Hochschule
Flensburg
University of
Applied Sciences

Angewandte Informatik

# Webprogrammierung

## Hausarbeit WS19

Von

*Duy Khoi Nguyen*

*Mtrn. 630305*

Hinweis:
Es müssen zwei Datenbanken importiert werden

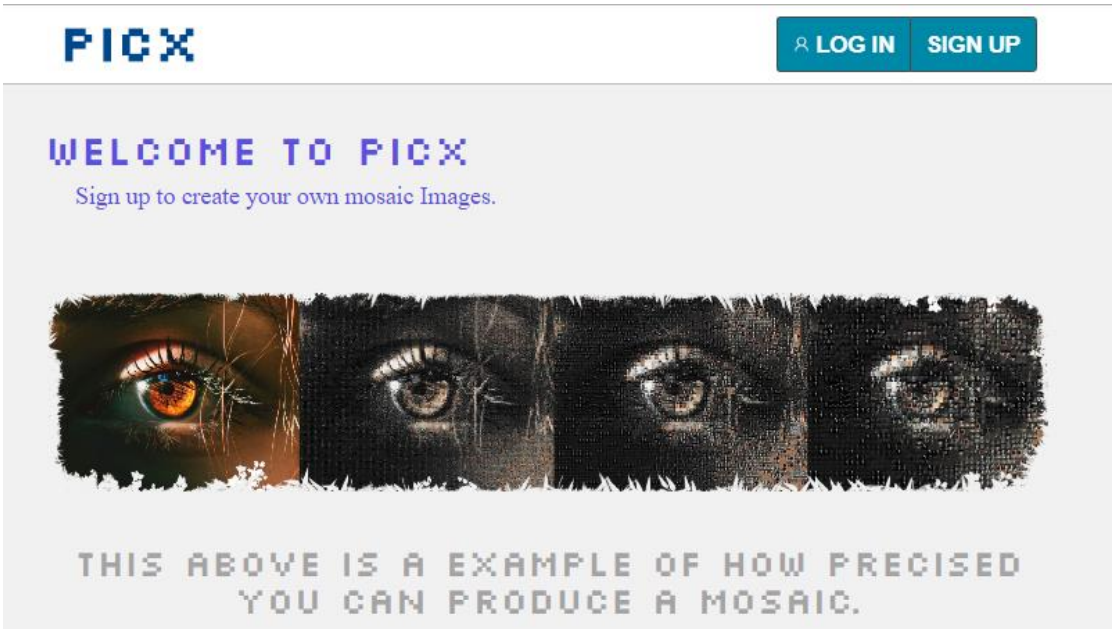
Eine DB für die Nutzerdaten:
***dump\HA19DB_Duy_Khoi_Nguyen_630305_Users***
Eine DB für die Bilder:
***dump\HA19DB_Duy_Khoi_Nguyen_630305_Bilder***

PICX anwendung aufrufbar per:
**http://localhost:4242/picx**

Benutzername: robert
Kennwort: silvers

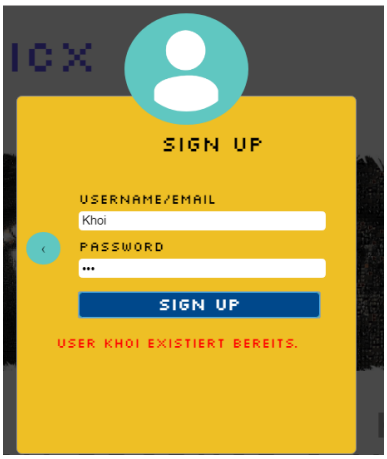
# Inhalt

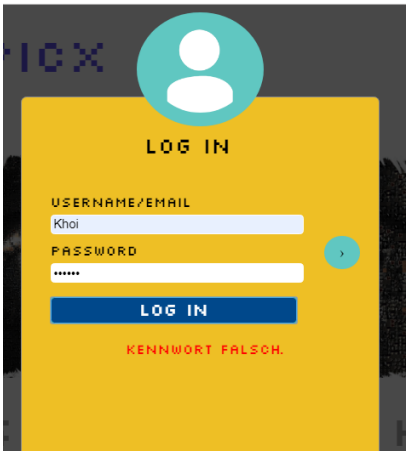
# 1. Das UI-Konzept und das Layout der Anwendung




Anfangs gelangt man auf die Startseite von PICX, hier bekommt man schon einen kleinen „Vorgeschmack“, wie die Mosaikumwandlung aussehen kann.

Oben rechts kann man sich Registrieren und Anmelden. Wenn man auf einen Button klickt erscheint dann auch ein Login/Register Popup auch Modale genannt.

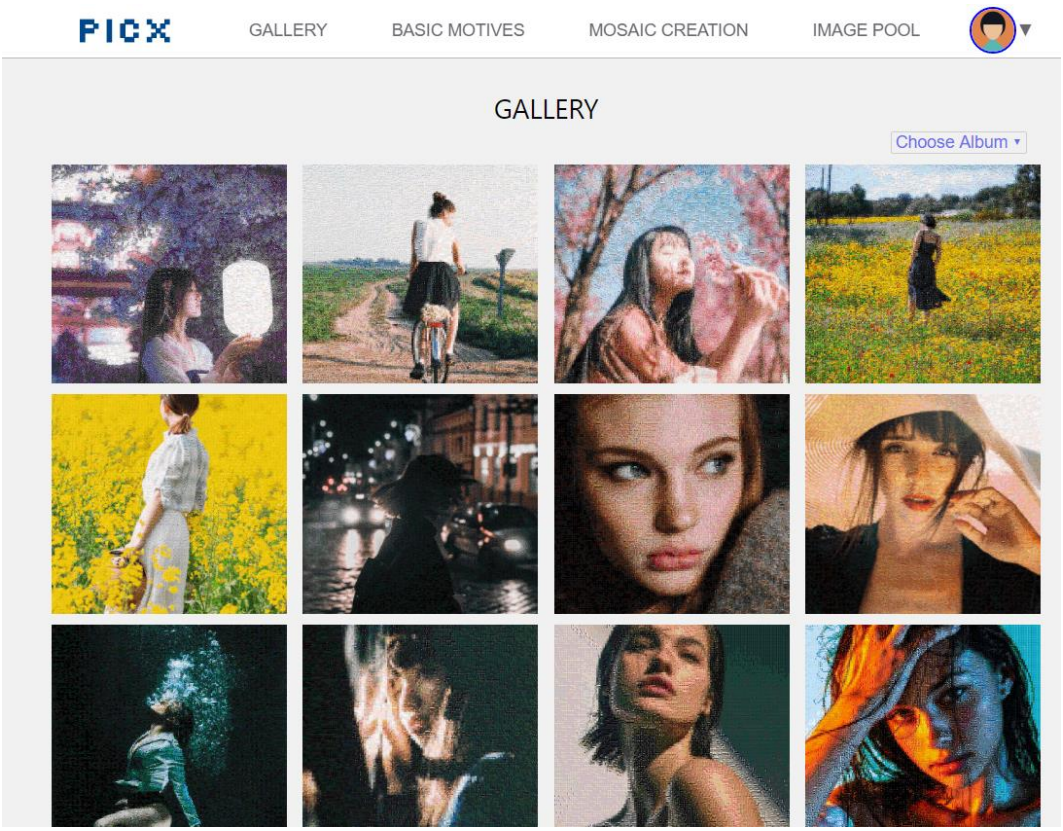



Beim Registrieren/Login wird auch dementsprechend Feedback abgegeben, ob:

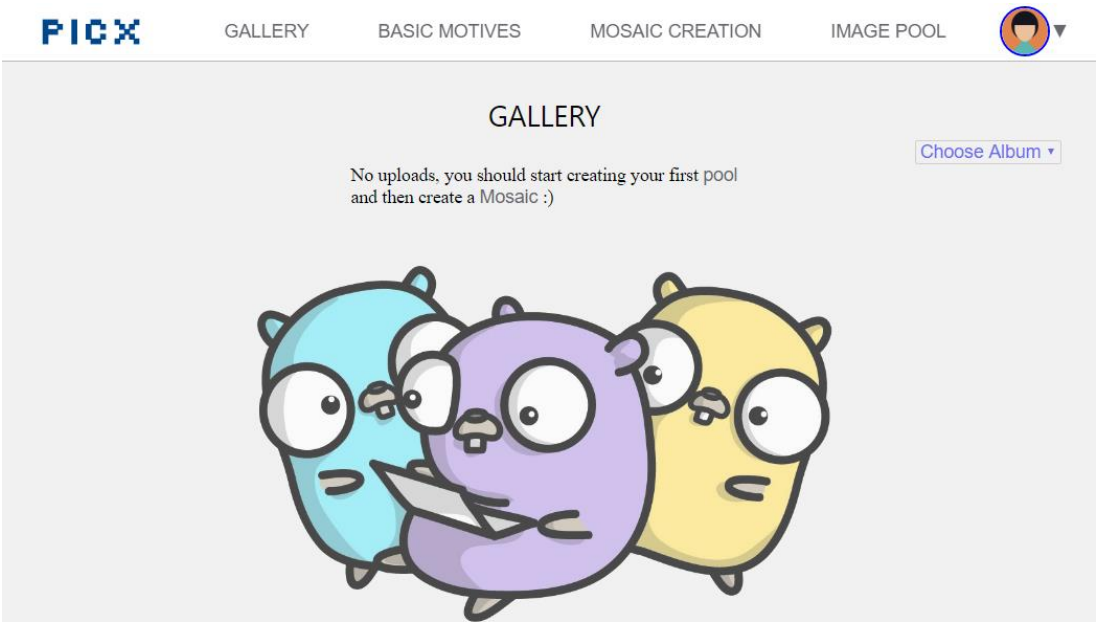
- es einen Nutzer bereits gibt
- die Registrierung erfolgreich war
- das Passwort bei der Anmeldung falsch war
- das Passwort zu kurz ist
- es den Nutzer überhaupt gibt

Bei der erfolgreichen Anmeldung wird man auf seiner Gallery (Mosaik Seite) weitergeleitet:




Hier seht man die Mosaikbilder, die man bereits erstellt hat. Mehr zur Gallery kommt später. (Hier handelt es sich um Mosaikbilder, wo kleine Kacheln benutzt wurden)

