# Chapter09, Working with Images.

*This is a story rewrite from the original version*

**Original Version**

In the original version of this chapter, we used two services in Alchemy Visual Recognition. We craeted an image classifier and we also created collections of images. Since that time, the (experimental) service to find similar images has been deprecated and is no longer available.

**Current Version**

The focus in this new version will be visual recognition using faces - kind of like an authentication via your face.

***Resources***

- the Watson Visual Recognition service
    - as you have in the previous sessions, log in to bluemix and create the Visual Recognition service.
    - go to the Manage tab and copy your credentials into your env.json file.
    - save your updated env.json file.
- a video of your face - we'll separate that into still images using ffmpeg
- to take a video of your face,
    - hold your phone so that you can see yourself on the screen and start recording
    - move your phone left and right, turning your phone so that it continues to point directly at you
    - come back to center and move up and down, again keeping the phone pointing directly at you
    - finally, move your phone in a circle around your face one or

    two times
    - stop the recording
- transfer the recorded video to your computer.
- create a series of images from your video
    - you'll need the free ffmpeg utility to automatically extract still images from the video. Go to this web page and download a version appropriate for your operating system
    - ffmpeg Download Page
    - follow the instructions on the download page to install ffmpeg
    - copy the movie to the images folder in Chapter09
    - in your terminal window or git-bash shell, go to the images folder
    - execute this command:
        - ffmpeg -i "face.MOV" -vf scale=216:384 -r 1 output_%04d.png
        - replacing faces.MOV with the name of the video file you just created
        - and replacing 216:384 with an appropriate reduced aspect ratio. This ratio is correct for an iPhone7. The smaller number in the ratio needs to be in the 225-250 range. I used 216:384 because it was simple math to divide the original dimensions of my movie (1080 x 1920) by 5. While we don't have to reduce image size, it makes testing and uploads much faster. VR has a 10,000 image, 100Mb limit to a zip file, much larger than we'll need in this tutorial.
    - this will create one image for every second in the video. Mine was 23 seconds long, so 23 images were created.
    - compress the files into a zip file and name it something memorable
- We now need some images to use as negative comparisons in the classifier
    - after doing a short google search for images, I found this

site:
    - http://cswww.essex.ac.uk/mv/allfaces/faces94.html
    - and downloaded the free 'faces94.zip' file
- If you use this zip file, then unzip it and copy the faces94 folder into the images folder in Chapter09
- You should have the following folder structure:

```
Chapter09
    --controller
    --Documentation
    --HTML
    --images
        --faces94
            --male
            --female
            --malestaff
```

- a script has been included which will take the first image from each folder under "male" and each folder under "female" and create two zip files for use by the classifier
    - you type ./neg_mages.sh followed by two parameters, the first is your api-key for visual recognition, the second is the name of the zip file you created earlier in this process.
    - the command is intended to be run from the Chapter09 folder and expects your zip file to be in the Chapter09/images folder
    - for example, my images are stored in a a file called me.zip, so to invoke the command, I would type the following:
    - ./neg_images.sh my-api-key-goes-here me ... note that the 2nd parameter is me and not me.zip
- the script ends by creating a new classifier called

visualAuthenticate.

- after the script has finished processing, it will print out a line like the following with the name of the newly created classifier
- Save this classifier id ==>"visualtest_1933636844"<== to your env.json file
- when you see the 'save this classifier id' message, copy the classifier id into your env.json file. When you're done, your file should look like this:

```
        "visual_recognition": {
       "url": "https://gateway-
a.watsonplatform.net/visual-recognition/api",
       "note": "It may take up to 5 minutes for
this key to become active",
       "api_key": "your-api-key-is-placed-here",
       "classifier_id":
"visualAuthenticate_xxxxxxxx",
     },
```

- the neg_images.sh script will also print out a copy of a curl command you can use to display the status of your newly created classifier

```
You can check the status of your classifier by
executing the following command, you're looking
for a 'ready' state'
curl -X GET "https://gateway-
a.watsonplatform.net/visual-
recognition/api/v3/classifiers/"visualAuthentica
te_xxxxxxxxxxx"?api_key=your-api-key-will-be-
automatically-inserted-here&version=2016-05-20"
```

***Set up is complete, now, on to coding***

We worked with three files in the original version:

- controller/restapi/features/images.js
- HTML/CSS/pageStyles.css
- HTML/js/z2c-image.js

Follow the original tutorial for changes to pageStyles.css, we will use that file in its original format.

