Painting Analyzer Intelligence Network

Ciobanu Bogdan	@bciobanu
Iordache Ioan-Bogdan	@iordachebogdan
Manghiuc Teodor-Florin	@eu3neuom
Marian Darius	@darius98
Popa Andrei	@AndreiNet

Grupa 331

18 Iunie 2020

PAIN

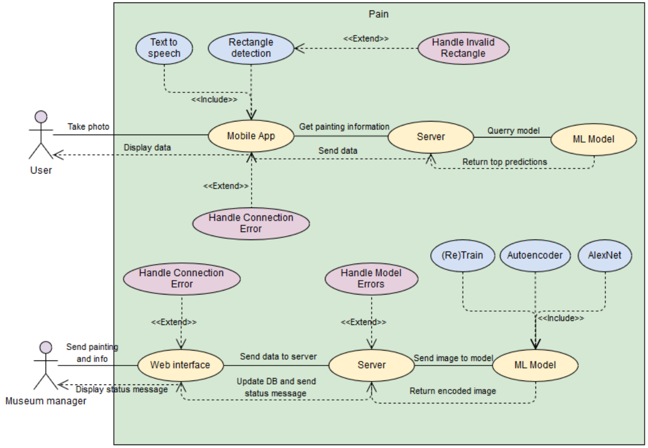


Figure 1: PAINT use case diagram

PAIN

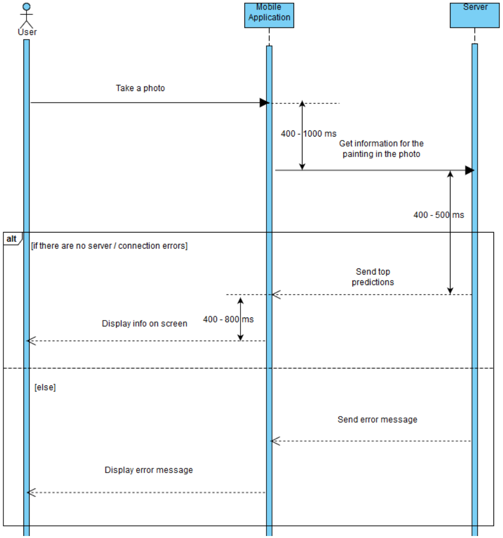


Figure 2: Client sequence diagram

PAIN

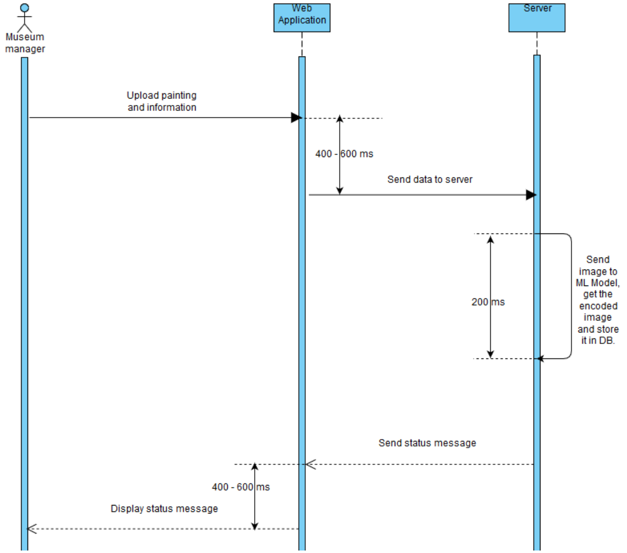


Figure 3: Museum manager sequence diagram

Dashboard

- Platform where museums upload their pictures and information about their paintings.
- Acts as the source of information for the mobile app.
- Needs to be reliable, fault tolerant and distributed.
- Built as a server-side model-view-controller Elixir application.

Why we chose Elixir

- Elixir is built on top of the Beam virtual machine and takes full advantage of it.
- Processes are given a design pattern such as servers, finite state machines, event handlers or supervisors, all packaged in reusable libraries.
- They abstract and take care of all tricky edge cases which occur with concurrent programming, providing a solid and tried approach to problem-solving, making the code easier to understand and maintain, reducing maintenance costs and stopping developers from reinventing a square wheel.