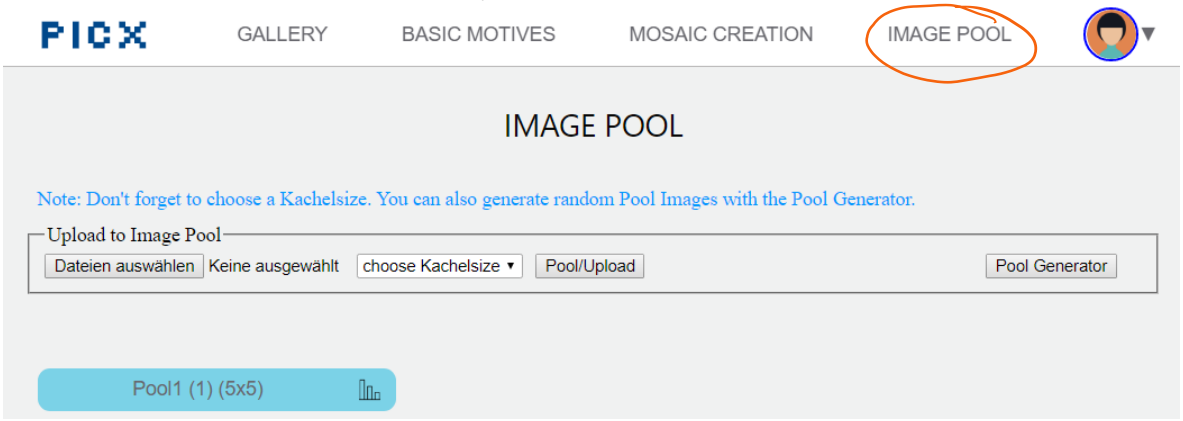
Wenn man ein neuer Nutzer, ohne Bilder ist, wird einem in der Gallery ein Gopher-Bildchen angezeigt mit Hinweis, dass man mal doch ein Mosaik erstellen sollte:



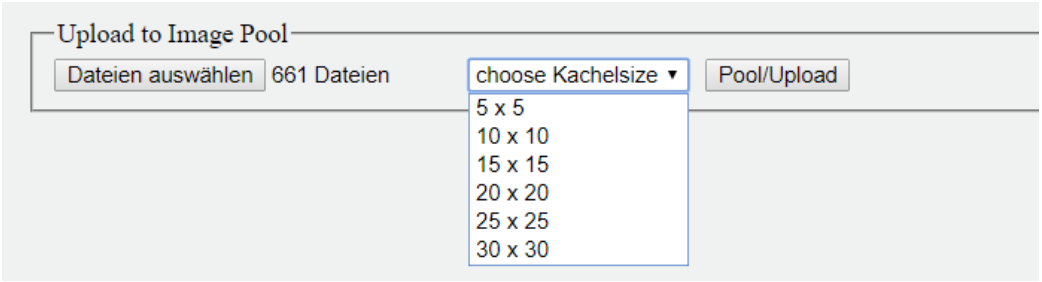
Wie man an der Navigationsbar sieht, ist PICX in vier Seiten gegliedert
(GALLERY, BASIC MOTIVES, MOSAIC CREATION und IMAGE POOL)


## 1.1 Poolseite, Poole und Poolgenerator

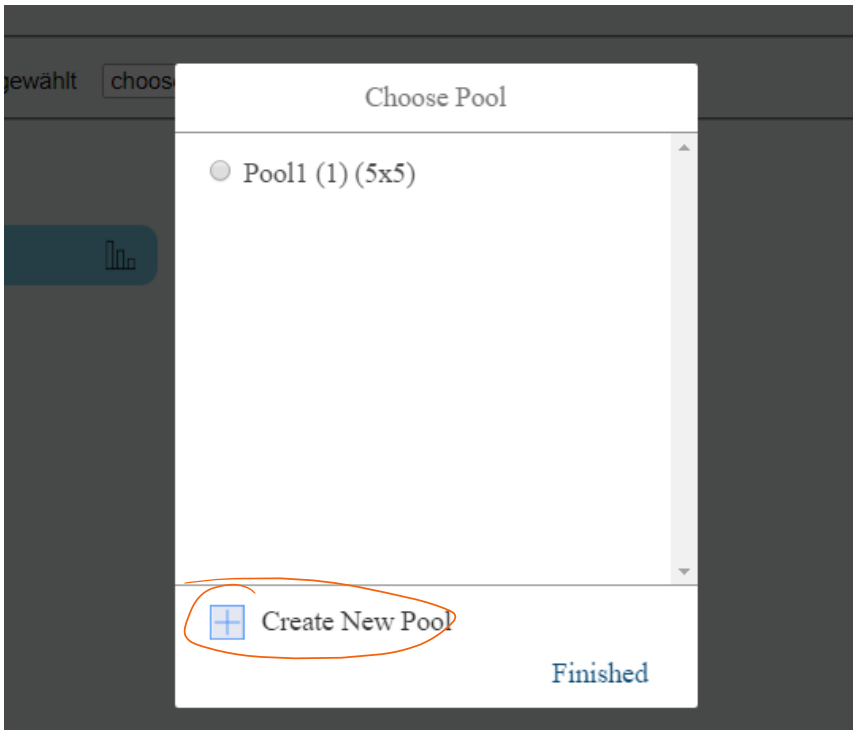
Nun kommen wir zur IMAGE POOL Seite, hier kann man Poole erstellen und Kacheln hochladen:



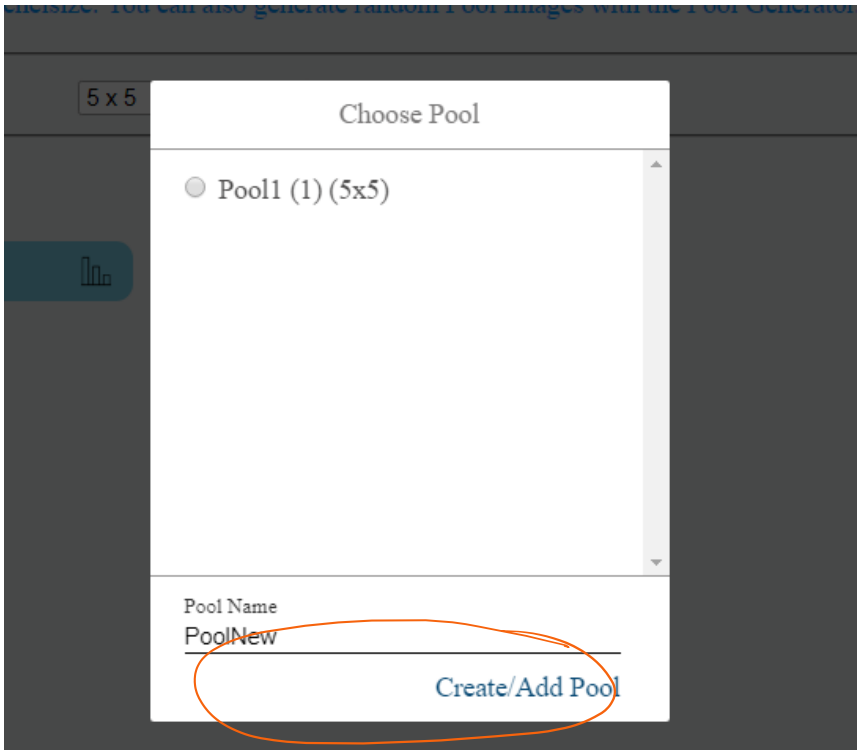

Nun wollen wir mal Kachel hochladen, hierzu laden wir unter „Dateien auswählen“ mehrere Bilder in unsere Form und wählen unter dem select field wo „choose Kachelsize“ steht eine Kachelgröße aus.



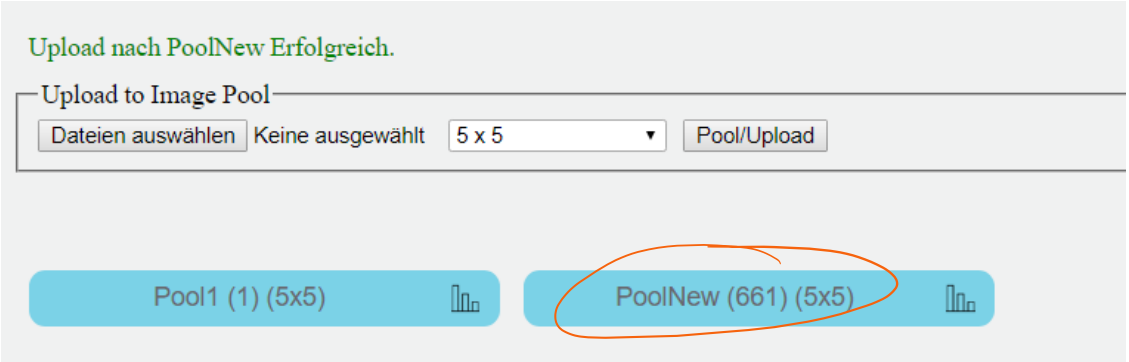
Klickt man dann auf den Button „Pool/Upload“, erscheint diese Modal.
Hier können wir auswählen ob wir in ein existierenden Pool die Bilder Hochladen.




Oder unten auf das Plus mit „Create New Pool“ einen neuen Pool erstellen und dort unsere Bilder Hochladen. Wir erstellen einen neuen Pool mit den namen „PoolNew“.

