Web RTC

Всем привет! Меня зовут Артём, я разработчик, и я хочу рассказать, как сделать аналог Zoom-а. Ну, или не аналог Zoom-а, а сервис для видеоконференций.

Как я до этой идеи дошел?

Я очень люблю учиться, особенно на практике. И последнее время я начал замечать, что все те видеоуроки и статьи, которые я поглощаю, стали очень узко освещать ту или иную технологию, но как её конкретно применить и где – это остается на усмотрение читающего или смотрящего.

И в какой-то момент я подумал, что не видел давно таких видео, где автор рассказывал процесс разработки “от идеи до реализации”, или “от задумки до mvp”.

Например, есть, вот есть у нас библиотека на джаваскрипте, делает она то-то очень быстро, давайте с её помощью сделаем todo-трекер. Или многопользовательский текстовый редактор. Или еще что-нибудь.

Еще одна мысль – кода стало очень много. Когда у меня появляется какая-то идея, я перво наперво делаю что? Правильно, иду в интернет и смотрю, а не сделал ли кто-нибудь что-то такое до меня.

Так вот, тема! Злободневная тема – видеозвонки и работа из дома.

Так почему бы нам не попробовать сделать свои видеозвонки?

Быстро. За пару вечеров. За один вечер!

Да еще и чтобы по-настоящему, чтобы работало.

И чтобы можно было реально созвониться с друзьями, а не только в localhost песочнице поиграться.

Ну, и еще стоит оговориться, что я разработкой уже достаточно давно занимаюсь, кое-какие навыки у меня уже есть, я пробовал до этого писать приложения и у меня получалось.

Поехали.

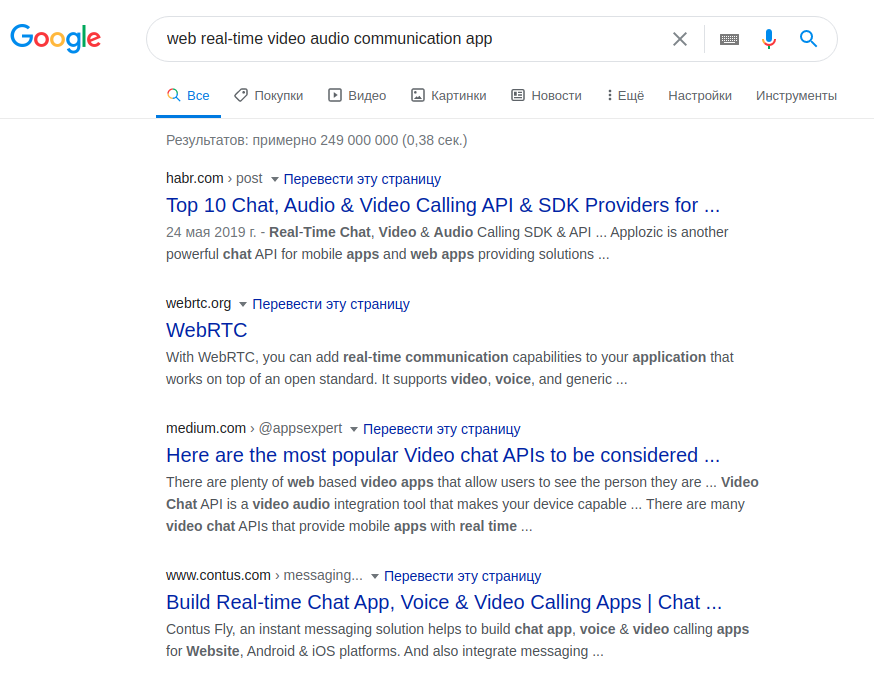
Сначала мы идем в Google и ищем там что-нибудь подходящее.

Что-то связанное с видеосвязью, real-time communication. Сузим поиск еще и тем, что будем делать наше приложение в браузере. Так будет быстрее всего сделать приложение, которым можно поделиться с друзьями.

Ну, потому что у всех есть браузер. Обычно.

**Идем в google, пишем там:**

**web real-time video audio communication app**



Опа, что-то есть, вторая ссылка. Про WebRTC я что-то слышал, посмотрим, что там.

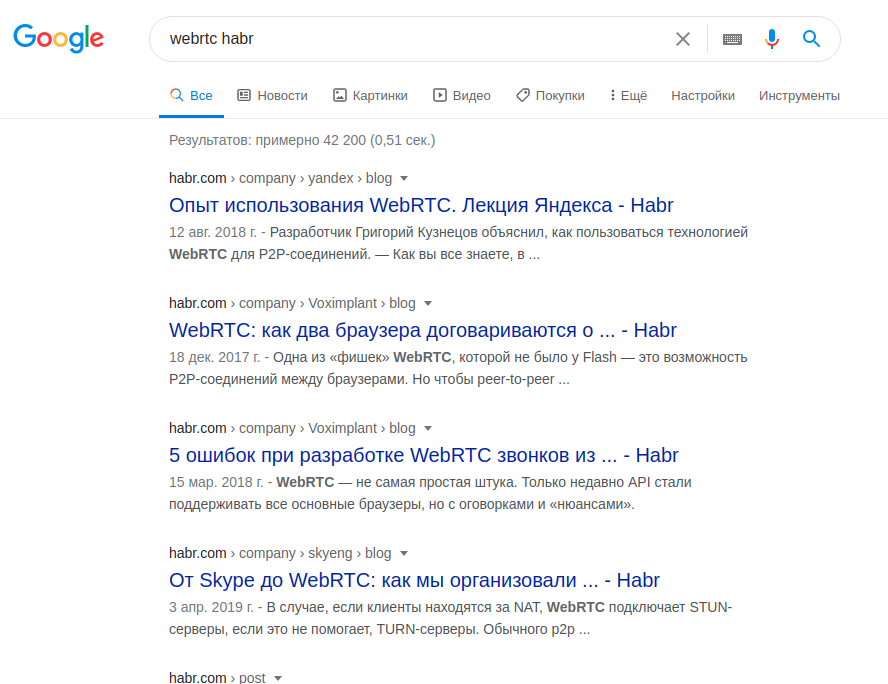
With WebRTC, you can add real-time communication capabilities, блабла, video audio блабла, javascript apis in all major browsers, такс, и делает это всё Google, а поддерживают Apple, MS, Mozilla и еще много кто.

https://webrtc.org/

Так, чудненько, надо бы теперь почитать, а что это вообще такое, наверняка что-то на хабре есть.