Supervisor trees

- The whole language is built with the purpose of taking crashes and failures, and making them so manageable it becomes possible to use them as a tool.
- They start with a simple concept, a supervisor, whose only job is to start processes, look at them, and restart them when they fail.
- When you structure Elixir programs, everything you feel is fragile and should be allowed to fail has to move deeper into the hierarchy, and what is stable and critical needs to be reliable is higher up.

Deploying

- We have used our student credits on Microsoft Azure.
- We deploy a docker image to Docker Hub, which is then deployed to our web server.
- Lightweight images were a priority when deploying. We had to make use of image caching so that our Docker builds do not take so long.
- The model code needs to be a part of this and deep learning applications are notoriously big.
- We ended up with an image that was roughly only 700 MBs.
- The base OS image and the Python dependencies account for 500 MBs of it.
- The big parts could be cached very well for our needs, for example, when a change was pushed in the dashboard code, only 60 MBs needed to be re-deployed.

Administration tooling

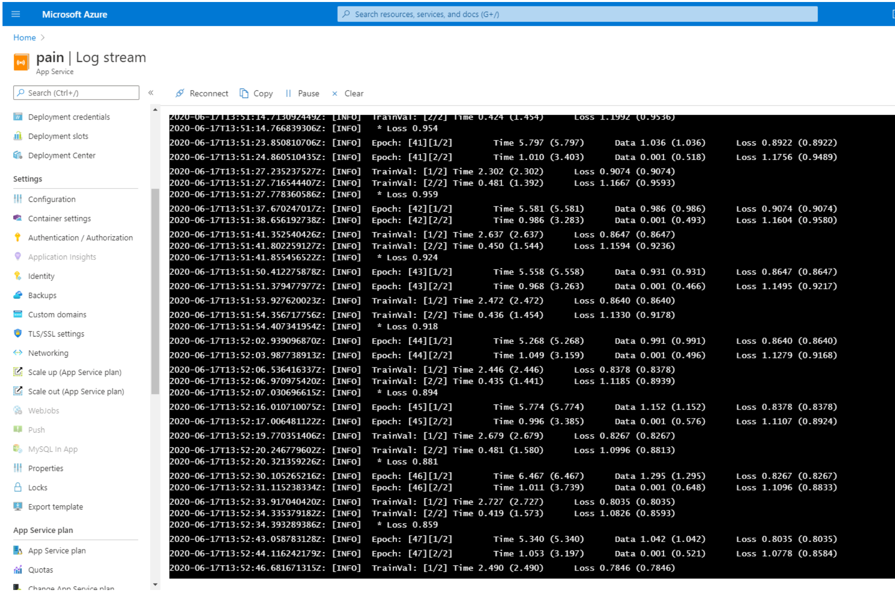
- Retraining will be started manually to care for excessive load and to be overseen thoroughly.

  ```
  curl -i -X POST -H "Content-Length: 0"
  https://pain.azurewebsites.net/api/train
  ```

- Reloading the worker in case of a failure is also as simple as doing a POST request.

  ```
  curl -i -X POST -H "Content-Length: 0"
  https://pain.azurewebsites.net/api/reload
  ```