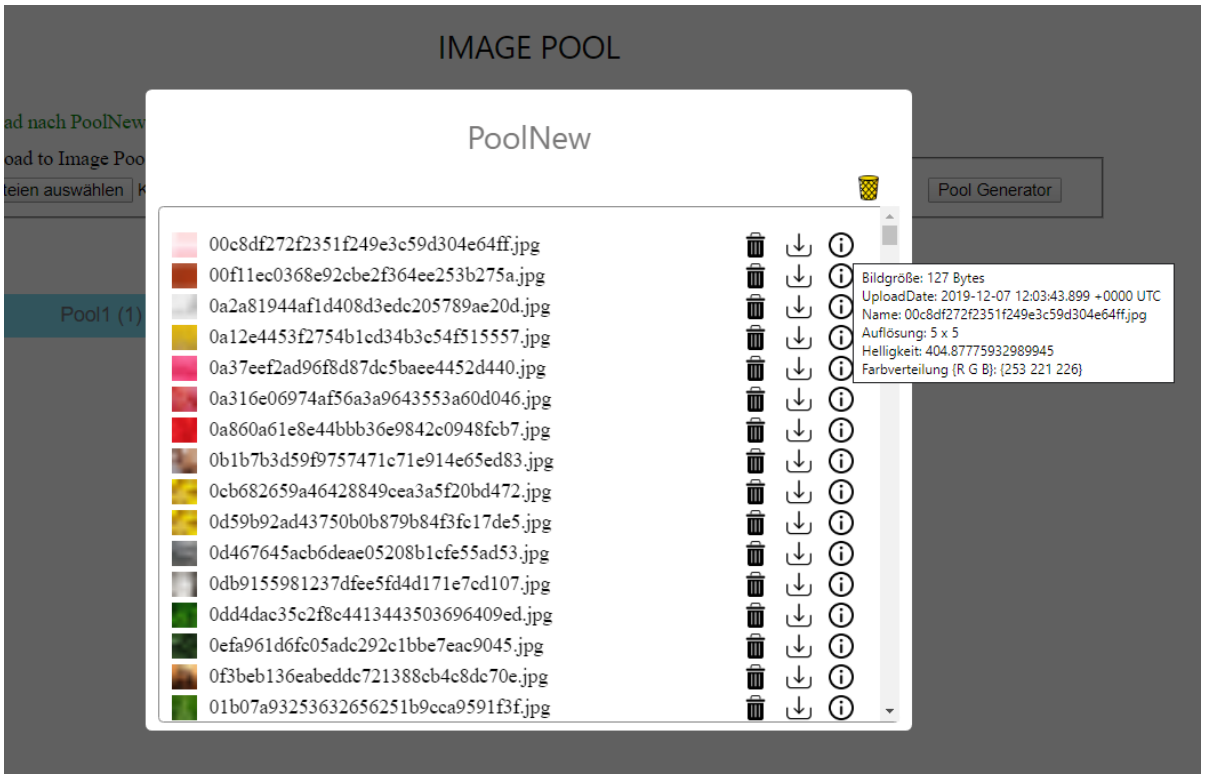


Nun habe wir einen Pool namens „PoolNew“ mit 661 Kacheln welche 5x5 Pixel groß sind.
Pools werden hier als blaue Kästchen dargestellt.

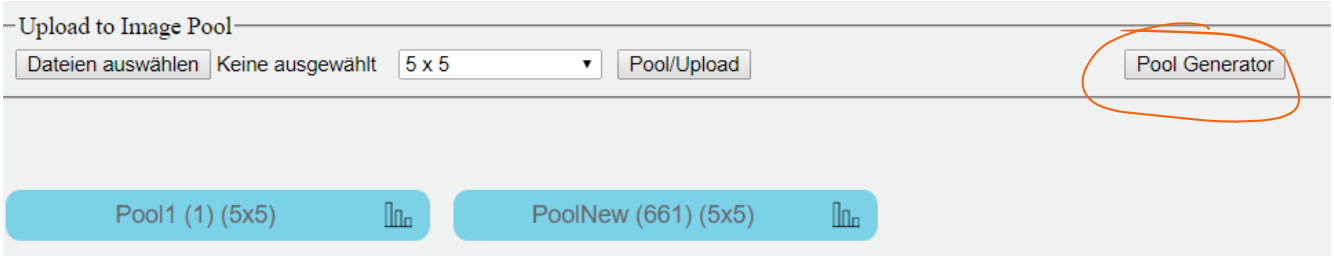



Man kann die Poole auch anklicken/auswählen und erhält dann eine genauere Ansicht.
Tut man dies, wird in einem Modal, die Bilddarstellung der Kacheln, Dateinamen dargestellt:

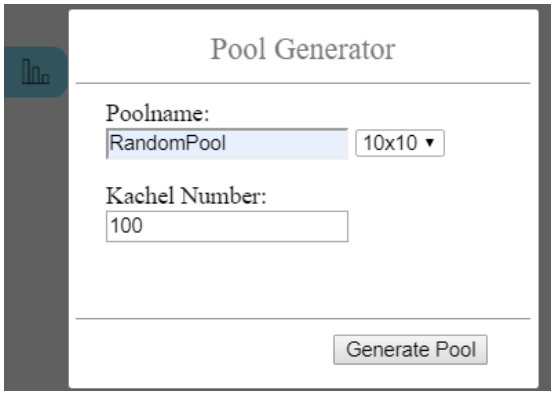
Man kann einzelne Kachel löschen (dunkler Mülleimer), Kacheln downloaden und Metainformation lesen, wenn man über das Info Icon seine Maus hält. Mit dem gelben Mülleimer wird der ganze Pool gelöscht.



Ebenfalls gibt es rechts im Fieldset einen Pool Generator

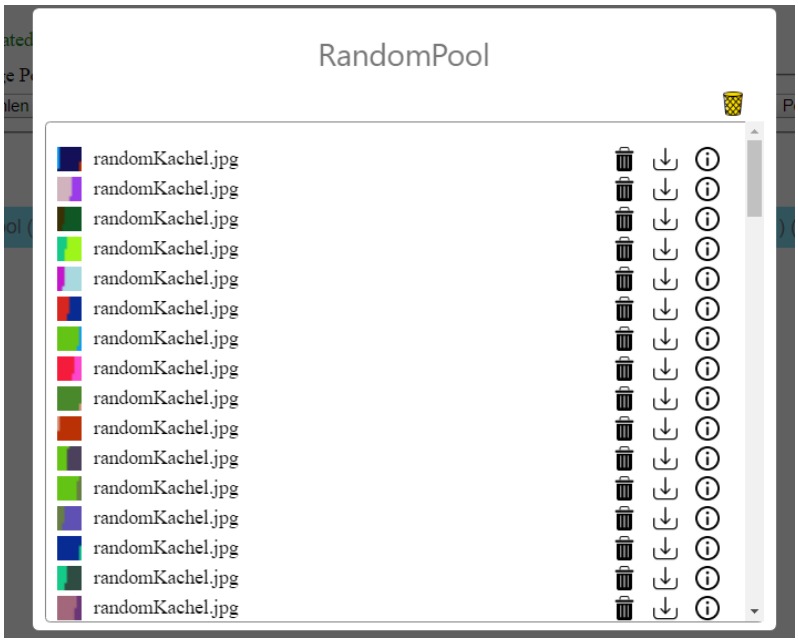



Beim Anklicken wird dann der Pool Generator angezeigt, hier kann man zufällige Kacheln erstellen und in einen Pool speichern, wir generieren uns einen Pool namens „RandomPool“ mit Kachelgröße 10x10px und mit der Kachelanzahl 100




Ein generierter Pool unterscheidet sich in der Funktion nicht von einem normalen Pool

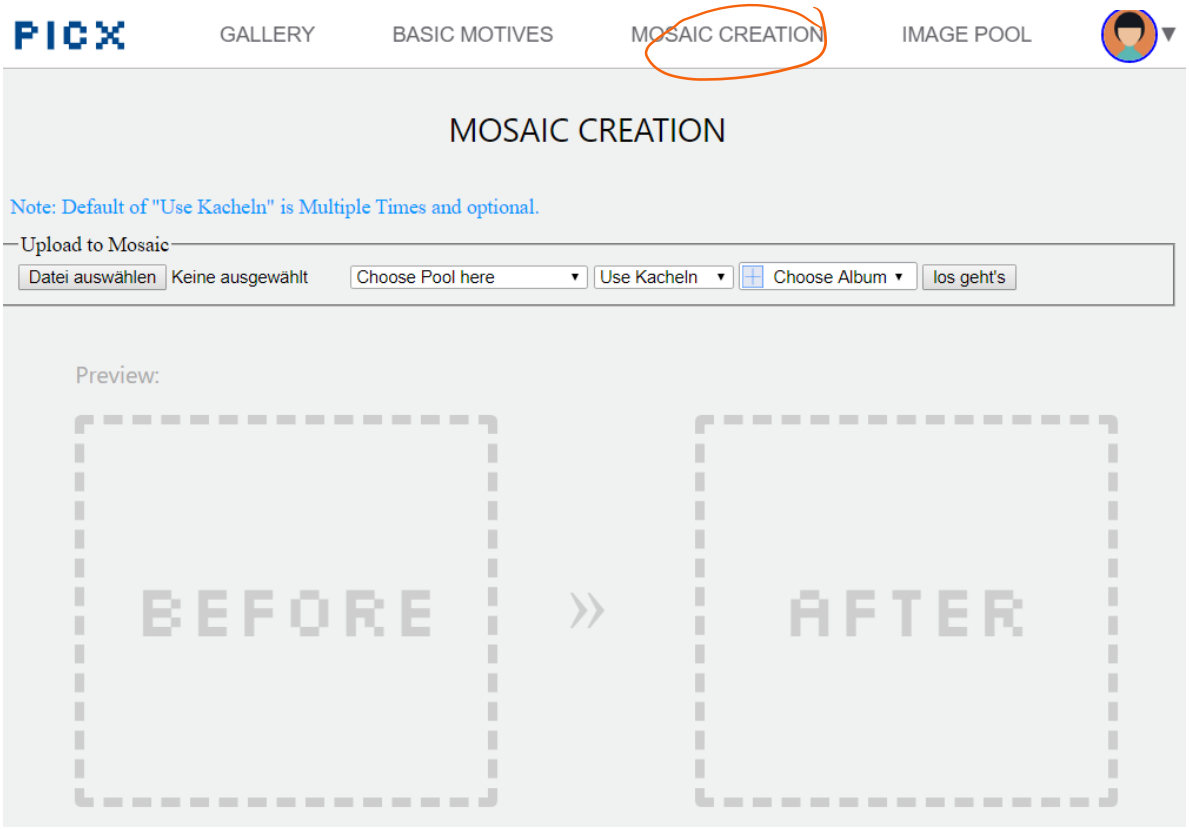

Hier ist eine Einsicht in den generierten Pool:



Ein weiteres Element des Pools ist das Diagramm Icon rechts. Beim Draufklicken wird einem die durchschnittlichen RGB und Helligkeitswerte alle Kacheln als kleines Diagramm angezeigt.

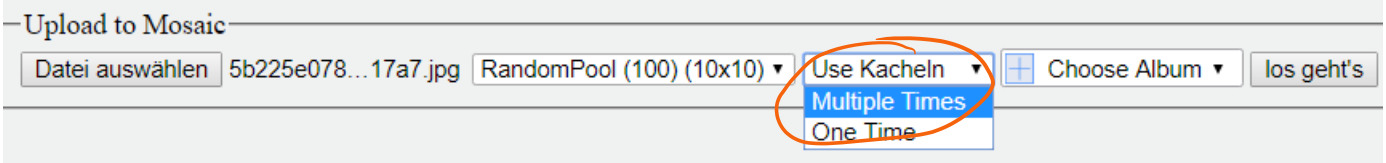

## 1.2 MOSAIC CREATION

Nun kommen wir zur Seite „Mosaic creation“, hier werden die Mosaik Bilder erstellt.