**Идем в google, пишем там:**

**webrtc habr**



Так, лекция яндекса, смотрим что там.

<https://habr.com/ru/company/yandex/blog/419951/>

Блаблабла, peer to peer, Adobe Flash, Voximplant, Google Hangouts, Facebook Messenger, блабла, SDP-диаграмма, маршрутизация, ICE Internet connectivity Establishment, STUN TURN блабла, WebRTC работает только по HTTPS, ваш сайт должен быть подписан сертификатом.

Хорошая статья, подробная. Про HTTPS нужно запомнить, нужно будет подписывать сайт валидным сертификатом.

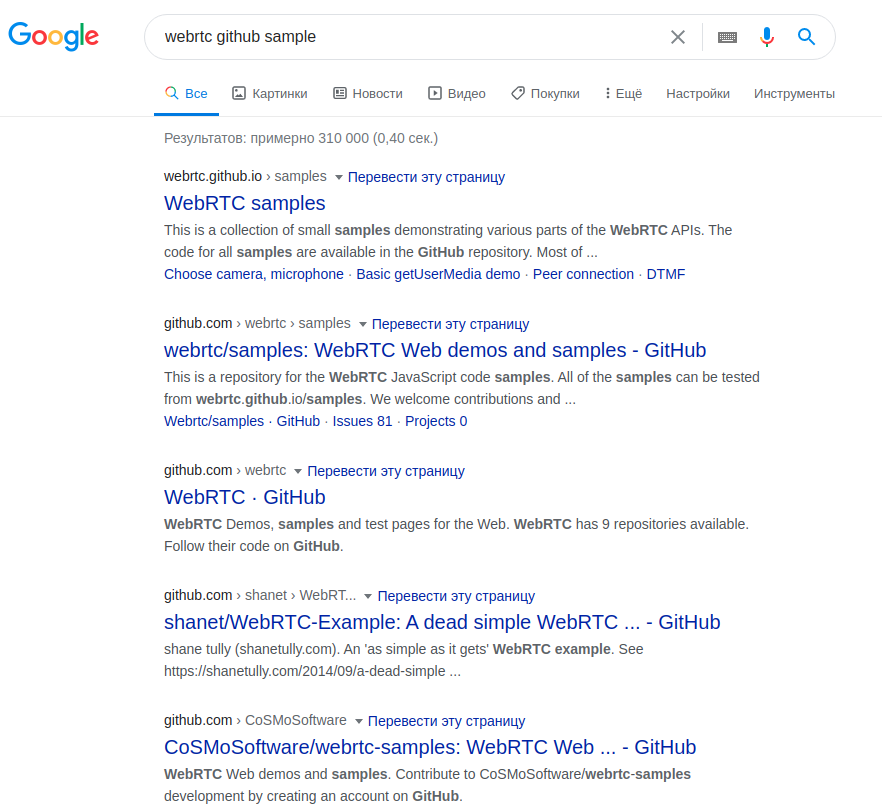
<https://habr.com/ru/company/Voximplant/blog/351234/>

Так, раз такое дело, надо бы поискать, может кто-то уже делал тестовое приложение.

Пошли поищем, желательно с исходниками.

**Идем в google, пишем там:**

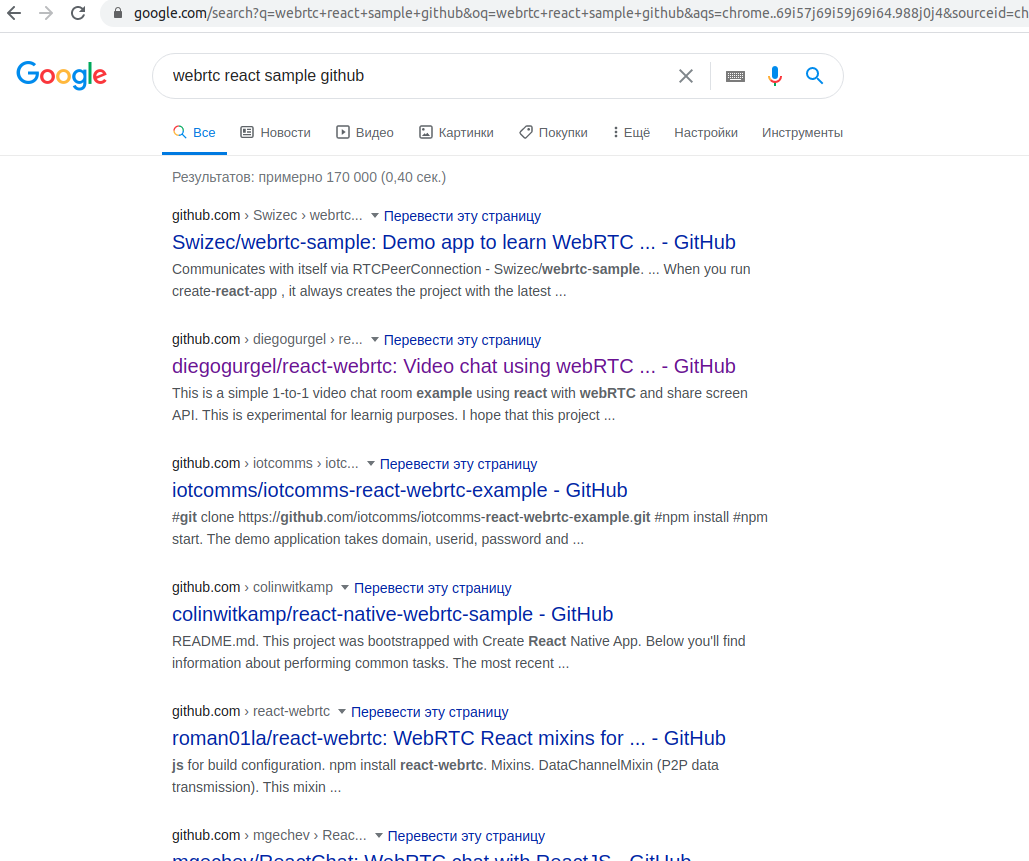
**webrtc sample github**



Так, тут что-то очень общее, нам надо поконкретнее, давай по-другому поищем. На react-е например.

**Идем в google, пишем там:**

**webrtc react sample github**



Вторая ссылка на гитхаб, смотрим туда.