# images.js

We no longer need the find routine, as that service is no longer available in Visual Recognition. We will execute two requests in the classification routine to extract both face information and ckassification information.

## classification

This function receives an image from the browser and then performs two transforms on that image. First, it looks to see if the image has a face in it and then classifies the image. We could, if we chose, stop the process after the face identification if it comes back with no identified faces. In this case we would need to update this function to execute a res.send() with an appropriately packaged JSON object. The display routine on the browser side would also have to be updated if the structure of the returned object changed.

```
/**
 * classify saves the provided image and then sends
that image to watson for classification
 * @param {object} req - nodejs object with the
request information
 * req.body holds post parameters
 * @param {object} res - nodejs response object
 * @param {object} next - nodejs next object - used
```

```
if this routine does not provide a response
 */
exports.classify= function(req, res, next){

  req.pipe(req.busboy);
  req.busboy.on('file', function(fieldname, file,
filename)
  {
    // not all file systems are friendly to names
with spaces in them.
    // if this name has spaces in it, replace them
with an underscore.
    var fileName = filename.replace(/ /g,"_");
    var newFile =
path.join(path.dirname(require.main.filename),'images'
,fileName);
    var fd = fs.openSync(newFile, 'w');
    var fstream = fs.createWriteStream(newFile, {fd:
fd});
    var _res = {};
    file.pipe(fstream);
    // now that we have the image stored on the
server, send it to watson
    fstream.on('close', function ()
    {
      var params = {images_file:
fs.createReadStream(newFile), classifier_ids:
[vr_classifier] };
      var visual_recognition =
watson.visual_recognition(
      {
        api_key: apiKey,
        version: 'v3', version_date: '2016-05-20'
      });
      visual_recognition.detectFaces(params,
function(err, faces)
      {
        if (err) {console.log(err);
res.send({'results': 'failed', 'where':
```

```
  'detectFaces', 'error': err});}
          else
          {
            console.log('detecFaces successful:
  '+JSON.stringify(faces)); _res.faces = faces;
            var params = {images_file:
  fs.createReadStream(newFile), classifier_ids:
  [vr_classifier] };
            visual_recognition.classify(params,
  function(err, classify_results)
            {
              if (err) {console.log(err);
  res.send({'results': 'failed', 'where': 'classify',
  'error': err});}
              else
              {
                console.log('classify successful:
  '+JSON.stringify(classify_results)); _res.classify =
  classify_results;
                res.send({'results': 'success', 'data':
  _res});
              }
            });
          }
        });
      });
    });
  }
```

## z2c-images.js

We have less work to do in z2c-image.js than before.

functions: ***initiateVR*** is called after authentication has succeeded. This function was changed from the original, specifically to support face identification, so the image is now displayed into an HTML canvas