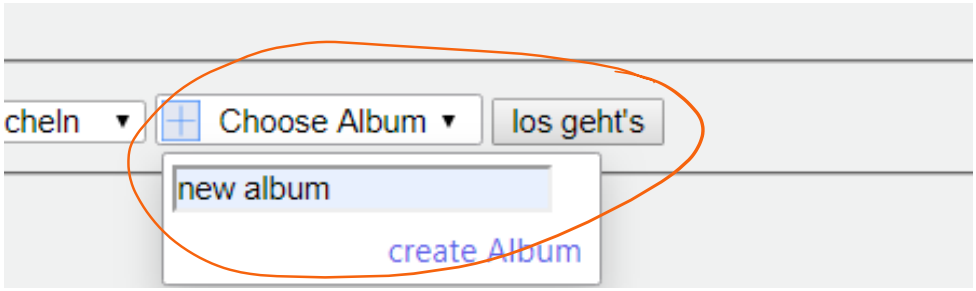



Wie man seht kann man hier ein Bild uploaden, welches das Basis Motiv ist. Wir fügen mal ein Bild ein.

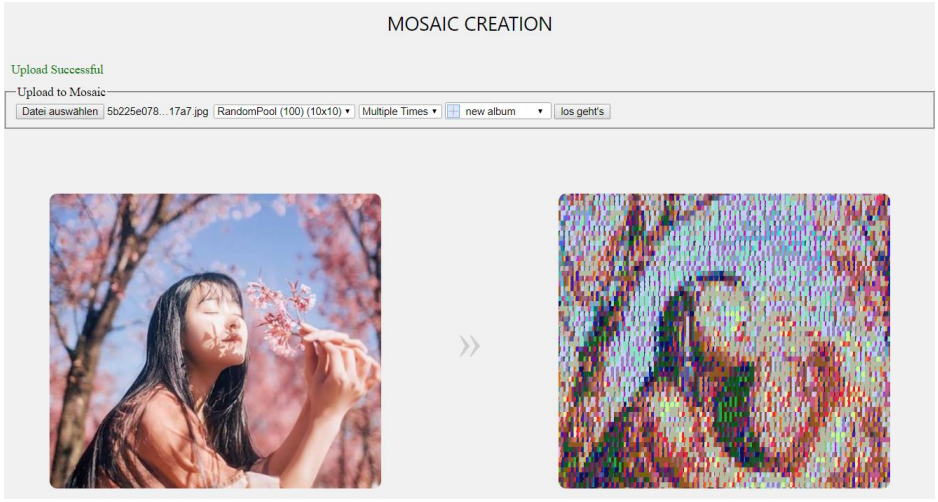

Nun wählen wir unter „Choose Pool here“ einen unserer erstellten Pools aus.
Unter „Use Kacheln“ wird angegeben ob man Kackeln mehrmals verwendet oder nur einmal.



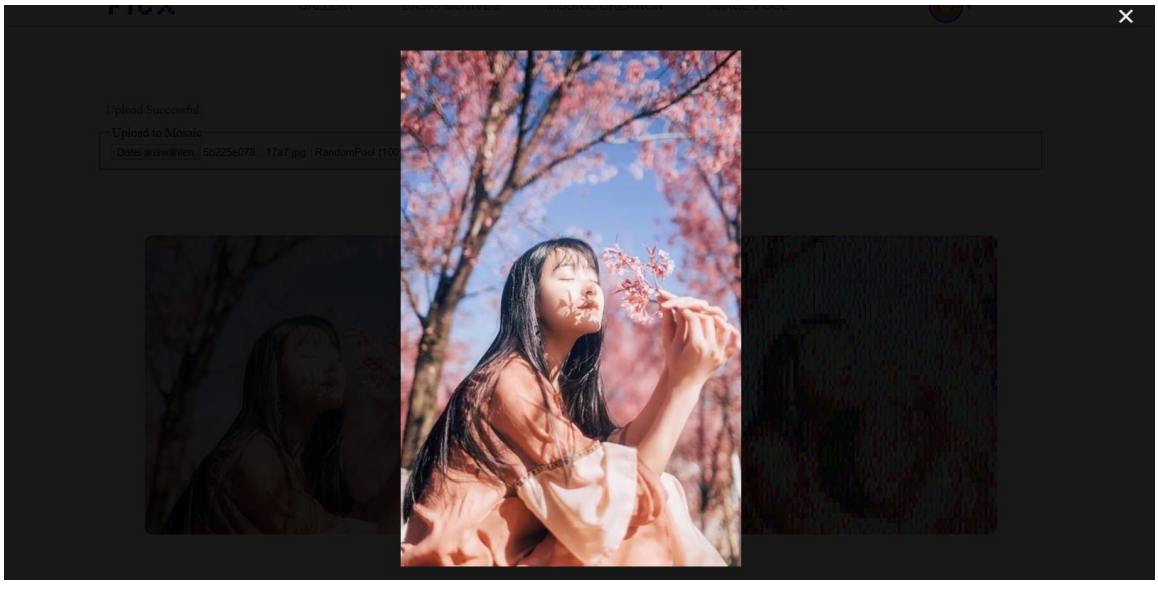
Unter „Choose Album“ wählt man aus in welchem Album oder Sammlung man seine Bilder speichert. Wir haben jedoch noch keine Alben und müssen auf das Pluszeichen nebenan drücken, um ein Album zu erstellen. Wir erstellen uns ein Album namens „new album“ und drücken dann auf „los geht’s“ um unser Mosaik zu erstellen.

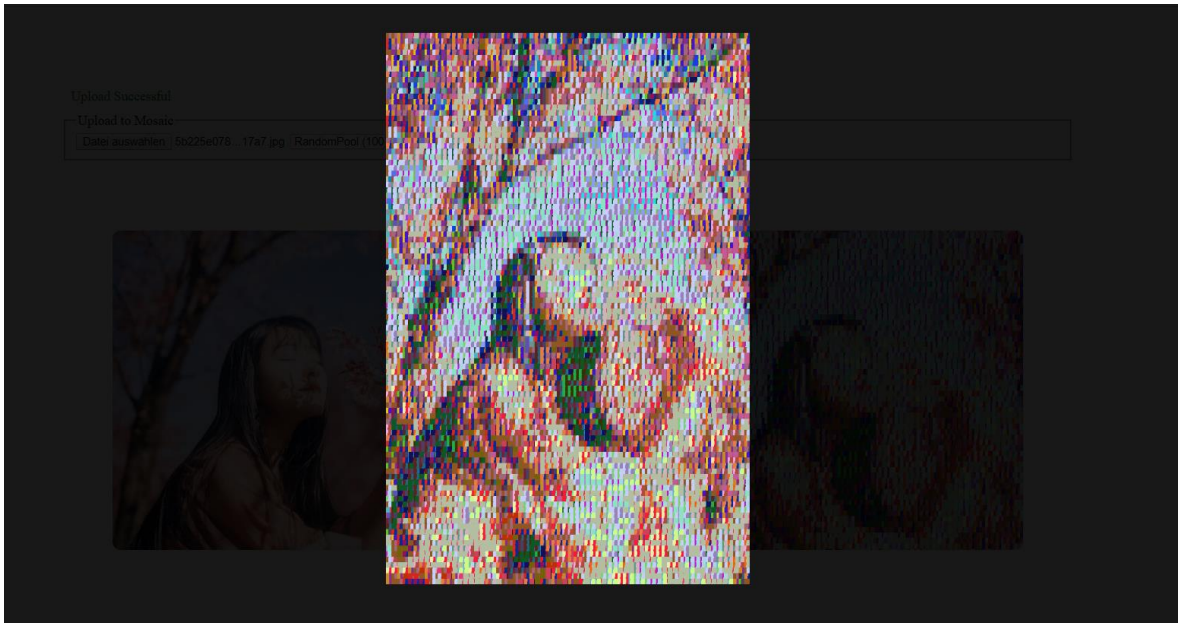



Nun erhalten wir hier einen Vergleich zwischen Basismotiv und erstellten Mosaik

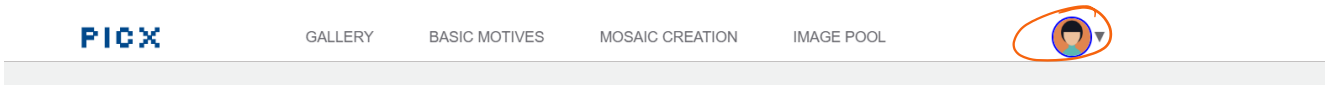



Will man die Bilder genauer in größer ansehen, kann man einfach auf diese klicken und dieses wird voll angezeigt, gilt für Mosaik und Basis Motiv:

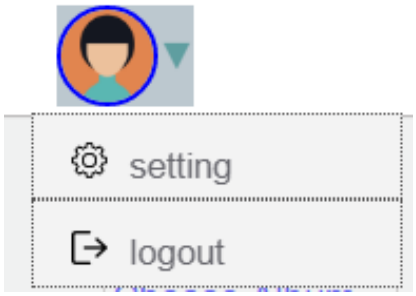




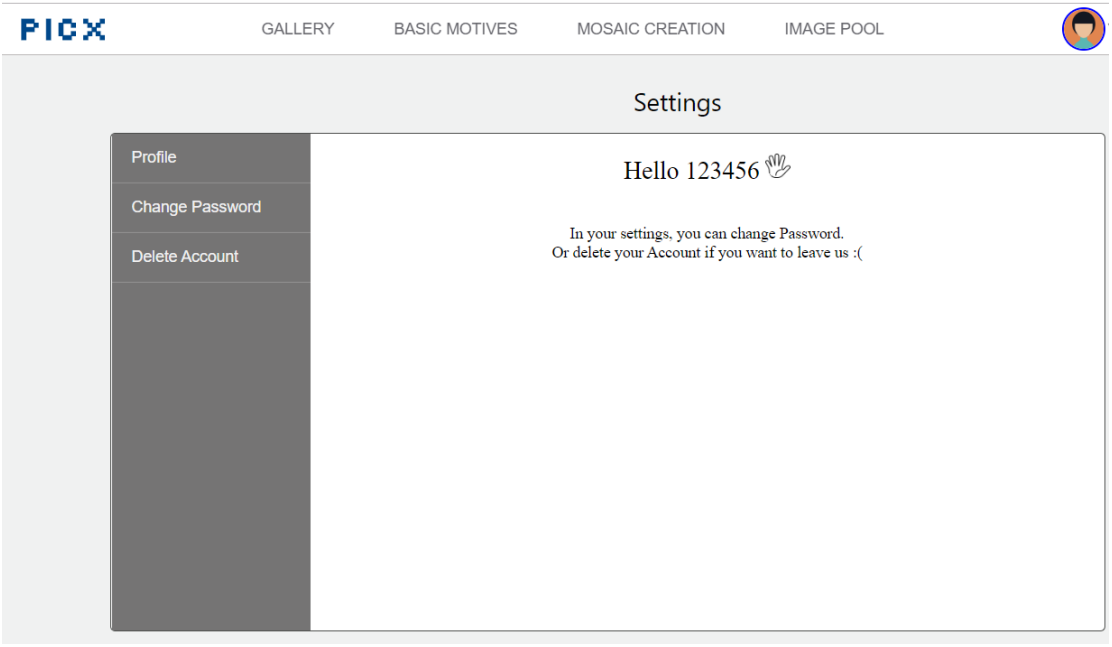

Eine weitere zusätzliche Seite ist Setting (wenn man über das User Icon oben rechts hovert):



