ПРИЛОЖЕНИЕ А
(обязательное)
Программный код проекта

```
import cv2
import face_recognition
from threading import Thread
import datetime
def read_imgs(nums: int)-> list:
    images=[]
    for i in range(1, nums+1):
        images.append( cv2.imread('images/getty'+str(i)+'.jpg',
cv2.IMREAD_UNCHANGED) )
    return images
class ThreadedCamera(object):
    def __init__(self, src=0):
        self.names=["kirill"]
        self.me = face_recognition.load_image_file('images/me.jpeg')
        self.me_enc = face_recognition.face_encodings(self.me)[0]
        self.known_faces = [self.me_enc]
        #capture camera
        self.capture = cv2.VideoCapture(src)
        self.capture.set(cv2.CAP_PROP_BUFFERSIZE, 1)
        self.frame = None
        # Start frame retrieval thread
        self.thread = Thread(target=self.update, args=())
        self.thread.daemon = True
        self.thread.start()
        # FPS = 1/X
        # X = desired FPS
        self.FPS = 1 / 10
        self.FPS_MS = int(self.FPS * 1000)
        #find face
        self.Founded = False
        self.founde_frame = None
        self.threadFind = Thread(target=self.find_face, args=())
        self.threadFind.daemon = True
        self.faces = True
        self.face_frame=None
        #naming_face
        self.threadNaming = Thread(target=self.naming_face, args=())
        self.threadNaming.daemon = True
        self.threadNaming.start()
```

```
    def update(self):
        while True:
            if self.capture.isOpened():
                (self.status, self.frame) = self.capture.read()
                self.face_frame = self.frame
                if self.faces:
                    print("started thred")
                    self.threadFind.start()
                    self.faces = False
            #time.sleep(self.FPS)
    def show_frame(self):
        print("start", datetime.datetime.now())
        cv2.imshow('frame', self.face_frame)
        cv2.waitKey(1)
        print("fin", datetime.datetime.now())
    def find_face(self):
        # gray = cv2.cvtColor(self.frame, cv2.COLOR_BGR2GRAY)
        while True:
            small_frame = cv2.resize(self.face_frame, (0, 0), fx=0.25, fy=0.25)
            rgb_small_frame = small_frame[:, :, ::-1]
            self.face_loc = face_recognition.face_locations(rgb_small_frame)
            face_enc=face_recognition.face_encodings(rgb_small_frame, self.face_loc)
            for face_en in face_enc:
                matches=face_recognition.compare_faces(self.known_faces, face_en)
                if True in matches:
                    self.Founded = True
            if self.Founded:
                print("draw")
                for (top, right, bottom, left), name in zip(self.face_loc, self.names):
                    cv2.rectangle(self.face_frame, (left, top), (right, bottom), (0, 0, 255),
2)
                    # Draw a label with a name below the face
                    cv2.rectangle(self.face_frame, (left, bottom - 35), (right, bottom), (0,
0, 255), cv2.FILLED)
                    font = cv2.FONT_HERSHEY_DUPLEX
                    cv2.putText(self.face_frame, name, (left + 6, bottom - 6), font, 1.0,
(255, 255, 255), 1)
            # self.frame=self.face_frame
```

```
    def naming_face(self):
        print("namnig")
if __name__ == '__main__':
    cam='rtsp://192.168.8.101:8080/h264_ulaw.sdp'
    thr_cam=ThreadedCamera(cam)
    while True:
        try:
            thr_cam.show_frame()
        except:
            pass
    cv2.destroyAllWindows()
```

app.py:

```
const h1 = document.getElementById('header__h1');

const search = document.getElementById('search');

const table = document.getElementById('table');

const urlPersonal = 'personal';

const urlUsers = 'users';

const urlCameras = 'cameras';

const urlCabinets = 'cabinets';

const urlTime = 'time';

const resultsPersonal = [

  {

    id_pers: 1,

    name: {

      username: 'kir',

      last_name: 'lname',

      first_name: 'fname',

    },

    dep_id: {

      name: 'dep1',

    },

  },
```

```
  {
    id_pers: 2,
    name: {
      username: 'user1',
      last_name: 'lname',
      first_name: 'fname',
    },
    dep_id: {
      name: 'dep3',
    },
  },
  {
    id_pers: 3,
    name: {
      username: 'kir',
      last_name: 'lname',
      first_name: 'fname',
    },
    dep_id: {
      name: 'dep3',
    },
  },
];
const resultsTime = [
  {
    id: 1,
    timedate: '2022-12-28T11:50:08.688797Z',
    direction: true,
```

```
    per_id: 1,
    cab_id: 1,
    cam_id: 1,
  },
  {
    id: 2,
    timedate: '2022-12-28T11:50:19.978811Z',
    direction: false,
    per_id: 1,
    cab_id: 1,
    cam_id: 2,
  },
];

const resultsCamera = [
  {
    id_cam: 1,
    cam_model: 'model1',
    cab_id: {
      name: 'cab1',
    },
    in_pos: true,
  },
  {
    id_cam: 2,
    cam_model: 'cam2',
    cab_id: {
      name: 'cab1',
```

```
    },
    in_pos: false,
  },
];

const resultsCabinets = [
  {
    id_cab: 1,
    name: 'cab1',
    floor: 1,
    dep_id: {
      name: 'dep1',
    },
  },
  {
    id_cab: 2,
    name: 'cab2',
