ruby on Rails application.



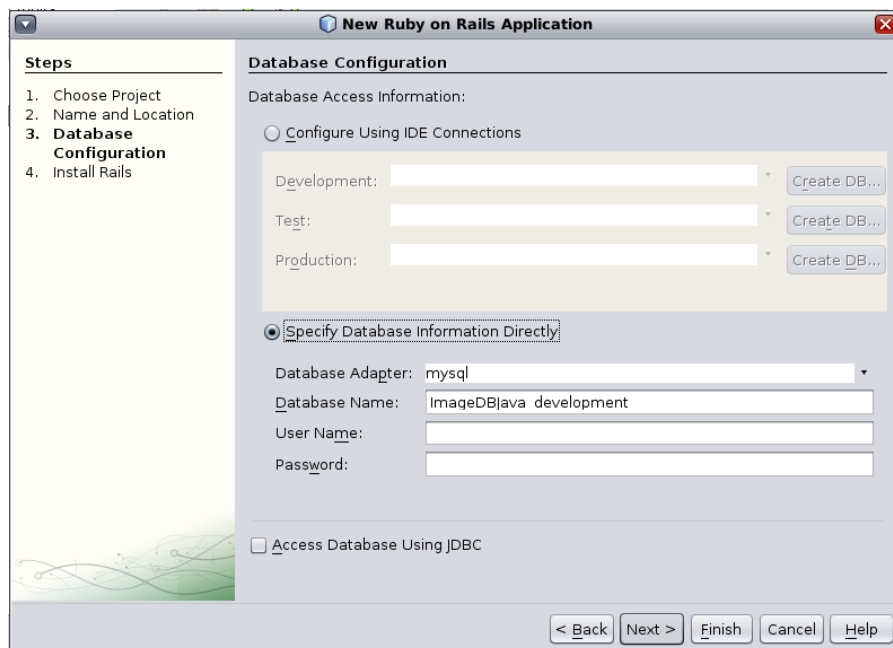
Click Next.

2. **Name your project** "ImageDBJava" and select "Add Rake Target to support App Server (.war)"