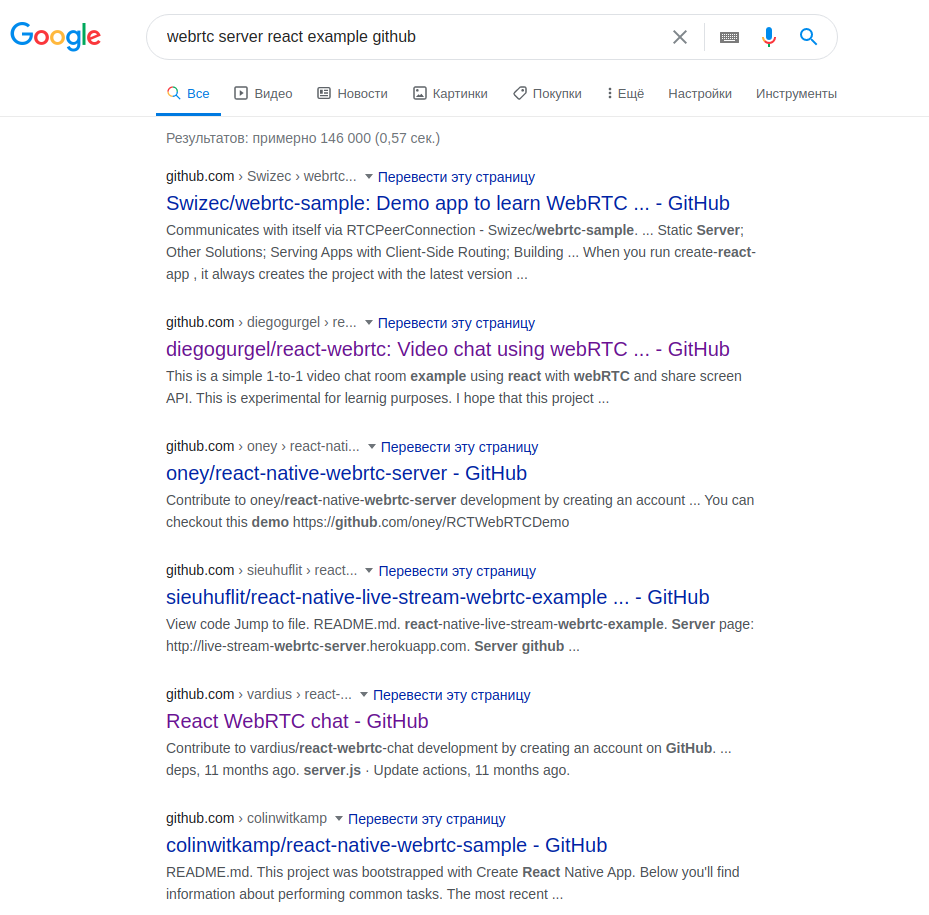
<https://github.com/diegogurgel/react-webrtc>

Пример хороший, но тут написано что он для 1to1 связи, а нам нужно чтобы конференция была, мы же хотим с компанией друзей общаться.

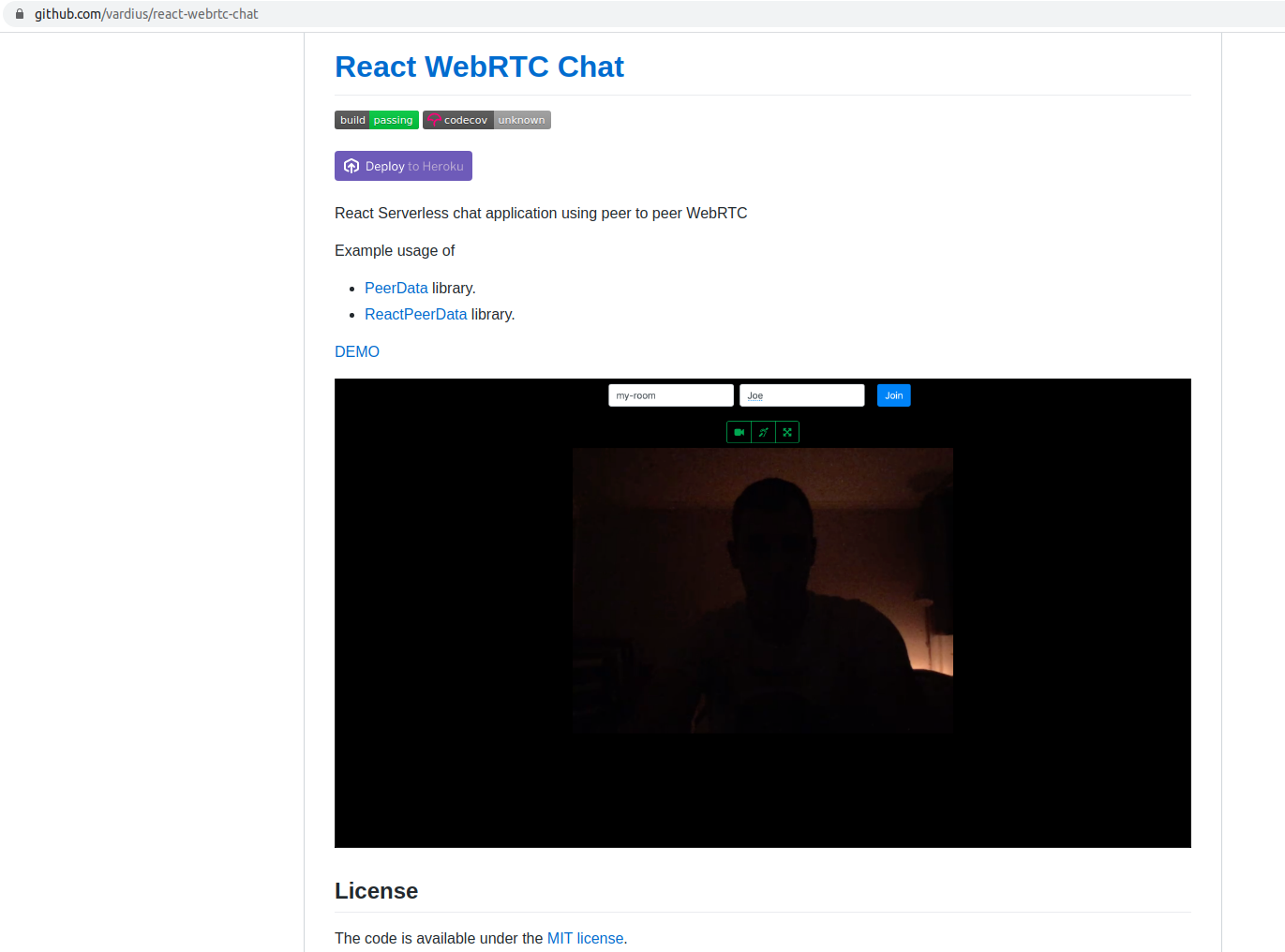
Давай еще изменим поисковый запрос.