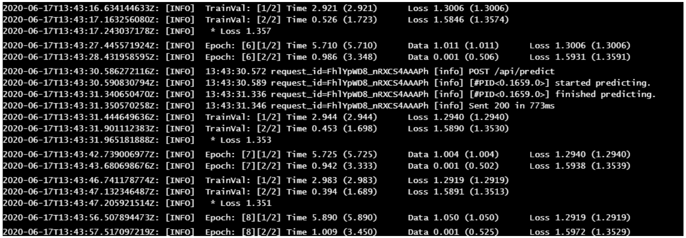
Retraining the model live



```
2020-06-17T13:51:14.713092449Z: [INFO]  TrainVal: [2/2] Time 0.424 (1.454)      Loss 1.1992 (0.9536)
2020-06-17T13:51:14.766839306Z: [INFO]   * Loss 0.954
2020-06-17T13:51:23.850810706Z: [INFO]  Epoch: [41][1/2]        Time 5.797 (5.797)      Data 1.036 (1.036)      Loss 0.8922 (0.8922)
2020-06-17T13:51:24.860510435Z: [INFO]  Epoch: [41][2/2]        Time 1.010 (3.403)      Data 0.001 (0.518)      Loss 1.1756 (0.9489)
2020-06-17T13:51:27.235237527Z: [INFO]  TrainVal: [1/2] Time 2.302 (2.302)      Loss 0.9074 (0.9074)
2020-06-17T13:51:27.716544407Z: [INFO]  TrainVal: [2/2] Time 0.481 (1.392)      Loss 1.1667 (0.9593)
2020-06-17T13:51:27.778360586Z: [INFO]   * Loss 0.959
2020-06-17T13:51:37.670247017Z: [INFO]  Epoch: [42][1/2]        Time 5.581 (5.581)      Data 0.986 (0.986)      Loss 0.9074 (0.9074)
2020-06-17T13:51:38.656192738Z: [INFO]  Epoch: [42][2/2]        Time 0.986 (3.283)      Data 0.001 (0.493)      Loss 1.1604 (0.9580)
2020-06-17T13:51:41.352540426Z: [INFO]  TrainVal: [1/2] Time 2.637 (2.637)      Loss 0.8647 (0.8647)
2020-06-17T13:51:41.802259127Z: [INFO]  TrainVal: [2/2] Time 0.450 (1.544)      Loss 1.1594 (0.9236)
2020-06-17T13:51:41.855456522Z: [INFO]   * Loss 0.924
2020-06-17T13:51:50.412275878Z: [INFO]  Epoch: [43][1/2]        Time 5.558 (5.558)      Data 0.931 (0.931)      Loss 0.8647 (0.8647)
2020-06-17T13:51:51.379477977Z: [INFO]  Epoch: [43][2/2]        Time 0.968 (3.263)      Data 0.001 (0.466)      Loss 1.1495 (0.9217)
2020-06-17T13:51:53.927620023Z: [INFO]  TrainVal: [1/2] Time 2.472 (2.472)      Loss 0.8640 (0.8640)
2020-06-17T13:51:54.356717756Z: [INFO]  TrainVal: [2/2] Time 0.436 (1.454)      Loss 1.1330 (0.9178)
2020-06-17T13:51:54.407341954Z: [INFO]   * Loss 0.918
2020-06-17T13:52:02.939096870Z: [INFO]  Epoch: [44][1/2]        Time 5.268 (5.268)      Data 0.991 (0.991)      Loss 0.8640 (0.8640)
2020-06-17T13:52:03.987738913Z: [INFO]  Epoch: [44][2/2]        Time 1.049 (3.159)      Data 0.001 (0.496)      Loss 1.1279 (0.9168)
2020-06-17T13:52:06.536416337Z: [INFO]  TrainVal: [1/2] Time 2.446 (2.446)      Loss 0.8378 (0.8378)
2020-06-17T13:52:06.970975420Z: [INFO]  TrainVal: [2/2] Time 0.435 (1.441)      Loss 1.1185 (0.8939)
2020-06-17T13:52:07.030696615Z: [INFO]   * Loss 0.894
2020-06-17T13:52:16.010710075Z: [INFO]  Epoch: [45][1/2]        Time 5.774 (5.774)      Data 1.152 (1.152)      Loss 0.8378 (0.8378)
2020-06-17T13:52:17.006481122Z: [INFO]  Epoch: [45][2/2]        Time 0.996 (3.385)      Data 0.001 (0.576)      Loss 1.1107 (0.8924)
2020-06-17T13:52:19.770351406Z: [INFO]  TrainVal: [1/2] Time 2.679 (2.679)      Loss 0.8267 (0.8267)
2020-06-17T13:52:20.246779602Z: [INFO]  TrainVal: [2/2] Time 0.481 (1.580)      Loss 1.0996 (0.8813)
2020-06-17T13:52:20.321359226Z: [INFO]   * Loss 0.881
2020-06-17T13:52:30.105265216Z: [INFO]  Epoch: [46][1/2]        Time 6.467 (6.467)      Data 1.295 (1.295)      Loss 0.8267 (0.8267)
2020-06-17T13:52:31.115238334Z: [INFO]  Epoch: [46][2/2]        Time 1.011 (3.739)      Data 0.001 (0.648)      Loss 1.1096 (0.8833)
2020-06-17T13:52:33.917040420Z: [INFO]  TrainVal: [1/2] Time 2.727 (2.727)      Loss 0.8035 (0.8035)
2020-06-17T13:52:34.335379182Z: [INFO]  TrainVal: [2/2] Time 0.419 (1.573)      Loss 1.0826 (0.8593)
2020-06-17T13:52:34.393289386Z: [INFO]   * Loss 0.859
2020-06-17T13:52:43.058783128Z: [INFO]  Epoch: [47][1/2]        Time 5.340 (5.340)      Data 1.042 (1.042)      Loss 0.8035 (0.8035)
2020-06-17T13:52:44.116242179Z: [INFO]  Epoch: [47][2/2]        Time 1.053 (3.197)      Data 0.001 (0.521)      Loss 1.0778 (0.8584)
2020-06-17T13:52:46.681671315Z: [INFO]  TrainVal: [1/2] Time 2.490 (2.490)      Loss 0.7846 (0.7846)
```