    floor: 2,
    dep_id: {
      name: 'dep2',
    },
  },
  {
    id_cab: 3,
    name: 'cab3',
    floor: 3,
    dep_id: {
      name: 'dep3',
```

```
    },
  },
  {
    id_cab: 4,
    name: 'cab4',
    floor: 9,
    dep_id: {
      name: 'dep2',
    },
  },
];

class Router {
  routes = [];
  mode = null;
  root = '/';

  constructor(options) {
    this.mode = window.history.pushState ? 'history' : 'hash';
    if (options.mode) this.mode = options.mode;
    if (options.root) this.root = options.root;
    this.listen();
  }
  add = (path, cb) => {
    this.routes.push({ path, cb });
    return this;
  };
  remove = (path) => {
```

```
  for (let i = 0; i < this.routes.length; i += 1) {
    if (this.routes[i].path === path) {
      this.routes.slice(i, 1);
      return this;
    }
  }
  return this;
};
flush = () => {
  this.routes = [];
  return this;
};
clearSlashes = (path) =>
  path.toString().replace(/\/$/, '').replace(/^\//, '');
getFragment = () => {
  let fragment = '';
  if (this.mode === 'history') {
    fragment = this.clearSlashes(
      decodeURI(window.location.pathname + window.location.search)
    );
    fragment = fragment.replace(/\?(.*)$/, '');
    fragment = this.root !== '/' ? fragment.replace(this.root, '') : fragment;
  } else {
    const match = window.location.href.match(/#(.*)$/);
    fragment = match ? match[1] : '';
  }
  return this.clearSlashes(fragment);
};
```

```
navigate = (path = '') => {
  if (this.mode === 'history') {
    window.history.pushState(null, null, this.root + this.clearSlashes(path));
  } else {
    window.location.href = `${window.location.href.replace(
      /#(.*)$/,
      ''
    )}#${path}`;
  }
  return this;
};
listen = () => {
  clearInterval(this.interval);
  this.interval = setInterval(this.interval, 50);
};
interval = () => {
  if (this.current === this.getFragment()) return;
  this.current = this.getFragment();
  this.routes.some((route) => {
    const match = this.current.match(route.path);
    if (match) {
      match.shift();
      route.cb.apply({}, match);
      return match;
    }
    return false;
  });
};
```

```
}
const router = new Router({
  mode: 'hash',
  root: '/',
});
router
  .add(/main_page/, () => {
    table.innerHTML = '';
    h1.textContent = 'In-Out status';
    table.style.marginRight = '1rem';
    search.style.display = 'none';
    // getData(urlTime);
    getTableHead(resultsTime);
    getTableBody(resultsTime);
  })
  .add(/cameras_status/, () => {
    table.innerHTML = '';
    h1.textContent = 'Cameras status';
    table.style.marginRight = '30rem';
    search.style.display = '';
    // getData(urlCameras);
    getTableHead(resultsCamera);
    getTableBody(resultsCamera);
  })
  .add(/cabinets_status/, () => {
    table.innerHTML = '';
    h1.textContent = 'Cabinets status';
    table.style.marginRight = '30rem';
```

```
    search.style.display = '';
    // getData(urlCabinets);
    getTableHead(resultsCabinets);
    getTableBody(resultsCabinets);
  })
  .add('', () => {
    table.innerHTML = '';
    h1.textContent = 'Personal info';
    table.style.marginRight = '30rem';
    search.style.display = '';
    // getData(urlPersonal);
    getTableHead(resultsPersonal);
    getTableBody(resultsPersonal);
  });
// async function getData(url) {
//   await fetch(`http://192.168.0.101/api/${url}/`, {
//     method: 'GET',
//     mode: 'cors',
//     headers: {
//       'Content-Type': 'application/json',
//     },
//   })
//     .then((response) => {
//       response.json().then((data) => {
//         return data.results;
//       });
//     })
//     .catch((e) => {
```

```
//       console.log(e);
//     });
// }
function createCircle(value, tr) {
  const td = document.createElement('td');
  const circle = document.createElement('div');
  circle.style.width = '1rem';
  circle.style.height = '1rem';
  circle.style.margin = '0 auto';
  circle.style.backgroundColor = value ? 'green' : 'red';
  circle.style.borderRadius = '50%';
  td.appendChild(circle);
  tr.appendChild(td);
  return table.appendChild(tr);
}
function createTh(value, tr) {
  const th = document.createElement('th');
  th.innerHTML = value.toString().replaceAll('_', ' ');
  tr.appendChild(th);
  return table.appendChild(tr);
}
function createTd(value, tr) {
  const td = document.createElement('td');
  td.innerHTML = value.toString().replaceAll(',', ' ');
  tr.appendChild(td);
  return table.appendChild(tr);
}
function getTableHead(data) {
```

```
  const keys = Object.keys(data[0]);
  const tr = document.createElement('tr');
  keys.forEach((key) => {
    createTh(key, tr);
  });
}
function getTableBody(data) {
  data.forEach((item) => {
    const values = Object.values(item);
    const tr = document.createElement('tr');
    values.forEach((value) => {
      if (typeof value === 'object') {
        const newValue = Object.values(value);
        createTd(newValue, tr);
      } else if (typeof value === 'boolean') {
        createCircle(value, tr);
      } else {
        createTd(value, tr);
      }
    });
  });
}
```