**Идем в google, пишем там:**

**webrtc server react chat example github**



Вот, пятая ссылка вроде подходит, давай посмотрим что там.



Во, тут и MIT лицензия, и демка, и библиотека, и сервер, и картинка даже вставлена.

Годится, пробуем запустить.

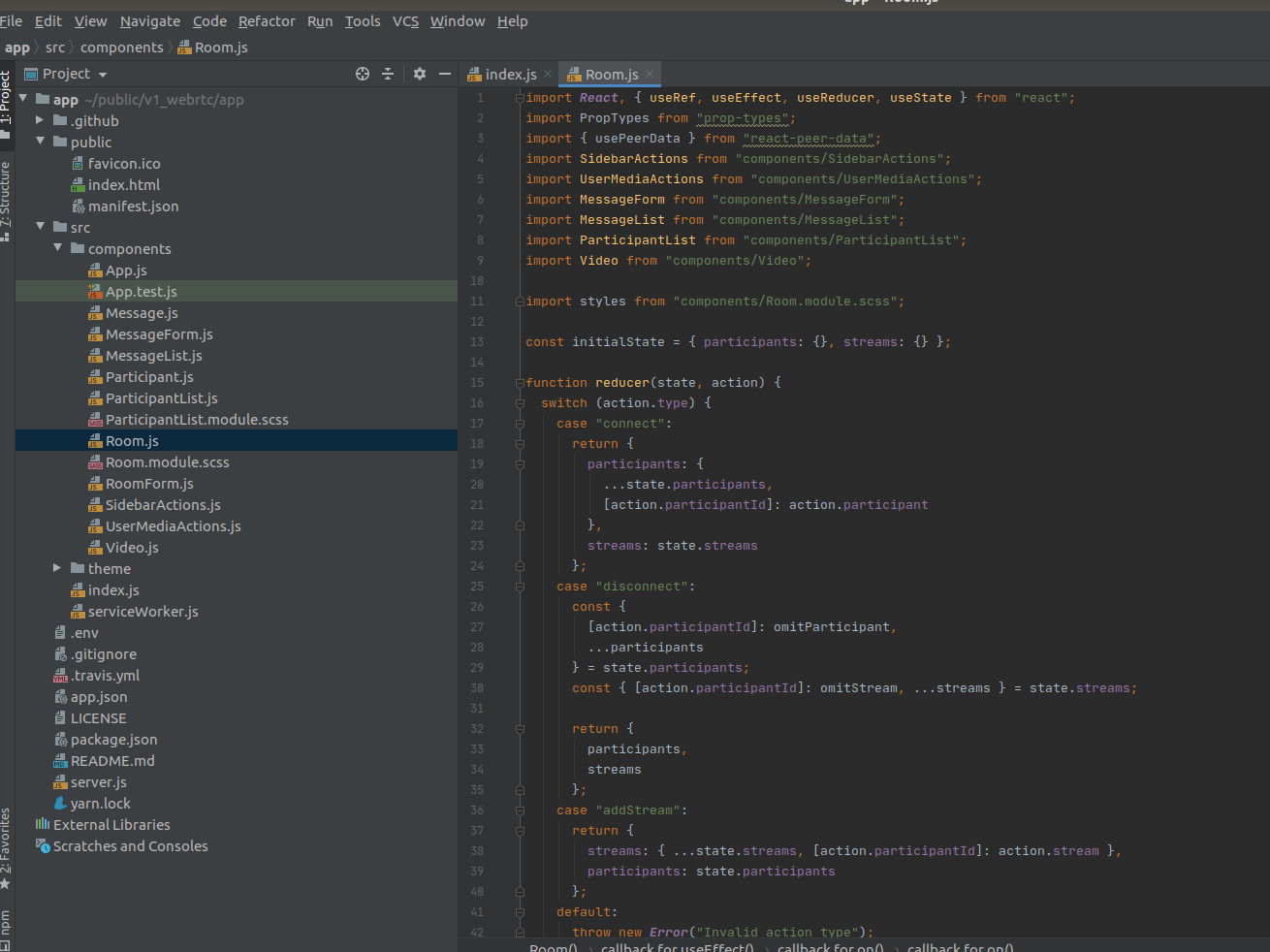
<https://github.com/vardius/react-webrtc-chat>

<https://github.com/vardius/peer-data>

<https://github.com/vardius/react-peer-data>

<https://github.com/vardius/peer-data-server>

Клонируем, открываем проект.



Сначала заходим в package.json, посморим, что к проекту подключено.

Так, есть скрипты для запуска сервера, есть для фронта, вроде всё нормально, восстанавливаем пакеты.

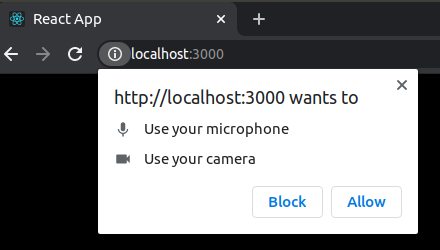
Выполняем команду yarn install или npm i.

Попробуем собрать...