element instead of an img element. You'll see those changes near the end of the function.

```
/**
* initialize the visual recognition page.
*/
function initiateVR()
{
 c_res = $("#classifyResults");
 _url = $("#imageURL");
 _image = $("#image");
 console.log("initiateVR");
 // using the checkImageDroppable function in z2c-
utilities.js, ensure that the browser supports drag
and drop operation
 b_Droppable = checkImageDroppable();
 if (b_Droppable)
 {  console.log("browser supports drag and drop:
"+b_Droppable);
     // initialize variables
     droppedFiles= false;
     $form = $('.image')
     var $input    = $form.find('.imageReceiver');
     var droppedFiles = false;
     // the image receiver is inside an html <form>
object
     $form.on('submit', function(e) {
       e.preventDefault();
       // the submit button was clicked, but no file
was dropped or selected
       if (droppedFiles == false) {c_res.append("
<h3>Error: please select a file, first."); return;}
       else
       // files have a max size of 2Mb
         {if (droppedFiles[0].size > maxSize)
{c_res.append("<h3>Error: File size too large. Image
must be smaller than 2MB."); return;}
         else
```

```
          // only jpeg and png files are supported
.... well, it works with gif, too, just not as well
           {if ((droppedFiles[0].type !=
"image/jpeg") && (droppedFiles[0].type !=
"image/png")) {c_res.append("<h3>Error: Only jpeg and
png files are supported</h3>"); return;}
              else
              {
                // everything is good. let's proceed
                // display a busy icon
              c_res.empty(); c_res.append("<center>
<img src='icons/loading.gif' /></center>");
              // get the image data
              var ajaxData = new FormData();
               console.log("processing files:
$input.attr('name'): "+ $input.attr('name'));
               ajaxData.append( droppedFiles[0].name,
droppedFiles[0] );
                // ajax is asynchronous javascript
execution. Send the request to the server
                // let the browser do other things
                // then respond when the server
returns
               $.ajax({
                 url: $form.attr('action'),
                 type: $form.attr('method'),
                 data: ajaxData,
                 dataType: 'json',
                 cache: false,
                 contentType: false,
                 processData: false,
                 // wait until everything comes back,
then display the classification results
                 complete: function(data) {
displayImageClassificationResults(c_res,
data.responseText)},
                 success: function(data) { },
                 // oops, there was an error, display
the error message
```

```
                error: function(err) {
console.log("error: "+err);
displayObjectValues("error:", err); }
              });
            }
          }
        }
      });
      // don't do any default processing on drag and
drop
      _image.on('drag dragstart dragend dragover
dragenter dragleave drop',
      function(e) { e.preventDefault();
e.stopPropagation(); });
      // change how the drag target looks when an
image has been dragged over the drop area
      _image.on('dragover dragenter', function() {
_image.addClass('dragover'); });
      // remove drag target highlighting when the
mouse leaves the drag area
      _image.on('dragleave dragend drop', function()
{ _image.removeClass('dragover'); });
      // do the following when the image is dragged
in and dropped
      _image.on('drop', function(e)
      {
        droppedFiles =
e.originalEvent.dataTransfer.files;
        console.log("dropped file name:
"+droppedFiles[0].name);
        // build a table to display image
information
        var fileSpecs = "<table width='90%'><tr>
<td>File Name</td><td>"+droppedFiles[0].name+"</td>
</tr>";
        // check image size
        var tooBig = (droppedFiles[0].size >
maxSize) ? " ... File size too large" : "";
        // check image type
```

```
            var imageType = ((droppedFiles[0].type ==
"image/jpeg") || (droppedFiles[0].type ==
"image/png")) ? "" : " ... Only jpeg and png files
are supported";
            fileSpecs += "<tr><td>File Size</td>
<td>"+droppedFiles[0].size+tooBig+"</td></tr>";
            fileSpecs += "<tr><td>File Type</td>
<td>"+droppedFiles[0].type+imageType+"</td></tr>
</table>";
            // clear the target
            c_res.empty();
            // display the table
            c_res.append(fileSpecs);
            // load the image to get the original size
of the inbound image.
            // we need this information to correctly
draw a box around the face later on.
                  var reader = new FileReader();
                  reader.onload = function(e) {
              var __image = '<img id="fileToLoad" src="'
+ e.target.result + '", height=200 />'
              _image.empty();  _image.append(__image);
_image.hide();
              var _img = $("#fileToLoad");
              window.setTimeout(function()
              {
                // the display window is 200 pixels
high. calculate the multiplication factor to fit this
window and save it for later use
                _factor = 200/_img[0].naturalHeight;
                // calculate the target width
                var _width =
_factor*_img[0].naturalWidth;
                var _height = 200;
                // create a drawing canvas and center it
in the imageReceiver div
                __image = '<center><canvas
id="fileCanvas" width="'+_width+'"
height="'+_height+'"></canvas></center>'
```

```
                // empty the div of the image we just
loaded and append this canvas to the now empty div
                _image.empty(); _image.append(__image);
                // get the drawing context for the
canvas
                var ctx = $("#fileCanvas")
[0].getContext("2d");
                // create a drawable image
                var imageObj = new Image();
                // link the source to the image dropped
on the imageReceiver div
                imageObj.src = e.target.result;
                // when the image has loaded into
memory, draw it
                imageObj.onload = function () {
                  ctx.drawImage(imageObj, 0, 0, _width,
_height);
                }
                _image.show();
                }, 100);
            }

reader.readAsDataURL(droppedFiles[0]);
           });
           // update the image area css
         _image.addClass("dd_upload");
 }
 // sorry, but your browser does not support drag and
drop. Time to finally do that upgrade?
 else {  console.log("browser does not support drag
and drop: "+b_Droppable); }
}
```