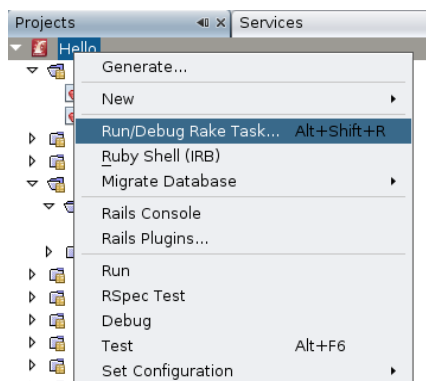
Click Next

3. Choose "Specify Database information Directly"

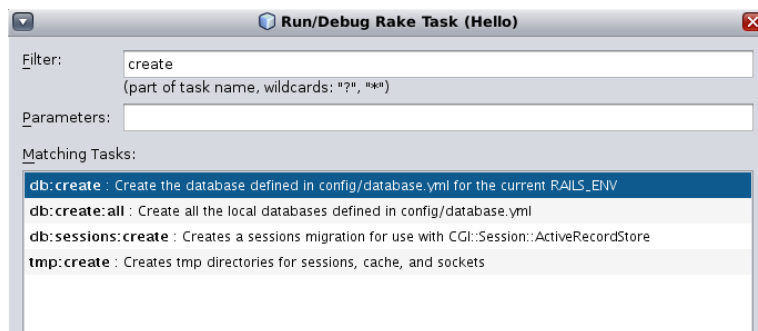


Click Finish

4. Create Database – Right click on the project and select "Run/Debug Rake Task..."



In the Filter field, write "create" the machine task will be db:create

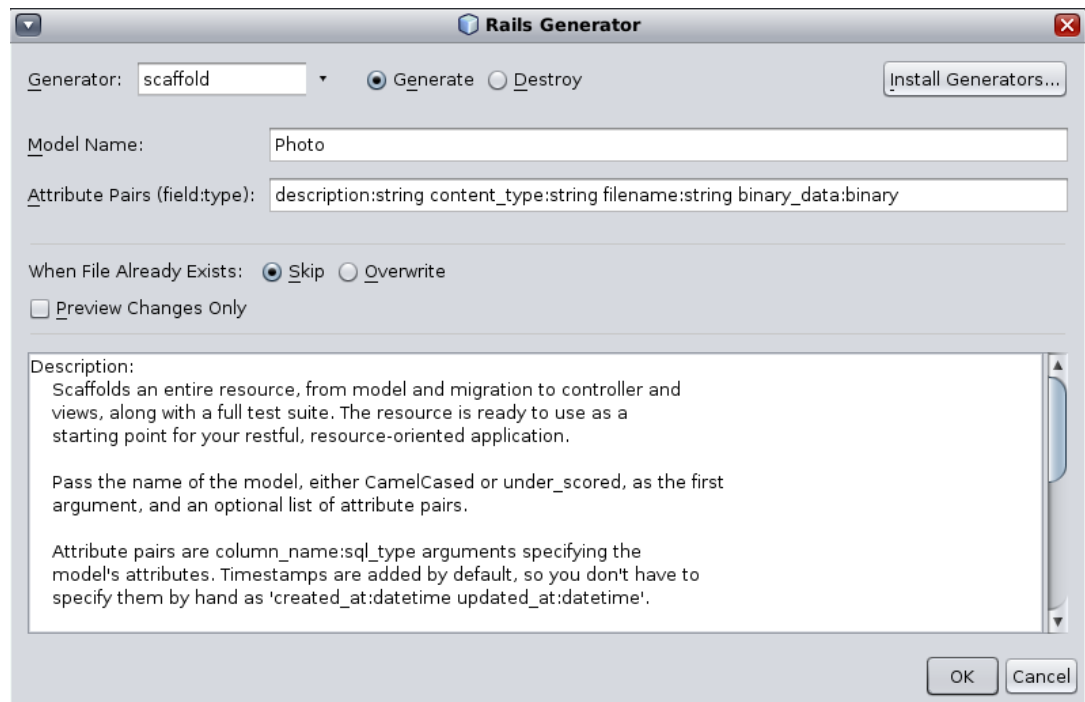


Click RUN

5. Go back to "Projects" Tab.

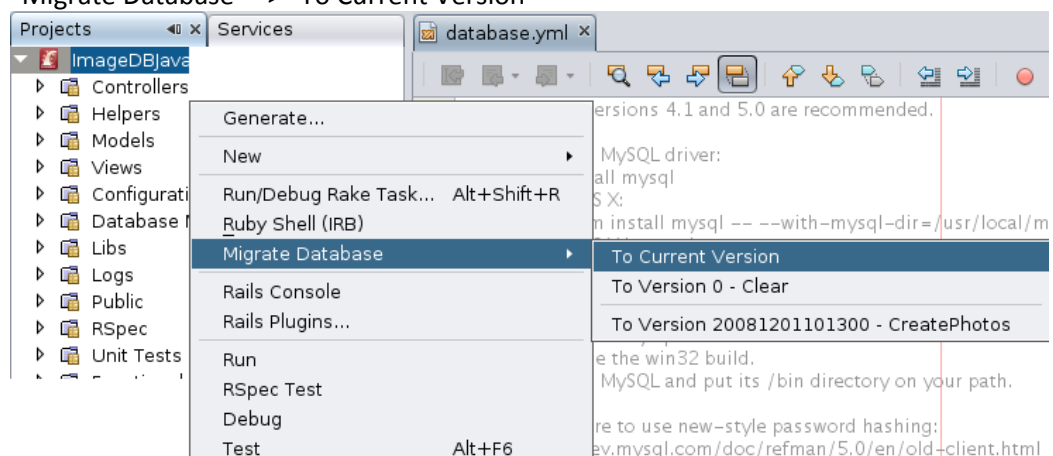
Part 1 - Upload Image to database

1. We'll use Rails' scaffolding generator to create the MVC and database migration file – Right click on the project and select "Generate"
2. **Scaffold** - Choose scaffold Generator, give a name to the model "**Photo**" and at the Attributes Pairs write:
description:string content_type:string filename:string binary_data:binary