Вызываем npm run build

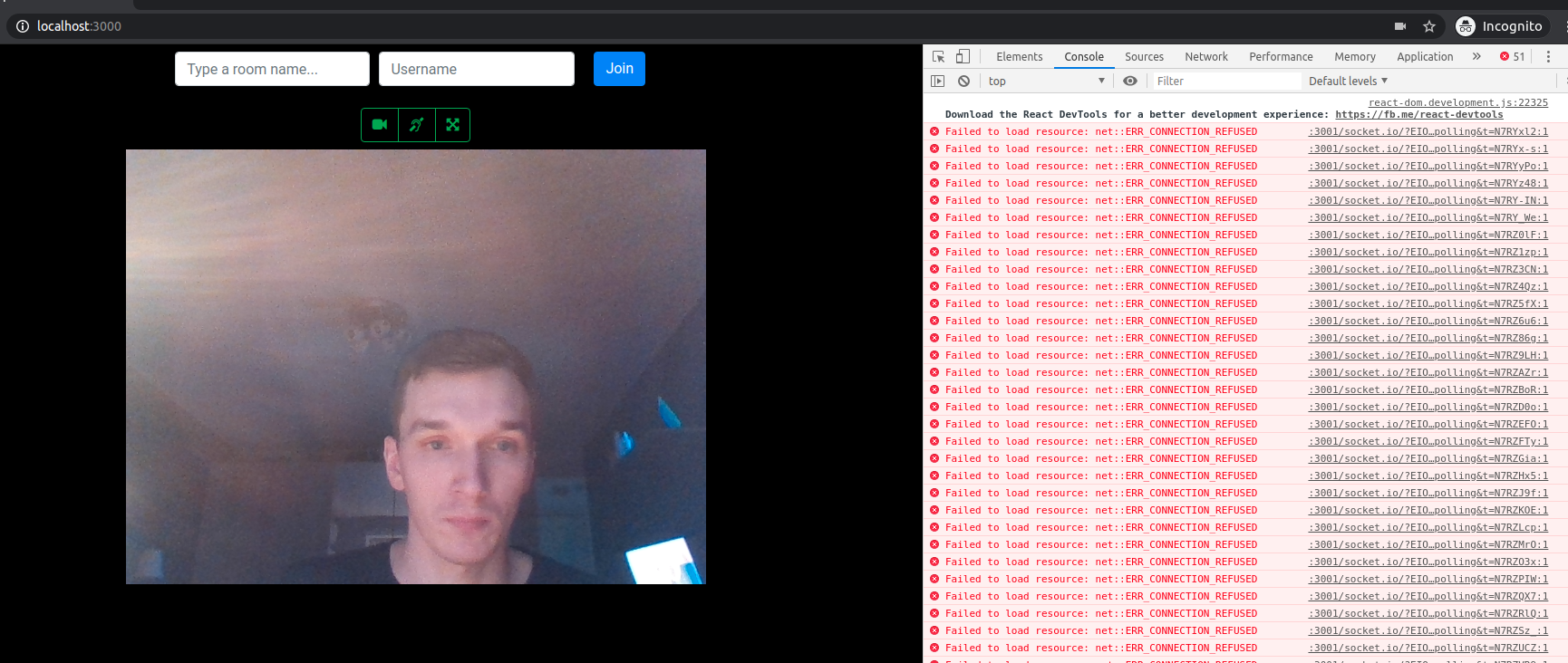
А теперь попробуем запустить...

npm run dev

Так, вроде запустилось, в консоли пишет что на 3000 порту работает, пошли посмотрим.

Разрешаем доступ к микрофону, смотрим дальше.

Камера показывает, кнопочки есть, посмотрим в консоль



Так, тут видно, что страница пытается пойти по адресу localhost:3001, но там connection refused.

Это мы забыли серверную часть запустить, давай запустим сервер и попробуем страничку обновить.

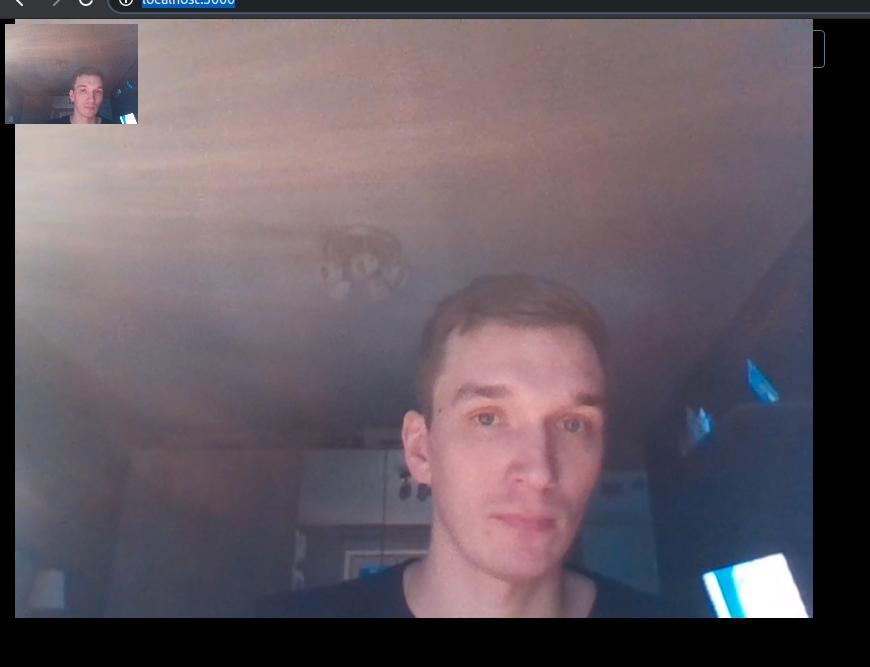
Запускаем терминал, запускаем сервер

node server.js

Обновляем страничку, смотрим в консоль.

Во, ошибки пропали, смотрим дальше, попробуем дальше, запустим две вкладки, подключимся к чату.

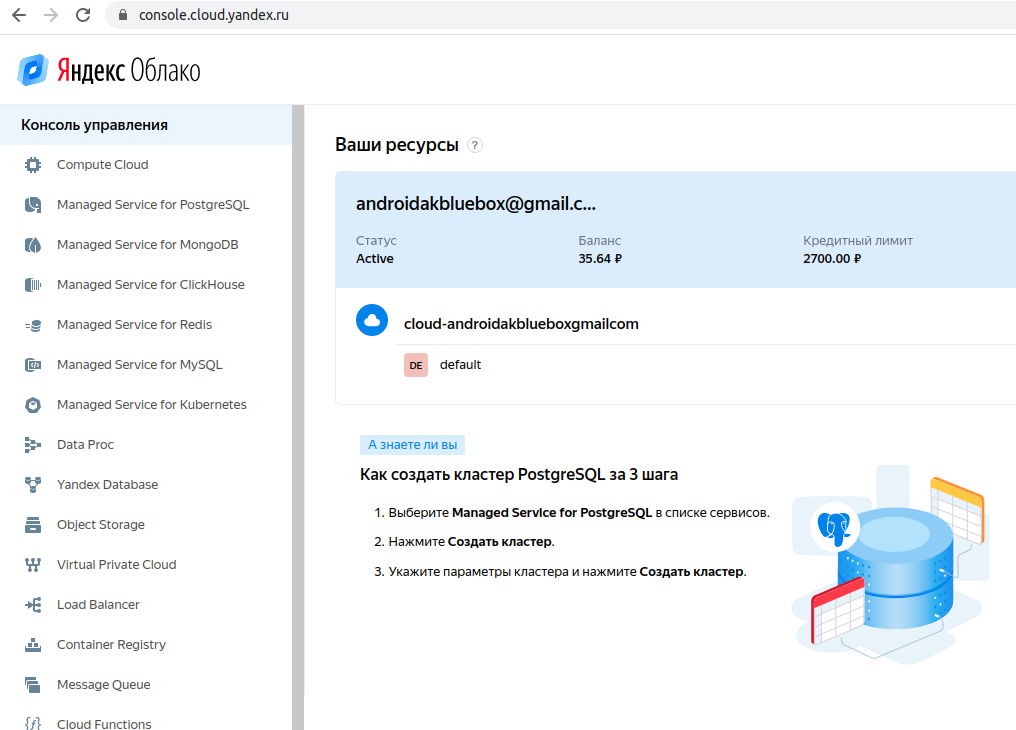
Работает!