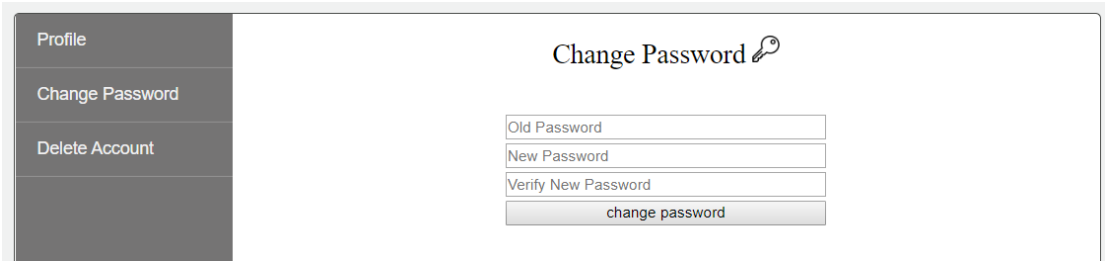

Dann erscheint ein Dropdown mit Logout und Setting:



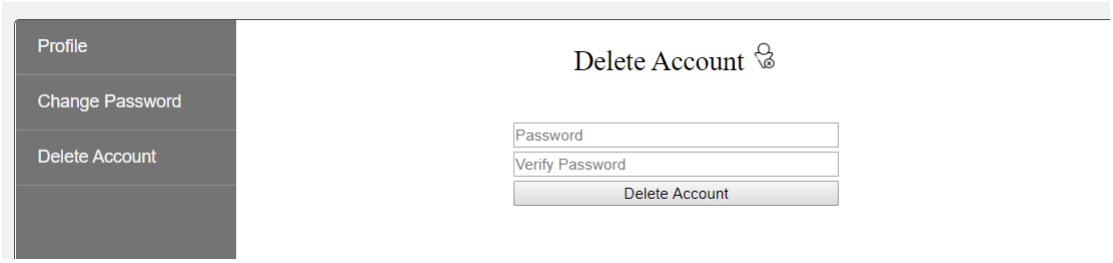
In den Settings kann man sein Passwort ändern oder den Account löschen, beim Löschen des Accounts werden auch alle Alben, Pools, Mosaikbilder und Basismotive des Nutzers gelöscht.




Passwort ändern (Nutzerfeedback ob z.B. falsches Passwort eingegeben, ist natürlich mit drin):

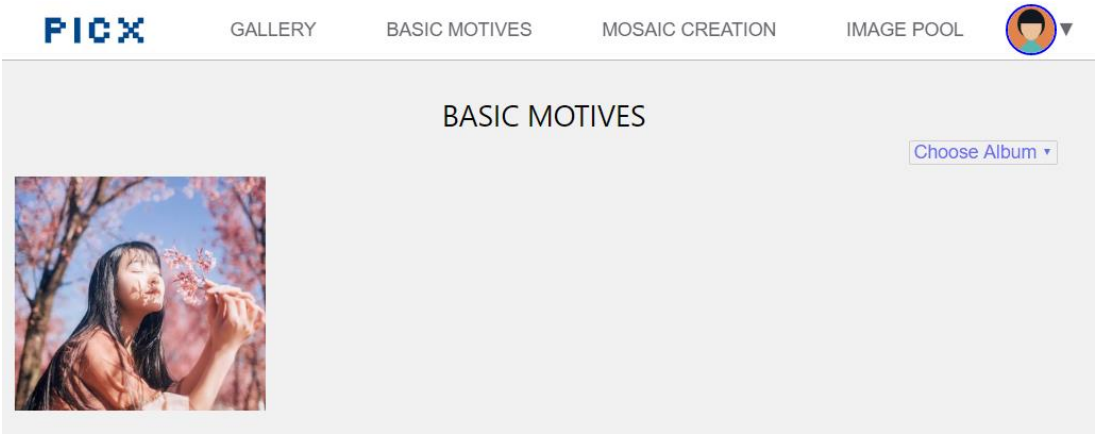



Account löschen (Nutzerfeedback ob z.B. falsches Passwort eingegeben, ist natürlich mit drin):



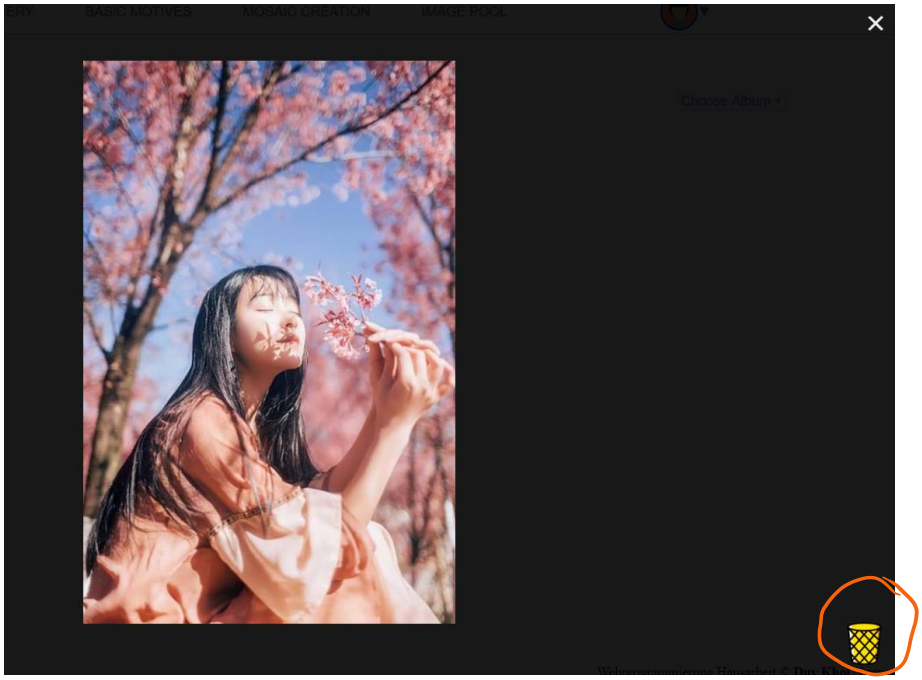
## 1.3 Basic Motives und Gallery (Mosaic Motives)

Basic Motives enthält die hochgeladen Basis Motive

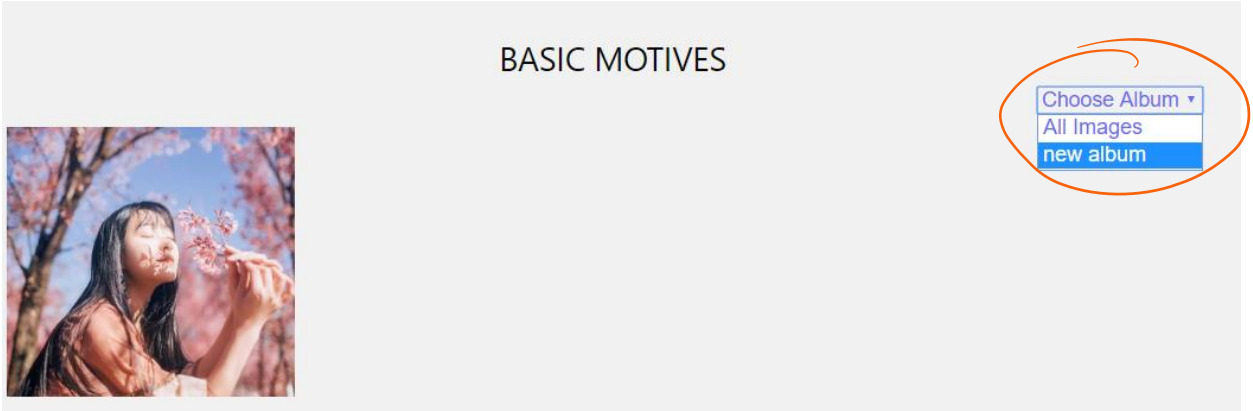



Wenn man über Bilder drüber hovert, erscheint ein Download Icon und ein Info Icon.
Mit diesen kann man das Bild downloaden oder Informationen rauslesen.

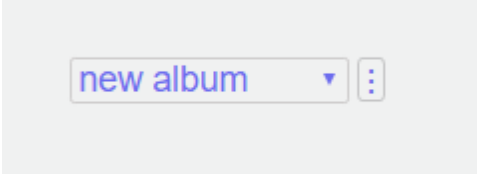
Ebenfalls lässt sich das Bild in groß ansehen oder löschen, wenn man rechts unten den gelben Mülleimer klickt. Löscht man das Basis Motiv wird das dazugehörige Mosaikbild gelöscht, vice versa.

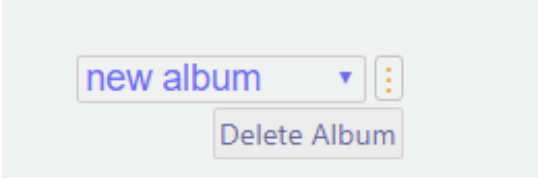


Oben rechts kann man seine Alben auswählen und es werden auch nur die Bilder angezeigt, welche auch in diesen sind.