Other workers are available

- We planned for an application that tries to be available all the time.
- Therefore, we must be able to handle predictions while retraining, albeit using an older model version.

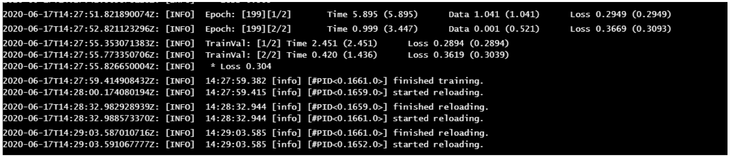


```
2020-06-17T13:43:16.634144633Z: [INFO]  TrainVal: [1/2] Time 2.921 (2.921)      Loss 1.3006 (1.3006)
2020-06-17T13:43:17.163256080Z: [INFO]  TrainVal: [2/2] Time 0.526 (1.723)      Loss 1.5846 (1.3574)
2020-06-17T13:43:17.243037178Z: [INFO]   * Loss 1.357
2020-06-17T13:43:27.445571924Z: [INFO]  Epoch: [6][1/2] Time 5.710 (5.710)      Data 1.011 (1.011)      Loss 1.3006 (1.3006)
2020-06-17T13:43:28.431958595Z: [INFO]  Epoch: [6][2/2] Time 0.986 (3.348)      Data 0.001 (0.506)      Loss 1.5931 (1.3591)
2020-06-17T13:43:30.586272116Z: [INFO]  13:43:30.572 request_id=FhlYpWD8_nRXCS4AAAPh [info] POST /api/predict
2020-06-17T13:43:30.590830794Z: [INFO]  13:43:30.589 request_id=FhlYpWD8_nRXCS4AAAPh [info] [#PID<0.1659.0>] started predicting.
2020-06-17T13:43:31.340650470Z: [INFO]  13:43:31.336 request_id=FhlYpWD8_nRXCS4AAAPh [info] [#PID<0.1659.0>] finished predicting.
2020-06-17T13:43:31.350570258Z: [INFO]  13:43:31.346 request_id=FhlYpWD8_nRXCS4AAAPh [info] Sent 200 in 773ms
2020-06-17T13:43:31.444649636Z: [INFO]  TrainVal: [1/2] Time 2.944 (2.944)      Loss 1.2940 (1.2940)
2020-06-17T13:43:31.901112383Z: [INFO]  TrainVal: [2/2] Time 0.453 (1.698)      Loss 1.5890 (1.3530)
2020-06-17T13:43:31.965181888Z: [INFO]   * Loss 1.353
2020-06-17T13:43:42.739006977Z: [INFO]  Epoch: [7][1/2] Time 5.725 (5.725)      Data 1.004 (1.004)      Loss 1.2940 (1.2940)
2020-06-17T13:43:43.680698676Z: [INFO]  Epoch: [7][2/2] Time 0.942 (3.333)      Data 0.001 (0.502)      Loss 1.5938 (1.3539)
2020-06-17T13:43:46.741178774Z: [INFO]  TrainVal: [1/2] Time 2.983 (2.983)      Loss 1.2919 (1.2919)
2020-06-17T13:43:47.132346487Z: [INFO]  TrainVal: [2/2] Time 0.394 (1.689)      Loss 1.5891 (1.3513)
2020-06-17T13:43:47.205921514Z: [INFO]   * Loss 1.351
2020-06-17T13:43:56.507894473Z: [INFO]  Epoch: [8][1/2] Time 5.890 (5.890)      Data 1.050 (1.050)      Loss 1.2919 (1.2919)
2020-06-17T13:43:57.517097219Z: [INFO]  Epoch: [8][2/2] Time 1.009 (3.450)      Data 0.001 (0.525)      Loss 1.5972 (1.3529)
```