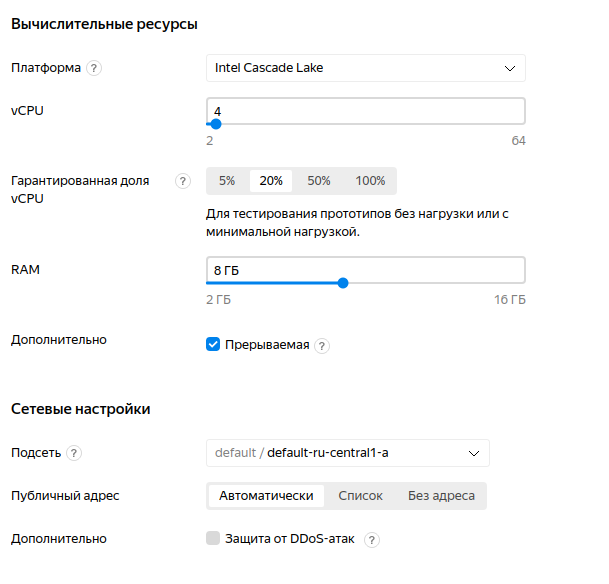
Ну что, дальше будем всё это дело куда-то публиковать, а то на localhost-е мы ни с кем этим приложением не поделимся.

Предлагаю опубликоваться в Яндекс.Облако, у меня там уже аккаунт есть.

Идем в консоль, там заводим виртуальную машину в Compute Cloud.



Создаем виртуалку, заполняем там всякие поля, ставим галочку “прерываемая” чтобы подешевле было.

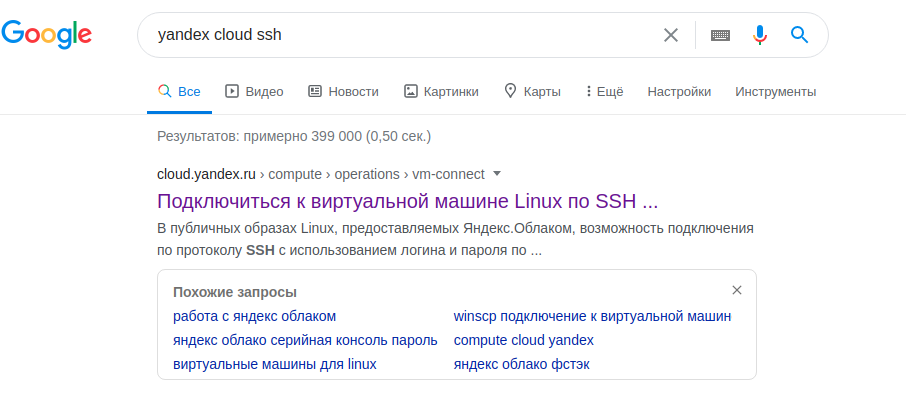


Так, а тут надо добавить ключ, чтобы потом можно было по ssh подключиться.

На всякий случай загуглим, чтобы не ошибиться

**Идем в google, пишем там:**

**yandex cloud ssh**



Вот, первая ссылка – и сразу же инструкция.