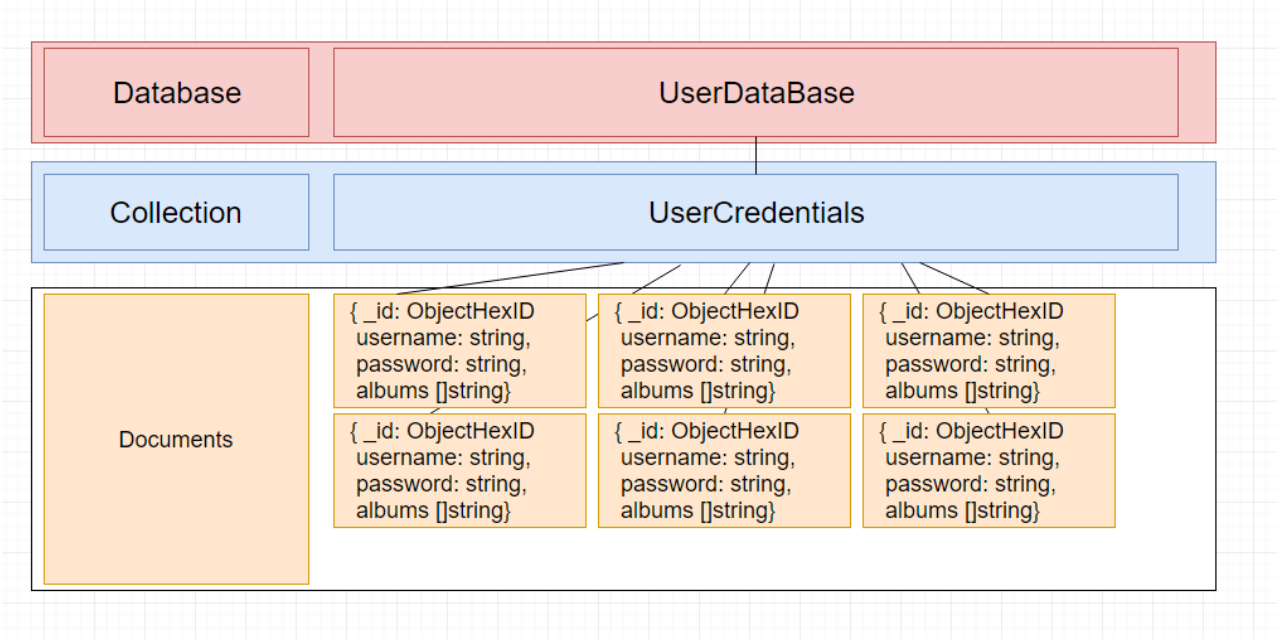

Hat man ein Album ausgewählt, erschein ein Dropdown Menü welchen einen erlaubt das ganze Album zu löschen, die Bilder in diesem Album werden ebenfalls automatisch gelöscht.



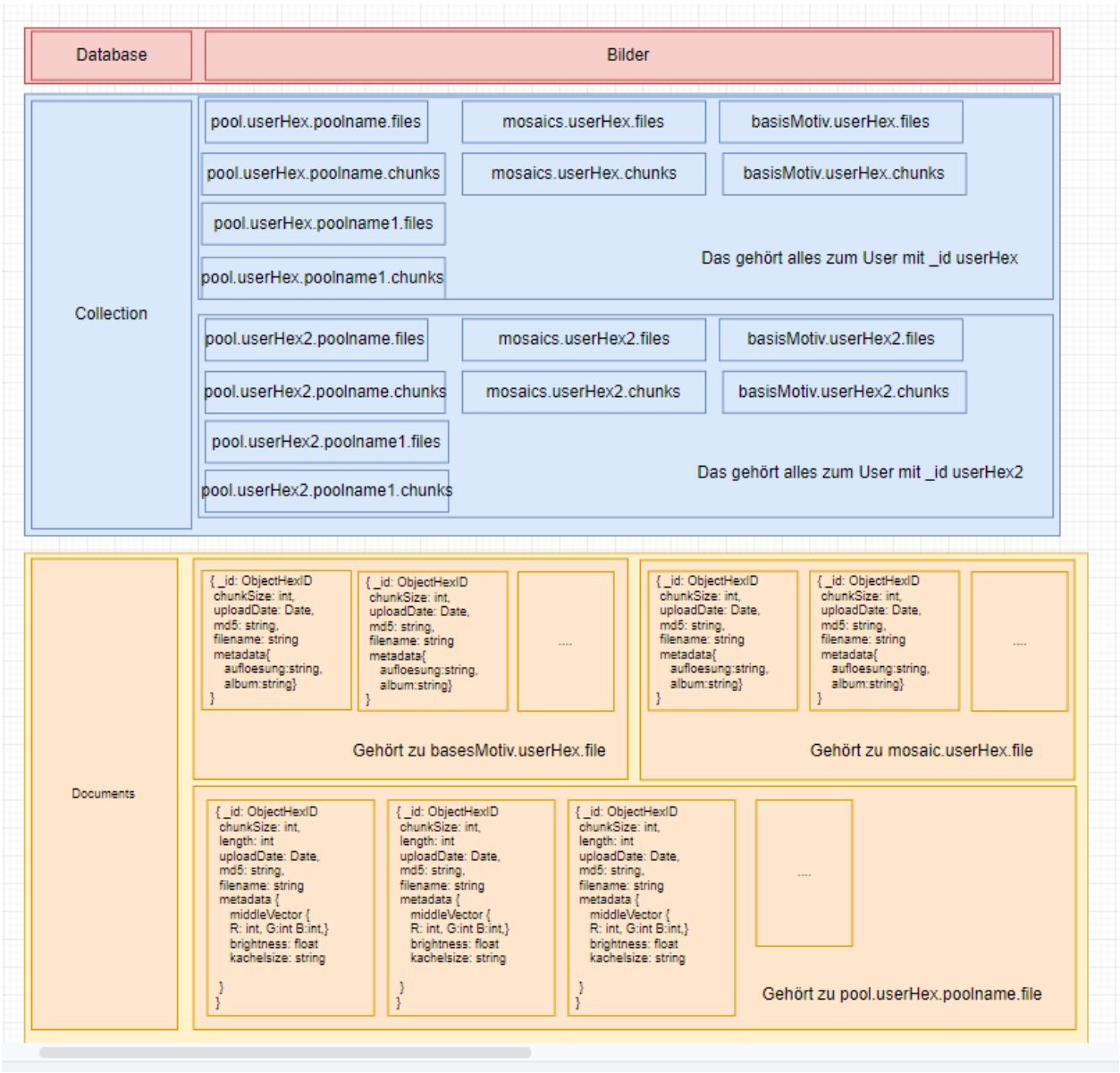

Die Mosaic Seite ist genauso wie die Basic Motives Seite aufgebaut, da beide dasselbe Template benutzen, nur das eben Mosaikbilder angezeigt werden. So wäre es redundant die Gallery Page vorzustellen.

# 2. Die Datenbankstruktur




Als erstes haben wir die Userdatabase, die besitzt nur eine Collection namens UserCredentials, in diesen sind alle Nutzerdaten gespeichert, so besteht das Document eines Users aus den Feldern: _id, username, password und er besitzt albums Array welche daten vom Typen string erhält. Im grundegenommen kann man sagen, dass jeder Nutzer ein Document ist.

Mit der zweiten Datenbank, in denen die Bilder gespiechert sind sieht es allerding komplizierter aus.




Hier sehen wir die Datenbank Bilder, die DB beinhaltet Collections für mehrere Pools, mosaike und BasisMotiven. Jedem User (aus Datenbank UserDataBank) wird mit seiner userHex nur eine basisMotiv und Mosaik Gridfs Collection zugeordnet, mit den Pools sieht es anders aus da kann jeder User mehrere Pool Gridfs Collections besitzen.
Mit der Userhex wird festgestellt, dass auch jeder User nur Zugang zu seinen eigenen Mosaiken, Basismotiven und Pools hat.
Die Basismotive und Mosaiken besitzen in den Metadaten extra einen string album, damit diese auch zu einem Album des Users zuzuordnen sind.

# 3. die Struktur der Go-Anwendung

Insgesamt hat die Go-Anwendung alleine ca. 2000 Zeilen Code, mit ca 45 Funktionen

## 3.1 Pakete

Es wurden folgende Pakete benutzt:

```
18  import (
19      "bytes"
20      "errors"
21      "fmt"
22      "image"
23      "image/color"
24      "image/draw"
25      "image/png"
26      "io"
27      "math"
28      "math/rand"
29      "net/http"
30      "sort"
31      "strconv"
32      "strings"
33      "text/template"
34      "time"
35
36      "github.com/disintegration/imaging"
37      "github.com/globalsign/mgo"
38      "github.com/globalsign/mgo/bson"
39  )
```

Imgrunde genommen sind dies alles builtin Pakete, außer die Pakete von github, welche wir ja importieren sollten um die Bilder zu bearbeiten.

Pakete „strconv“, „strings“,“ math“ wurden alle benutzt um z.B die Nutzereingaben zu verarbeiten.

„math“ wurde benutzt um vorallem die funktionen wie Wurzel ziehen zu nutzen.

„math/rand“ um z.B. den Poolgenerator zu implementieren.

„image“ Paket wurde z.B genutzt um files in Images umzuwandeln mit image.Decode(), welche dann mit den anderen image Paketen bearbeitet werden kann.

```
1299        //file in Image decoden
1300        img, _, err := image.Decode(file)
```

Images müssen jedoch wieder als File verfügbar sein, dazu ist ist das nächste Paket zuständig.

„bytes“, „io“ um image.Images in einen bytes.Buffer zu speichern und dann den Buffer mit bytes.NewReader() zu einen Reader umzuwandeln, welchen wir dann in den Gridfile kopieren können.