Click OK

3. **Migrate database** – Right click on the project and select :
"Migrate Database" => "To Current Version"



*If all has gone well, you should now have a table named **photos** in your database with all the fields you need to hold your image data.*

*Now we have the foundations laid, it's time to handle uploading a new photograph to the database. Rails' scaffolding generator has already created needed actions in the **photos_controller.rb** controller (create, update, delete, show, etc.), so we don't need to add code! Instead we'll concentrate on the view for the new action. The scaffold generator created the view according to the table field. We want the view to enable just a file upload, and the It turns out that when uploading a file, you don't get receive just one string, such as the binary image data, but a more complex object containing not just the file's binary data but its filename and content-type (such as **image/jpg** or **image/gif**), We'll store this information in :image_file.*

4. Expand "Views" => "photos" and open the "new.rhtml" file and change it to look like this:

```
<h1>New photo</h1>

<% form_for(:photo, @photo, :url => {:action=>'create'}, :html=> {:multipart=>true})
do |f| %>

  <%= f.error_messages %>
  <p>
    <%= f.label :image_file %><br />
    <%= f.file_field :image_file %>
  </p>
  <p>
    <%= f.label :description %><br />
    <%= f.text_field :description %>
  </p>
  <p>
    <%= f.submit "Create" %>
  </p>
<% end %>

<%= link_to 'Back', photos_path %>
```

5. Now what we'll need to do is extract the data and assign it the correct model attributes. The best place to do that is in the photo model, in `app/models/photo.rb`. Add this code to the model, notice we're adding validation too.

```
validates_presence_of :description

def validate
  errors.add(:image_file, "Please upload image") unless binary_data
end

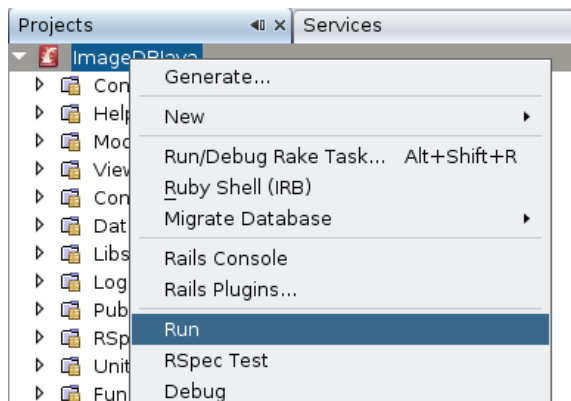
def image_file=(input_data)
  if input_data.blank?
    self.binary_data = nil
    self.filename = nil
    self.content_type = nil
  else
    self.filename = input_data.original_filename
    self.content_type = input_data.content_type.chomp
    self.binary_data = input_data.read
  end
end
```

Here, we take the contents of `image_file` and we use three methods to extract the data and assign it to the model attributes that match our database table: the methods are `original_filename`, `content_type` and `read`.

- **`original_filename`** gives you surprisingly enough the original filename of the file
- **`content_type`** provides you with the content-type of the data, such as whether it is an image/gif or an audio/wav which is useful for validation
- **`read`** lets you actually get at the binary data

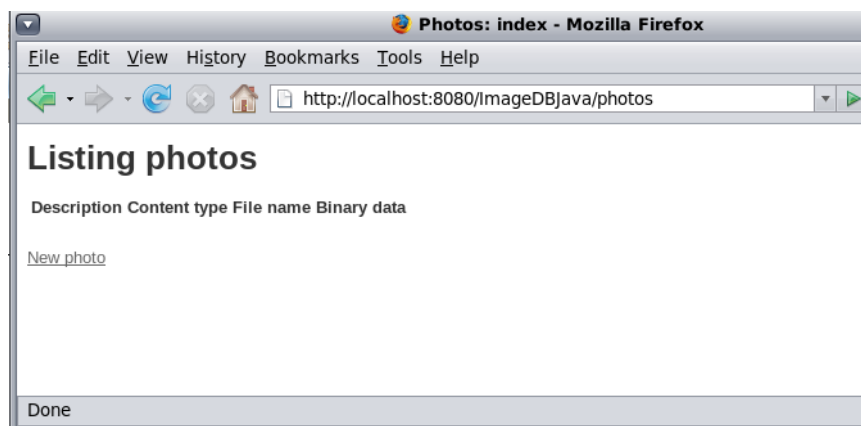
*The **chomp** method at the end of **content_type** simply removes any extraneous newline characters, to make it neater.*