Figure 4: Workers handling a predict query during retraining

Hot reloading the model

After a new model has been trained, every worker gets updated with the new model with virtually no downtime.



```
2020-06-17T14:27:51.821890074Z: [INFO]  Epoch: [199][1/2]       Time 5.895 (5.895)      Data 1.041 (1.041)      Loss 0.2949 (0.2949)
2020-06-17T14:27:52.821123296Z: [INFO]  Epoch: [199][2/2]       Time 0.999 (3.447)      Data 0.001 (0.521)      Loss 0.3669 (0.3093)
2020-06-17T14:27:55.353071383Z: [INFO]  TrainVal: [1/2] Time 2.451 (2.451)      Loss 0.2894 (0.2894)
2020-06-17T14:27:55.773350706Z: [INFO]  TrainVal: [2/2] Time 0.420 (1.436)      Loss 0.3619 (0.3039)
2020-06-17T14:27:55.826650004Z: [INFO]   * Loss 0.304
2020-06-17T14:27:59.414908432Z: [INFO]  14:27:59.382 [info] [#PID<0.1661.0>] finished training.
2020-06-17T14:28:00.174080194Z: [INFO]  14:27:59.415 [info] [#PID<0.1659.0>] started reloading.
2020-06-17T14:28:32.982928939Z: [INFO]  14:28:32.944 [info] [#PID<0.1659.0>] finished reloading.
2020-06-17T14:28:32.988573370Z: [INFO]  14:28:32.944 [info] [#PID<0.1661.0>] started reloading.
2020-06-17T14:29:03.587010716Z: [INFO]  14:29:03.585 [info] [#PID<0.1661.0>] finished reloading.
2020-06-17T14:29:03.591067777Z: [INFO]  14:29:03.585 [info] [#PID<0.1652.0>] started reloading.
```

Figure 5: Workers reloading their model sequentially after the model was updated

Deep Learning Model Architecture

We used deep learning to build a predictor model for the paintings. This model has two main components: an autoencoder and a prediction algorithm. We will present them separately.

Autoencoder

- We used the pytorch library to implement a convolutional autoencoder. This architecture has two parts: an encoder and a decoder. The encoder uses subsampling to generate an embedding vector for an image. The decoder uses upsampling to try to recreate the original image from the embedding vector.
- The encoder is built from 6 convolutional layers with kernel dimensions ($size \times out\_channels$): 3×16, 3×32, 3×64, 3×128, 3×256, 3×512. The decoder is built symmetrically using the same dimensions, in reverse order, for 6 inverse convolutional layers. We used inplace ReLU activation for every layer.