Hier ist eine funktion saveIMGinDB welche das oben benannte umsetzt.

```
func saveIMGinDB(w http.ResponseWriter, img image.Image, filename string, gridfsName s
    session, err := mgo.Dial(server)
    check_ResponseToHTTP(err, w)
    defer session.Close()
    db := session.DB(dbNamePics)
    gridfs := db.GridFS(gridfsName)
    gridFile, err := gridfs.Create(filename) // grid-file mit diesem Namen erzeugen:
    if setIdbool {
        gridFile.SetId(setID)
    }
    check_ResponseToHTTP(err, w)
    buff := new(bytes.Buffer) //create buffer
    err = png.Encode(buff, img)
    var bound = img.Bounds()
    gridFile.SetMeta(bson.M{"aufloesung": strconv.Itoa(bound.Max.X) + "x" + strconv.It
    check_ResponseToHTTP(err, w)
    reader := bytes.NewReader(buff.Bytes()) //convert buffer to reader
    _, err = io.Copy(gridFile, reader)      //Copy reader in GridFS
    check_ResponseToHTTP(err, w)
    buff.Reset() //reset Buffer
    gridFile.Close()
    return gridFile.Id()
}
```

## 3.2 Strukturen

Ein Struct, um Nutzerdaten aus der DB herauszulesen und zu verarbeiten/überprüfen:

```
47  type UserCredential2 struct {
48      Id       bson.ObjectId `bson:"_id"`
49      Username string
50      Password string
51      Albums   []string `bson:"albums"`
52  }
```

Ein Struct, um den User per Template Feedback zu geben, ob z.B. falsche Eingabe gemacht wurden etc.

```
54  type LoginSignInFeedback struct {
55      Feedback string
56      Color    string
57  }
```

Struct um Files aus der DB zu lesen:

```
78
79  type fileTemplateStrc struct {
80      ID          bson.ObjectId `bson:"_id"`
81      Filename    string        `bson:"filename"`
82      Length      int32         `bson:"length"`
83      UploadDate  time.Time     `bson:"uploadDate"`
84      Source      string        `bson:"source"`
85      Metadata    Metadatas     `bson:"metadata"`
86      Aufloesung  string        `bson:"aufloesung"`
87      IDHexstring string
88      AuflosungX  string
89      AuflosungY  string
90      DbFileDir   string
91  }
92
```

Struct um Metadaten eines files im Gridfs herauszulesen:

```
107
108 type Metadatas struct {
109     //MiddleColor    color.Color `bson:"middleColor"`  //MiddleColor besteht as r g b a
110     MiddleColorVec Vector3D `bson:"middleVector"` //MiddleColorVec besteht as r g b
111     Brightness     float64  `bson:"brightness"`   //Helligkeit, die länge von MiddleColor
112     Kachelsize     string   `bson:"kachelsize"`
113     Aufloesung     string   `bson:"aufloesung"`
114 }
115
```

Struct Vector3D um die RGB werde zu verarbeiten und berechnen für den Mosaikalgorithmus vorallem wichtig bei der Berechnung der Helligkeit und Farbabständen

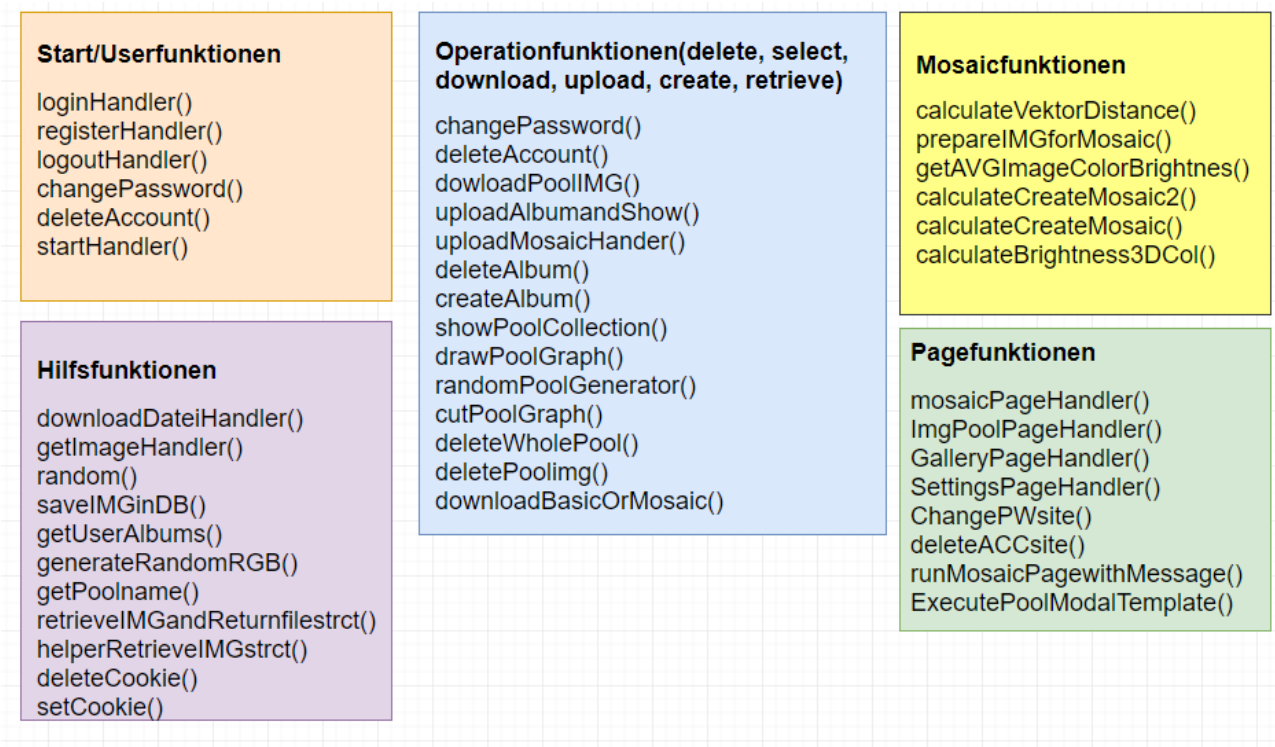
```
116  type Vector3D struct {
117      X, Y, Z uint8
118  }
119
120  type Vector3Df struct {
121      X, Y, Z float64
122  }
```

Struct um die für SVG graphic (für die Farbverteilung der Poole)

```
131  type Graphstrct struct {
132      Poolname      string
133      AvgRGB        Vector3Df
134      AvgBrightness float64
135      AvgDrawCoord  Vector3Df
136  }
137
```

## 3.3 Funktionen

Da es recht viele Funktionen gibt versuche ich hier mit dieser Graphik einen Überblick zu schaffen




Hier habe ich versucht, die Funktionen in Start/Userfunktionen, Hilfsfunktionen, Operationfunktionen, Mosaicfunktionen und Pagefunktionen einzuteilen.

**Start/Userfunktionen** sind die Funktionen, welche ausgeführt werden, wenn sich ein User einloggt, registriert, ausloggt oder etwas mit seinem Account macht.

**Operationsfunktionen** sind Funktionen die damit zu tun hat, das man Daten in der DB selektiert, löscht, erstellt, herausnimmt, downloaded oder uploaded.

**Hilfsfunktionen** sind Funktionen die von den anderen Funktionen genutzt werden. Z.B. wird deleteCookie() und setCookie() funktion häufig genutzt, da ich die Zustände nicht so umgesetzt habe, das sie in der DB gespeichert werden, sondern in Cookies.

**Mosaikfunktionen** sind die Funktionen, die für das Erstellen des Mosaik nötig sind.

**Pagefunktionen** sind die Funktionen welche due Templates ausführen

Es kann sein, dass es viele Überschneidungen zwischen den Funktionen gibt, die ich übersehen habe.

## 3.4 Handler

```
2063  func main() {
2064      //static Fileserver
2065      http.Handle("/", http.FileServer(http.Dir("static")))
2066      http.HandleFunc("/deleteAccount", deleteAccount)
2067      http.HandleFunc("/changePassword", changePassword)
2068      http.HandleFunc("/drawPoolGraph", drawPoolGraph)
2069      http.HandleFunc("/deleteAccSite", deleteAccSite)
2070      http.HandleFunc("/changePWSite", changePWSite)
2071      http.HandleFunc("/deleteAlbum", deleteAlbum)
2072      http.HandleFunc("/selectAlbumAndShow", selectAlbumAndShow)
2073      http.HandleFunc("/createAlbum", createAlbum)                          // http://localhost:
2074      http.HandleFunc("/downloadMosaicOrBasic", downloadBasicOrMosaicImage) // http://localhost:
2075      http.HandleFunc("/deleteMosaicAndBasic", deleteBasicAndMosaicImage)   // http://localhost:
2076      http.HandleFunc("/gallery", galleryPageHandler)
2077      http.HandleFunc("/deleteWholePool", deleteWholePoolHandler) // http://localhost:4242/delet
2078      http.HandleFunc("/downloadPoolImg", downloadPoolImg)        //  http://localhost:4242/down
2079      http.HandleFunc("/deletePoolImg", deletePoolImageHandler)   // http://localhost:4242/delet
2080      http.HandleFunc("/gridGetImage", getImageHandler)           // http://localhost:4242/gridG
2081      http.HandleFunc("/showPool", showPoolCollection)
2082      http.HandleFunc("/settings", settingsPageHandler)
2083      http.HandleFunc("/baseMotive", baseMotifPageHandler)
2084      http.HandleFunc("/mosaic", mosaicPageHandler)
2085      http.HandleFunc("/imgPool", imgPoolPageHandler) //http://localhost:4242/imgPool
2086      http.HandleFunc("/logout", logoutHandler)
2087      http.HandleFunc("/login", loginHandler)
2088      http.HandleFunc("/register", registerHandler) // http://localhost:4242/register
2089      http.HandleFunc("/picx", startHandler)        // http://localhost:4242/picx
2090      err := http.ListenAndServe(":4242", nil)
2091      if err != nil {
2092          fmt.Println(err)
2093      }
```

Es sind insgesamt 24 Handler registriert.