Идем по шагам:

~~~

Откройте терминал.

Создайте новый ключ с помощью команды ssh-keygen:

$ ssh-keygen -t rsa -b 2048

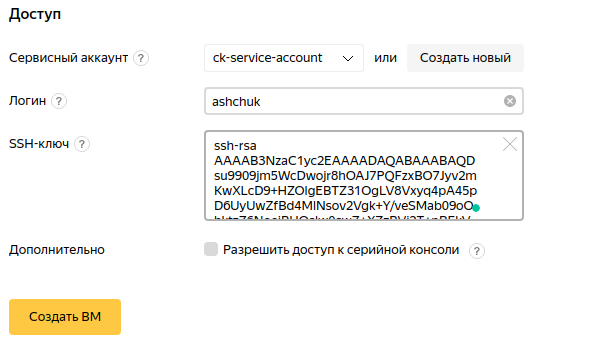
После выполнения команды вам будет предложено указать имена файлов, в которые будут сохранены ключи и ввести пароль для закрытого ключа. По умолчанию используется имя id\_rsa, ключи создаются в директории ~./ssh.

Публичная часть ключа будет сохранена в файле <имя\_ключа>.pub. Вставьте эту часть ключа в поле SSH-ключ при создании новой виртуальной машины через консоль управления.

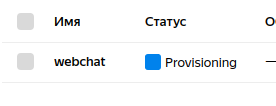
Заполняем, жмем на кнопку “создать ВМ”, ждем.

Дождались, можно подключаться.

Идем в консоль, делаем там



Так, создали, смотрим.



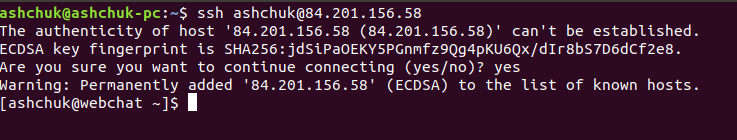
Создается виртуалочка, ждем дальше.



Попробуем подключиться.

*ssh ashchuk@84.201.156.58 (это публичный ip адрес машины)*

Так, подключились успешно



Так, дальше можно публиковаться.

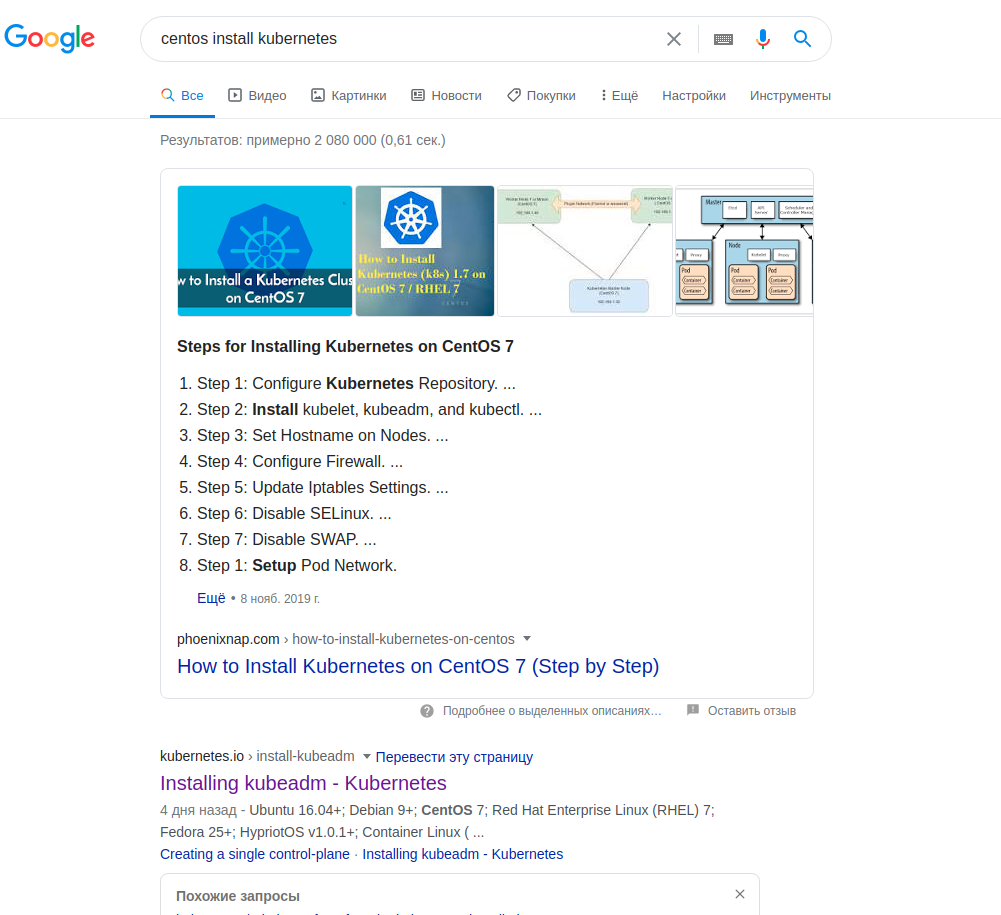
Предлагаю запаковать всё в docker и собранные образы отправить в docker registry яндекса.

Ну, и чтобы поинтересней было, давай всё это кубернетисом опубликуем.

Ставим kubectl kubeadm kubelet на виртуалку.

**Идем в google, пишем там:**

**centos install kubernetes**

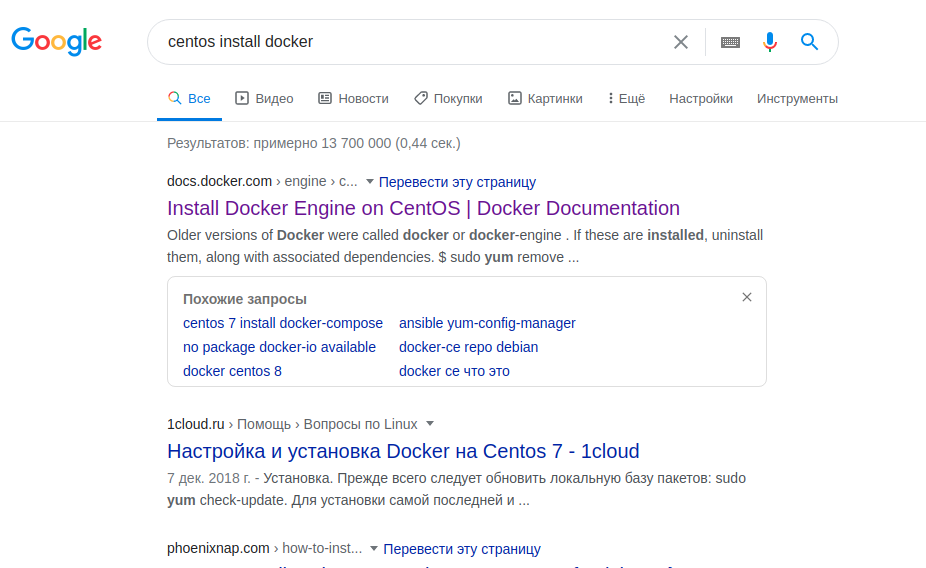


Вторая ссылка – на официальную документацию. Туда и идем.

Так, первый пункт – установить docker. Значит ставим docker.

**Идем в google, пишем там:**

**centos install docker**



Идем по шагам, по инструкции.