Autoencoder

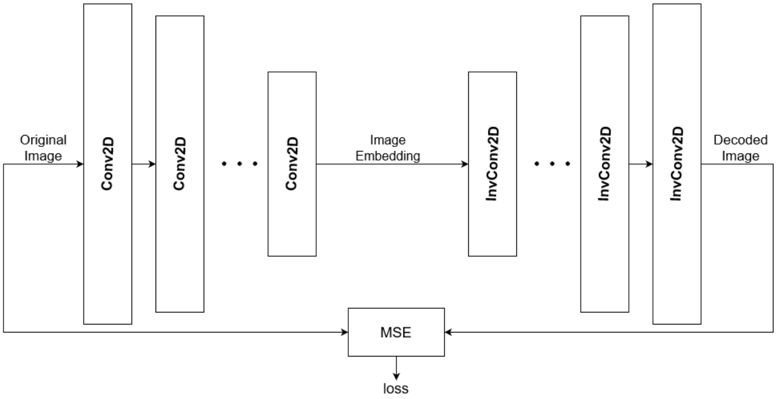


Figure 6: Autoencoder architecture

Autoencoder

- We split the dataset into training and validation sets. Normalization was applied on each image by resizing and cropping to 256×256 pixels and applying standardization technique on intensity values.
- The dataset we used is the Best Artworks of All Time Dataset available online. We used Mean Squared Error for the loss function to determine the difference from the original image to the decoded one. For training we used Adam optimizer with learning rate $= 10^{-2}$ and weight decay $= 10^{-4}$; mini-batches of size 16. The total numbers of epochs was set to $12K$, we selected the model which performed best on the validation set.

Predictor

For prediction, for each painting in the database, the embedding vector is computed, L2-normalized and stored in a KD-Tree. The KD-Trees are based on Spotify's Annoy implementation. For a photo of a painting submitted by the user, we employ the next strategy: normalize the image, compute embedding vector, find top n closest images from the database using the KD-Tree.

Predictor

- For prediction accuracy we used a backup encoder based on the pretrained version of AlexNet. The embedding is computed by taking the values from the penultimate fully-connected layer of AlexNet, the intuition being that this layer better encapsulates general features of images, rather than task specific ones.
- Also for better precision we retrain our own model when gathering more paintings from the museums. The training is done in background and the new model is hot-reloaded into the app with virtually no downtime.
- The final prediction is made by merging the predictions from our model and the predictions from AlexNet.

iOS Client

For implementing the client on iOS, we chose UIKit as our core framework. We chose UIKit over SwiftUI because the later is newer and it is not supported on iOS version prior to 13.0. Other frameworks used were CoreImage and Vision for image processing.

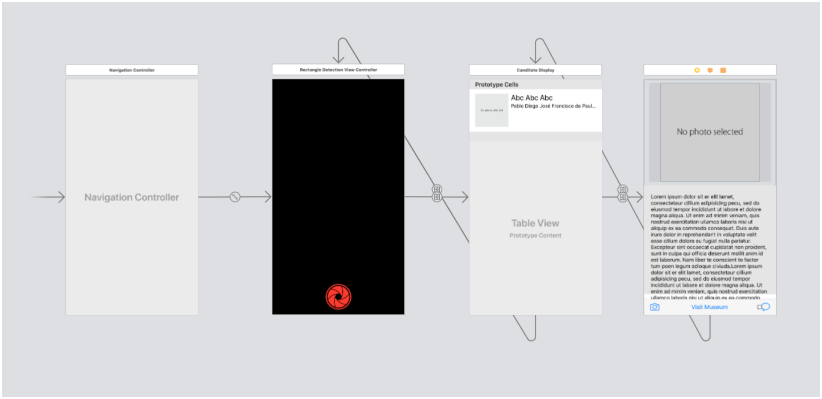
Programming language

The programming language used was Swift as it is a powerful language for making iOS applications because of the following features:

- type system which makes the code more understandable and also safer to write while providing good performance
- optional values instead of null pointers which makes the code safer in comparison with other languages like C/C++ where pointer manipulation can lead to bugs
- OOP features, such as inheritance, polymorphism, interfaces which make designing interfaces easier in strongly-typed languages

Interface

The interface is composed of three main view controllers, which represent the types of pages the user can see: camera, a list of paintings, a paintings with all its details.