Now, if you run the application , Right click on the project and select "Run"



NetBeand will deploy the application to Glassfish and open Firefox to the root directory of the application.

Direct your browser to <http://localhost:8080/ImageDBJava/photos>,



You should be able to upload a file. Try a small one at first (under 10k would be good) You'll know if it works because you'll be returned to the /photos/list action, and you'll see the details of your file, along with a lot of gibberish under the binary column. That's because Rails is rendering your binary image data as text. Instead, we want a way to get the binary image data back out of the database and display it as an actual image...

Rendering an image stored in the database

1. First we must register the jpg type for use in respond_to blocks

Add this line to `configuration/initializers/mime_types.rb`

```
Mime::Type.register_alias "image/jpeg", :jpg
```

2. Change the views:

- a. Change views/photos/index.html.erb:

```
<td><%=h photo.binary_data %></td>
```

to:

```
<td><%= image_tag(formatted_photo_path(photo, "jpg"), :alt => photo.filename) %></td>
```

such that the image will be rendered to the browser, and not the binary data

- b. Change views/photos/show.html.erb:

```
<p>
  <b>Binary data:</b>
  <% h @photo.binary_data=%>
</p>
```

to:

```
<p>
  <b>Image:</b>
  <%= image_tag(formatted_photo_path(@photo, "jpg"), :alt => @photo.filename =%>
</p>
```

3. To make it work, we'll add a new option in the "show" method in our `photo_controller.rb` controller, change the "show" method to look like this:

```
# GET /photos/1
# GET /photos/1.xml
# GET /photos/1.jpg
# GET /photos/1.jpg?bw=true
def show
  @photo = Photo.find(params[:id])

  respond_to do |format|
    format.html # show.html.erb
    format.xml { render :xml => @photo }
    format.jpg {
      data = params[:bw] ? @photo.bw_image : @photo.binary_data
      send_data(data, :type => @photo.content_type, :filename => @photo.filename,
        :disposition => 'inline')
    }
  end
end
```

In the above action we have taken the `id` of the `Photo` from the params supplied by the form, and retrieved it from the database into the `@Photo` object, then by

checking the `bw` parameter, Boolean parameter, we know if to show it in colors or black and white.

The binary data extracted out of the `binary_data` field and used Rails' `send_data` method to render the binary image to the browser, is `bw` is true, the binary data goes through the `bw_image` method we'll add to the model.

The `send_data` method can take several options:

- a. `:filename` - suggests a filename for the browser to use.
- b. `:type` - specifies an HTTP content type. Defaults to `application/octet-stream`.
We've used the existing `content_type` information that was stored in the database when we saved the image
- c. `:disposition` - specifies whether the file will be shown inline or downloaded.
Valid values are `inline` and `attachment` (default). We want the image displayed in the browser, so we've used `inline`.
- d. `:status` - specifies the status code to send with the response. Defaults to `200 OK`. We don't really need to worry about this today.

Part 2 - Accessing Java Libraries from Rails

The primary advantage of developing with JRuby is that you have access to Java libraries from a Rails application. For example, say you might want to create an image database and a web application that allows processing of the images. You can use Rails to set up the database-backed web application and use the powerful Java 2D™ API for processing the images on the server-side.

This walkthrough shows you how to get started using Java libraries in a Rails application while stepping you through building a simple Rails application that does basic image processing with the Java 2D API.