***displayImageClassificationResults*** has a minor change to it. The original code follows, near the end of it you'll see the following line of code:

```
      _tbl += '<tr><td class="col-md-6'+_disabled+'">
<a
onclick="findInCollection(\''+_image+'\',\''+_obj[_id
x].class+'\')" class="btn btn-primary,
showfocus">'+_obj[_idx].class+'</a></td>
<td>'+_obj[_idx].score+'</td></tr>"';
```

we're going to remove the hyperlink <a> processing so that we just display the class name, so the line changes to the following:

```
      _tbl += '<tr><td class="col-md-6">
<b>'+_obj[_idx].class+'</b></td>
<td>'+_obj[_idx].score+'</td></tr>"';
```

===> Complete code <===

```
/**
 * display the results of the image classification
process
 * @param {String} _target - the html element to
receive the results
 * @param {String} _data - the image information to
display
 */
function displayImageClassificationResults(_target,
_data)
{
    // empty the html target area
    _target.empty();
    // turn the returned string back into a JSON
object
    var imageResults = JSON.parse(_data);
    // check to see if the request for face detection
and clssification were both successful
```

```
    if (imageResults.results !== 'success')
    {
      _target.append('Visual Recognition failed at:
'+imageResults.where+' with error:
<br/>'+imageResults.err.message);
      return;
    }
    // check to make sure that there was at least one
face in the provided image
    if
(imageResults.data.faces.images[0].faces.length ===
0)
    {
      _target.append('There is no face in the
provided image:
'+imageResults.data.faces.images[0].image);
      return;
    }
    else
    {
      // get the rectangle of the first face found in
the image
      var _imgRect =
imageResults.data.faces.images[0].faces[0].face_locat
ion;
      // the top value provided is consistently about
1.3 times the actual vertical offset of the face
      // the following correction factor is to
address that bug
      var vert_correction = 0.3;
      // get the drawing context of the canvas
      var ctx = $("#fileCanvas")[0].getContext("2d");
      // create a rectangle to fit around the face
      ctx.rect(_factor*_imgRect.left,
vert_correction*_factor*_imgRect.top,
_factor*_imgRect.width, _factor*_imgRect.height);
      // set the line width to 6 pixels
      ctx.lineWidth="6";
      // set the line color to blue
```

```
      ctx.strokeStyle="blue";
      // draw the rectangle.
      ctx.stroke();

    }

    // create a display table
    var _tbl = "<table width=90%><tr><th>Image
Class</th><th>Probability</th><tr>";

    var _image =
imageResults.data.classify.images[0].image;
    // iterate through the classification results,
displaying one table row for each result row
    if
(imageResults.data.classify.images[0].classifiers.len
gth === 0)
    { _tbl += "<tr><td>No Results with higher than
50% probability</td></tr>"}
    else
    {
      for (each in
imageResults.data.classify.images[0].classifiers[0].c
lasses)
      {
        (function (_idx, _obj) {
        _tbl += '<tr class="showFocus"><td
class="col-md-6"><b>'+_obj[_idx].class+'</b></td>
<td>'+_obj[_idx].score+'</td></tr>';
        })(each,
imageResults.data.classify.images[0].classifiers[0].c
lasses)
      }
    }
    // close the table
    _tbl += "</table>";
    // and append it to the target.
    console.log(_tbl);
    _target.append(_tbl);
```

```
}
```

### *functions no longer needed*

The two functions findInCollection() and displayCollectionResults() are no longer required and are now removed from the tutorial.

### *Using your code*

Now that you've completed your code updates, what will this program do?

- start your app by typing npm start
- direct your browser to localhost:xxxx where xxxx is the port number displayed when you started the app
- log in
- drag an image onto the image window
    - you'll see the image displayed in that window
- click on the classify an image button
- review the results
    - you'll see either a message telling you that no results were returned. If that's the case, then the supplied image had a less than 50% match on any of the supplied classifier zip files.
    - you'll see a probability listing with one or more rows. There are three possible classes which can be displayed, if you used the supplied script to build the classifier
        - male – not you, but probably a male
        - female – not you, but probably a female
        - you – this will display the name of your zip file (hence the earlier recommendation to make the name memorable)
        - The application will also draw a box around the first (highest probability) face found in the provided image.

- you can find test images simply by doing a web search for face images, which allows you to use different images in your demonstration.