Lorem ipsum dolor sit er elit lamet, consectetaur cillium adipisicing pecu, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. Nam liber te conscient to factor tum poen legum odioque civiuda.Lorem ipsum dolor sit er elit lamet, consectetaur cillium adipisicing pecu, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo

The camera view controller

On this page, the frames captured by the camera at a resolution of 640x480 are displayed on the whole screen, meaning that the resolution seen is actually smaller. The reason for choosing this resolution is that it is just big enough to capture all the details of a painting to search it while also allowing the phone to continuously detect possible paintings in the frames.
Over the camera layer there is the detection layer, where a transparent rectangle is drawn over the found image (when there is one). To be detected, the painting should be centered on the screen by the following guidelines.

Image detection guidelines

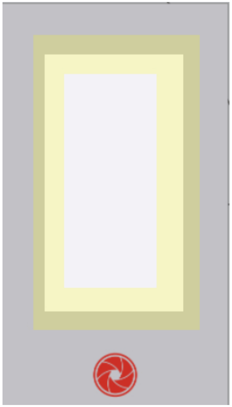


Figure 7: The gray area is shown on the screen of the app. The image should be detected if its corners are in the yellow area (covering a part of the gray area, not show in the UI)

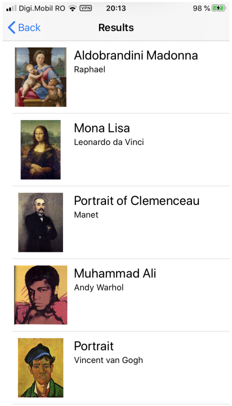
Client image processing summary

Each frame received by the camera is processed using the Vision framework provided by Apple. The steps are the following:

- feed the photo to a *VNDetectRectanglesRequest* from Vision which detects all rectangles (eventually rotated in 3D space) in the image
- from the received rectangles, keep just those that have the corners in the proximity of the frame drawn in the UI
- keep only the rectangles which are not contained in other rectangles

If there is only one rectangle after the filtering, the user can take the photo. The rectangle will be cropped and skewed to its 2D form using the Core Image framework. The image obtained will be sent to the server to be searched for.

The painting table view controller



Aldobrandini Madonna
Raphael

Mona Lisa
Leonardo da Vinci

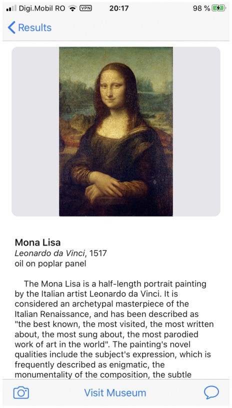
Portrait of Clemenceau
Manet

Muhammad Ali
Andy Warhol

Portrait
Vincent van Gogh

Figure 8: A list with the predicted images. The view is also used to display other images from the same museum (as seen later).

The painting view controller



Mona Lisa
Leonardo da Vinci, 1517
oil on poplar panel

The Mona Lisa is a half-length portrait painting by the Italian artist Leonardo da Vinci. It is considered an archetypal masterpiece of the Italian Renaissance, and has been described as "the best known, the most visited, the most written about, the most sung about, the most parodied work of art in the world". The painting's novel qualities include the subject's expression, which is frequently described as enigmatic, the monumentality of the composition, the subtle

Figure 9: A view displaying the details of a painting. The left button takes the user back to the camera. The right one activates the speech to text functionality. The middle button shows pictures from the same museum.

The *View Museum* is disabled if you already came from that page.

Bibliography

- Which companies are using Erlang, and why? (https://www.erlang-solutions.com/)
- The Zen of Erlang (https://ferd.ca/the-zen-of-erlang.html)
- Why am I interested in Elixir? (http://underjord.io/why-am-i-interested-in-elixir.html)
- Deep Clustering with Convolutional Autoencoders (https://www.researchgate.net/publication/320658590_Deep_Clustering_with_Convolutional_Autoencoders)

Bibliography

- Best Artworks of All Time (https://www.kaggle.com/ikarus777/best-artworks-of-all-time)
- ImageNet classification with deep convolutional neural networks (https://papers.nips.cc/paper/4824-imagenet-classification-\with-deep-convolutional-neural-networks.pdf)
- Spotify's Annoy (https://github.com/spotify/annoy)
- pyTorch (https://pytorch.org/docs/stable/nn.html)