This application demonstrates the following concepts involved in using Java libraries in a Rails application:

- 1.1.1. Giving your model access to Java libraries.
- 1.1.2. Creating constants to refer to Java classes.
- 1.1.3. Performing file input and output using the `java.io` and `javax.imageio` packages.
- 1.1.4. Assigning Java objects to Ruby objects.
- 1.1.5. Calling Java methods and using variables.
- 1.1.6. Converting arrays from Java language arrays to Ruby arrays.
- 1.1.7. Streaming files to the client.

We'll add another method to the `photos` model and display the edited image in the index file.

1. **Add the following lines to photos model**, right after the class declaration:

```
include Java
BI = java.awt.image.BufferedImage
BA = java.io.ByteArrayInputStream
```

BI and BA are constants for the BufferedImage and ByteArrayInputStream classes so that you can refer to them by the shorter name.

2. **Add this method at the end of the photos controller:**

```
def bw_image
  return nil unless binary_data

  java_bytes = binary_data.to_java_bytes
  ba = BA.new(java_bytes)
  #Read the file into a BufferedImage object and
  # creates a Graphics2D object from it so that you can
  #perform the image processing on it.
  bi = javax.imageio.ImageIO.read(ba)
  w = bi.getWidth()
  h = bi.getHeight()
  bi2 = BI.new(w, h, BI::TYPE_INT_RGB)
  big = bi2.getGraphics()
  big.drawImage(bi, 0, 0, nil)
  bi = bi2
  biFiltered = bi
  # convert the image to grayscale:
  colorSpace =
  java.awt.color.ColorSpace.getInstance(java.awt.color.ColorSpace::CS_GRAY)
  op = java.awt.image.ColorConvertOp.new(colorSpace, nil)
  dest = op.filter(biFiltered, nil)
  big.drawImage(dest, 0, 0, nil);
  #Stream the file to the browser:
  os = java.io.ByteArrayOutputStream.new
  javax.imageio.ImageIO.write(biFiltered, "jpeg", os)
  String.from_java_bytes(os.toByteArray)
end
```

Notice that you don't need to declare the types of the variables, filename or imagefile. JRuby can tell that filename is a String and imagefile is a java.io.File instance because that's what you assigned them to be.

You can call Java methods in pretty much the same way in JRuby as you do in Java code.

You don't have to initialize any variables.

You can just create a variable and assign anything to it. You don't need to give it a type.

The End - Displaying the image inline

1. Let's change the `show.rhtml` view for the `photos` controller, so that instead of displaying a lot of gibberish, we can display the image itself under the "binary" column. If we open the `show.rhtml` view we should see the following at line 18:

```
<%=h @photo.binary_data %>
```

We'll change that to :

```
<b>Image:</b><BR>
```

```
<%= image_tag(formatted_photo_path(@photo, "jpg"), :alt => @photo.filename) %>
```

2. To display the black and white image Add these lines after the last `</p>` tag:

```
<p>
```

```
<b>BW Image:</b><BR>
```

```
<%= image_tag(formatted_photo_path(@photo, "jpg", :bw=>true), :alt => @photo.filename) %>
```

```
</p>
```

Try it! Go to <http://localhost:8080/ImageDBJava/photos/show/1>

(or whatever the image id you want to display) and your image should be displayed.

If it is, then we use the `image_tag` command to render an image. The URL of the image is simply the action that we just created in the "show" method in the `photos` controller to encode and render the image go to:

<http://localhost:8080/photos/1.jpg>

To render the image in black and white go to:

<http://localhost:8080/photos/1.jpg?bw=true>

Where id is the "id" of the image.

3. In the same way, we'll also change the `index.rhtml` to display the image, notice that in this code the image variable is "photo".

Change line 16 to:

```
<td><%= image_tag(formatted_photo_path(photo, "jpg"), :alt => photo.filename) %></td>
```

4. **Change the `index.rhtml` to display the B/W images:**

We'll add a new column to our table to display the processed images.

Add a new line, after line 8:

```
<th>Processed Image</th>
```

5. **Add a new line after line 17** (used to be 16):

```
<td><%= image_tag(formatted_photo_path(photo, "jpg", :bw=>true), :alt => photo.filename) %></td>
```

6. Direct firefox to <http://localhost:8080/ImageDBJava/photos>

Now you can upload images to the database and display them