# 4. Algorithmus zur Mosaikerstellung

Es wurde der Algorithmus genommen welcher, die mittleren Farbabstände berechnet

die Funktion calculateCreateMosaic, erstellt aus einem „sourceImage“ ein Mosaikbild und returnt es. Als erstes wird aus dem ausgewähltem Pool alle Poolbilder heraus gezogen und in einem array von stuct fileTempaltestrc{} (Zeile. 1477 result) gespeichert. Dann wird in einer schleife eine Fläche die, die Kachelgröße enspricht aus dem SourceImage den mittleren RGB farbe berechnet (Zeile 1492 farbVector)
Dazu wurde eine Funktion getAVGImageColorAndBrightness erstellt, die die mitleren RGB werte in einem 3Dvektor speichert und returnt. Dann wird durch „result“ , welche alle Poolbilder enthält durchgeloopt und geschaut welche Kachel den geringsten Farbabstand hat. Um den farbabstand zwischen 2 Vektoren zu berechnen wurde eine Funktion CalculateVectorDistance () erstellt

```
1463
1464  //version mit Farbabstand----------
1465  func calculateCreateMosaic(w http.ResponseWriter, r *http.Request, sourceImg image.Image, kachelsize int) image.Image
1466      fmt.Println(kachelsize)
1467      cookie, _ := r.Cookie(currentUser)
1468      poolname := r.PostFormValue("selectedPool")
1469      // DB-Verbindung:
1470      session, err := mgo.Dial(server)
1471      check_ResponseToHTTP(err, w)
1472      defer session.Close()
1473      db := session.DB(dbNamePics)
1474      //eines der Pool GridFs-collection wählen :
1475      gridfsName := poolFsName + "." + cookie.Value + "." + poolname
1476      gridfs := db.GridFS(gridfsName)
1477      var result = []fileTemplateStrc{}
1478      //alle Bilder aus dem Pool holen (limit 1000 gesetzt):
1479      iter := gridfs.Find(nil).Limit(9000).Iter()
1480      err = iter.All(&result)
1481      check_ResponseToHTTP(err, w)
1482      iter.Close() //close Iter
1483      bounds := sourceImg.Bounds()
1484      rowNumb, colNumb := bounds.Max.X/kachelsize, bounds.Max.Y/kachelsize
1485      var maxFarbLength = 10
1486      var farbabstaende = []Kachelstrct{}
1487      m := image.NewRGBA(image.Rect(0, 0, bounds.Max.X, bounds.Max.Y))
1488      draw.Draw(m, m.Bounds(), sourceImg, image.Point{0, 0}, draw.Src)
1489      x2, y2 := kachelsize, kachelsize
1490      for y := 0; y < colNumb; y++ {
1491          for x := 0; x < rowNumb; x++ {
1492              farbVector, _ := getAvgImageColorAndBrightness(kachelsize*x, x2, kachelsize*y, y2, kachelsize, sourceImg)
1493              for _, el := range result {
1494                  var farbabstand = CalculateVectorDistance(farbVector, el.Metadata.MiddleColorVec)
1495                  if len(farbabstaende) < maxFarbLength {
1496                      farbabstaende = append(farbabstaende, Kachelstrct{
1497                          Farbabstand: farbabstand,
1498                          ID:          el.ID,
1499                      })
1500                  } else {
1501                      sort.Sort(FarbabstandSort(farbabstaende))
1502                      if farbabstaende[maxFarbLength-1].Farbabstand > farbabstand {
1503                          farbabstaende[maxFarbLength-1].Farbabstand = farbabstand
1504                          farbabstaende[maxFarbLength-1].ID = el.ID
1505                      }
1506                  }
1507              }
1508              file, _ := gridfs.OpenId(farbabstaende[random(0, 6)].ID)
1509              //file, _ := gridfs.OpenId(farbabstaende[0].ID)
1510              defer file.Close()
1511              kachel, _, _ := image.Decode(file)
1512              farbabstaende = nil
1513              draw.Draw(m, m.Bounds(), kachel, image.Point{-kachelsize * x, -kachelsize * y}, draw.Over)
1514              x2 += kachelsize
1515          }
1516          x2 = kachelsize
1517          y2 += kachelsize
1518      }
1519      return m
1520  }
```

Nachdem durchgeloppt wurde, wird die Kachel mit dem geringsten Farbabstand auf die Fläche des SourceImages gelegt und die Schleife beginnt für die nächsgelegene Fläche, die passendste Kachel zu finden.

Die dazu benötigten Stucts sind ein **Vector3D** struct

```
116   type Vector3D struct {
117       X, Y, Z uint8
118   }
```

und **fileTemplatestrc** um die mittleren RBG Werte der Kacheln aus der DB zu lesen.

```
79  type fileTemplateStrc struct {
80      ID          bson.ObjectId `bson:"_id"`
81      Filename    string        `bson:"filename"`
82      Length      int32         `bson:"length"`
83      UploadDate  time.Time     `bson:"uploadDate"`
84      Source      string        `bson:"source"`
85      Metadata    Metadatas2    `bson:"metadata"`
86      Aufloesung  string        `bson:"aufloesung"`
87      IDHexstring string
88      AuflosungX  string
89      AuflosungY  string
90      DbFileDir   string
91  }
```

Die Hilfsfunktionen die verwendet wurden:

Eine Funktion **getAVGColorAndBrightness** welche von einem Image für eine Fläche die mittleren RGB und Helligkeit berechnet und returnt.

```
1559  func getAvgImageColorAndBrightness(xstart int, xBound int, ystart int, yBound int, kachelsize int, i image.Image) (Vector3D, float64) {
1560      var r, g, b uint32
1561      //bounds := i.Bounds()
1562      for y := ystart; y < yBound; y++ {
1563          for x := xstart; x < xBound; x++ {
1564              pr, pg, pb, _ := i.At(x, y).RGBA()
1565              r += pr //pixelrotanteil akkumulieren
1566              g += pg //pixelgelbanteil akkumulieren
1567              b += pb //pixelblauanteil akkumulieren
1568          }
1569      }
1570      d := uint32(kachelsize * kachelsize) //Kachelfläche
1571      r /= d
1572      g /= d
1573      b /= d
1574      var rgbVector = Vector3D{X: uint8(r / 0x101), Y: uint8(g / 0x101), Z: uint8(b / 0x101)}
1575      var brightness = CalculateBrightness3DCol(rgbVector)
1576      //color.NRGBA{uint8(r / 0x101), uint8(g / 0x101), uint8(b / 0x101), 255},
1577      return rgbVector, brightness
1578  }
```

Einmal die Funktion **CalculateBrightness** die, die Helligkeit eines RGB Vektors berechnet und **CalculateVectorDistance** welche den Farbabstand zwischen zwei Vektoren berechnet

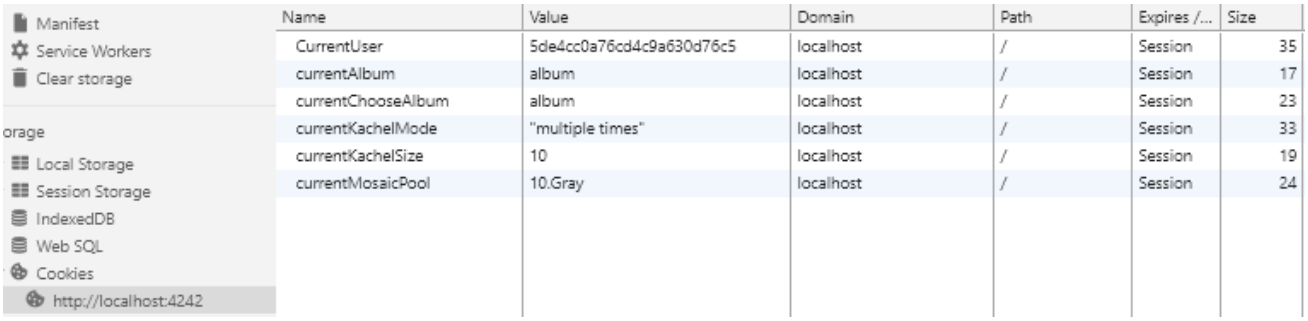
```
//-------------------------------------------------------------------------------------------------
func CalculateBrightness3DCol(n Vector3D) float64 { //auch vektorlänge genannt
    dx := float64(n.X)
    dy := float64(n.Y)
    dz := float64(n.Z)
    return math.Sqrt(dx*dx + dy*dy + dz*dz)
}

//-------------------------------------------------------------------------------------------------
func CalculateVectorDistance(n1 Vector3D, n2 Vector3D) float64 {
    //Farbabstand zwischen zwei Vektoren
    //vorsicht uint8 nimmt nur zahlen von 0 - 255
    dx := float64(n2.X) - float64(n1.X)
    dy := float64(n2.Y) - float64(n1.Y)
    dz := float64(n2.Z) - float64(n1.Z)
    return math.Sqrt(dx*dx + dy*dy + dz*dz)
}
```

# 5. Benutzer/innen- und Zustandsverwaltung

Die Zustandverwaltung findet leider nicht wie in der Aufgabe verlangt in der Datenbank statt, sondern mit Hilfe von Cookies die mit dem Go-Programm realisiert wurden.

So werden die Optionen, die man bei der Erstellung von Poolen oder Mosaiken in Cookies gesetzt, damit der User diese nicht wieder neu setzen muss.



# 6. Ajax-Kommunikation

```
var xhr5 = new XMLHttpRequest();
var xhr4 = new XMLHttpRequest();
var xhr3 = new XMLHttpRequest();
var xhr2 = new XMLHttpRequest();
var xhr = new XMLHttpRequest();
xhr.addEventListener("load", function () {···
});
xhr5.addEventListener("load", function () {···
});
xhr4.addEventListener("load", function () {···
});
xhr2.addEventListener("load", function () {···
});
xhr3.addEventListener("load", function () {···
});
```

Insgesamt wurden fünf XMLHttpRequest instanziiert.

Für folgende Zwecke:

- Eine gesamte neue Seite aktualisieren
- Das Grafikmodal aktualisieren (damit ist das RGB Diagram gemeint)
- Das PoolModal aktualisieren (damit ist die genauere Darstellung eines Pool mit Bilder gemeint)
- Die Settingspage aktualisieren
- Den UserfeedBack aktualisieren

# 7. Zusätzlich: sehr kurzer Testbericht (wie wurde getestet ?)

Es wurde vorwiegend im Program mittels fmt.Printf z.B. getestet ob RGB, Helligkeitswerte richtig berechnet wurden. So wurde oftmals festgestellt das unsinnige Werte rauskammen und für Bilder welche eine hohe Helligkeit aufweisen sollten, eine Helligkeit von 0 berechnet wurde.
Dies lag daran das die RGB werte in uint8 format gespeichert und weiterberrechnet worden sind. Jedoch war dies ein großer Fehler da uint8 nur Werte von 0-255 annimmt, weshalb man in float umwandeln sollte.

Es  wurde auch per fmt.Println getestet, ob Anfragen per Javascript überhaupt die Go Handler erreichten.
Ebenfalls wurde per Javascript mittels console.log gestestet ob Handler überhaupt angingen.

Ein weiteres Problem stellte sich beim Pool Generator heraus, es wurde per fmt.prinln festgestellt das rand.Intn immer die selbe Zahl liefert.

Da der PoolGenerator verschiedenen Kacheln generieren soll, wäre es fatal wenn rand.Intn dieselbe Zahl liefert.
Der Grund liegt daran, dass rand.Intn deterministisch ist (es von einem gegebenen Startzustand, immer die gleiche Ausgabe hat).

So muss man einen Seed setzen muss, um an verschiedene Zahlen zu gelangen. Hier ist eine Funktion Random welche einen Seed neu setzt.

```
func random(min, max int) int {
    rand.Seed(time.Now().UTC().UnixNano())
    return rand.Intn(max-min) + min
}
```