*$ sudo yum install -y yum-utils*

*$ sudo yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo*

*$ sudo yum install docker-ce docker-ce-cli containerd.io*

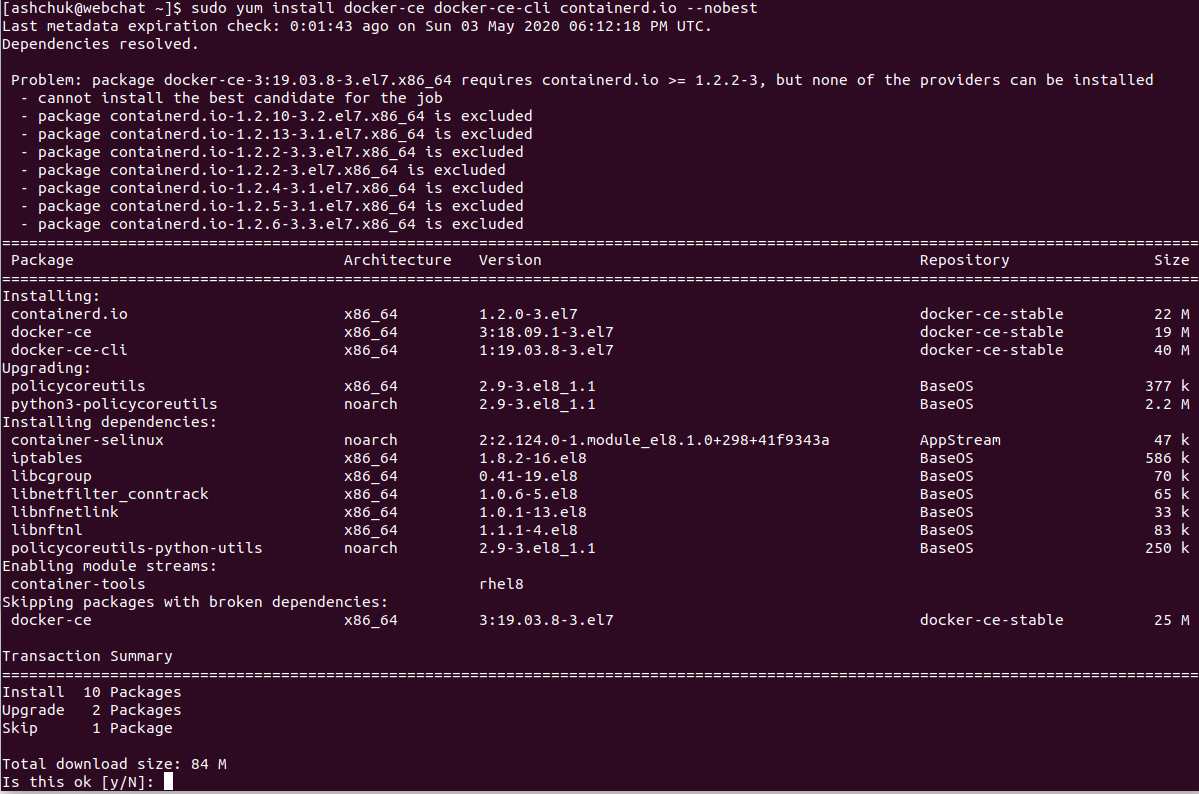
**И… не работает.**

*Problem: package docker-ce-3:19.03.8-3.el7.x86\_64 requires containerd.io >= 1.2.2-3, but none of the providers can be installed*

Так, и тут же он пишет, что пакет такой в репозитории есть, но идеального кандидата для установки нет.

Ну ничего, попробуем неидеальный вариант, добавляем флаг –nobest, посмотрим что будет предложено.

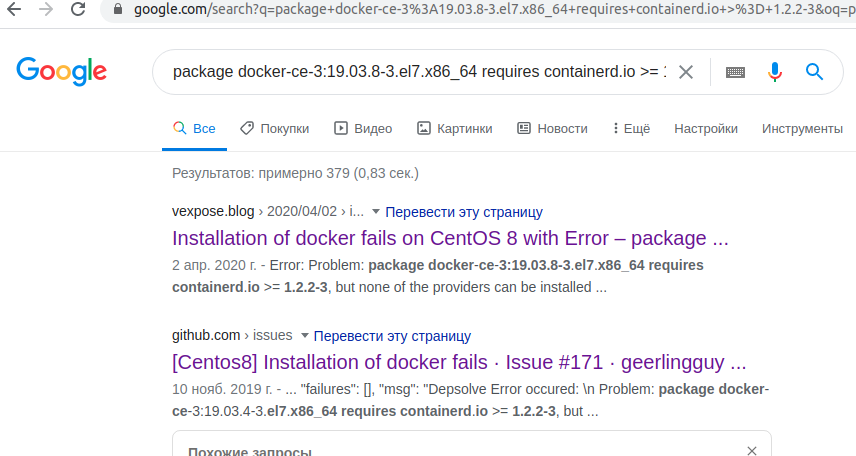
$ sudo yum install docker-ce docker-ce-cli containerd.io --nobest



Так, ну я тут вижу что версии немного разнятся, что-то я не уверен что это хорошая идея, попробуем загуглить.

**Идем в google, пишем там:**

**package docker-ce-3:19.03.8-3.el7.x86\_64 requires containerd.io >= 1.2.2-3**

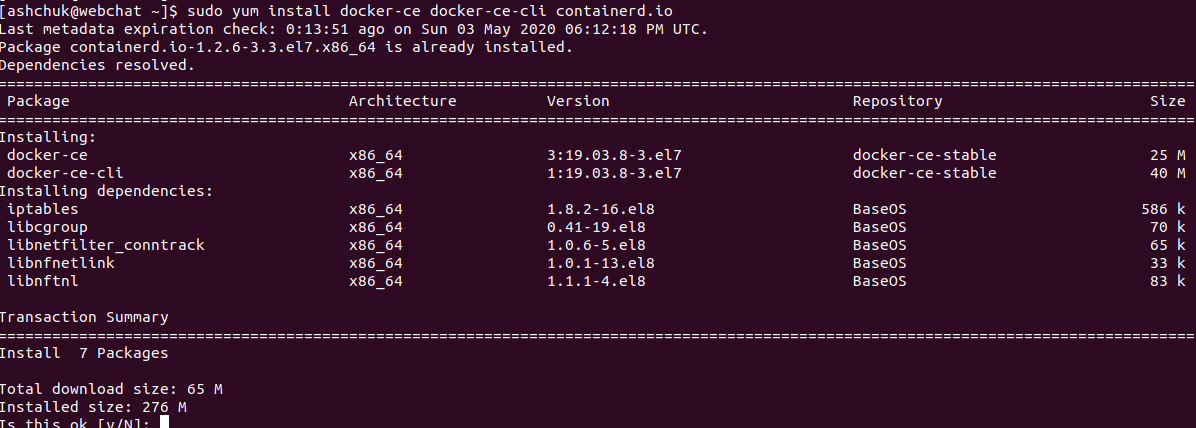


Первая ссылка, там пишут что можно вручную поставить нужную версию containerd

Ставим, и повторяем предыдущую попытку

*$ yum install -y https://download.docker.com/linux/centos/7/x86\_64/stable/Packages/containerd.io-1.2.6-3.3.el7.x86\_64.rpm*

Вот, теперь нормально прошло.



Так, docker поставили, идем дальше по kubernetes-у.

Утанавливаем нужные системные переменные в нужное состояние

*$ cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf*

*net.bridge.bridge-nf-call-ip6tables = 1*

*net.bridge.bridge-nf-call-iptables = 1*

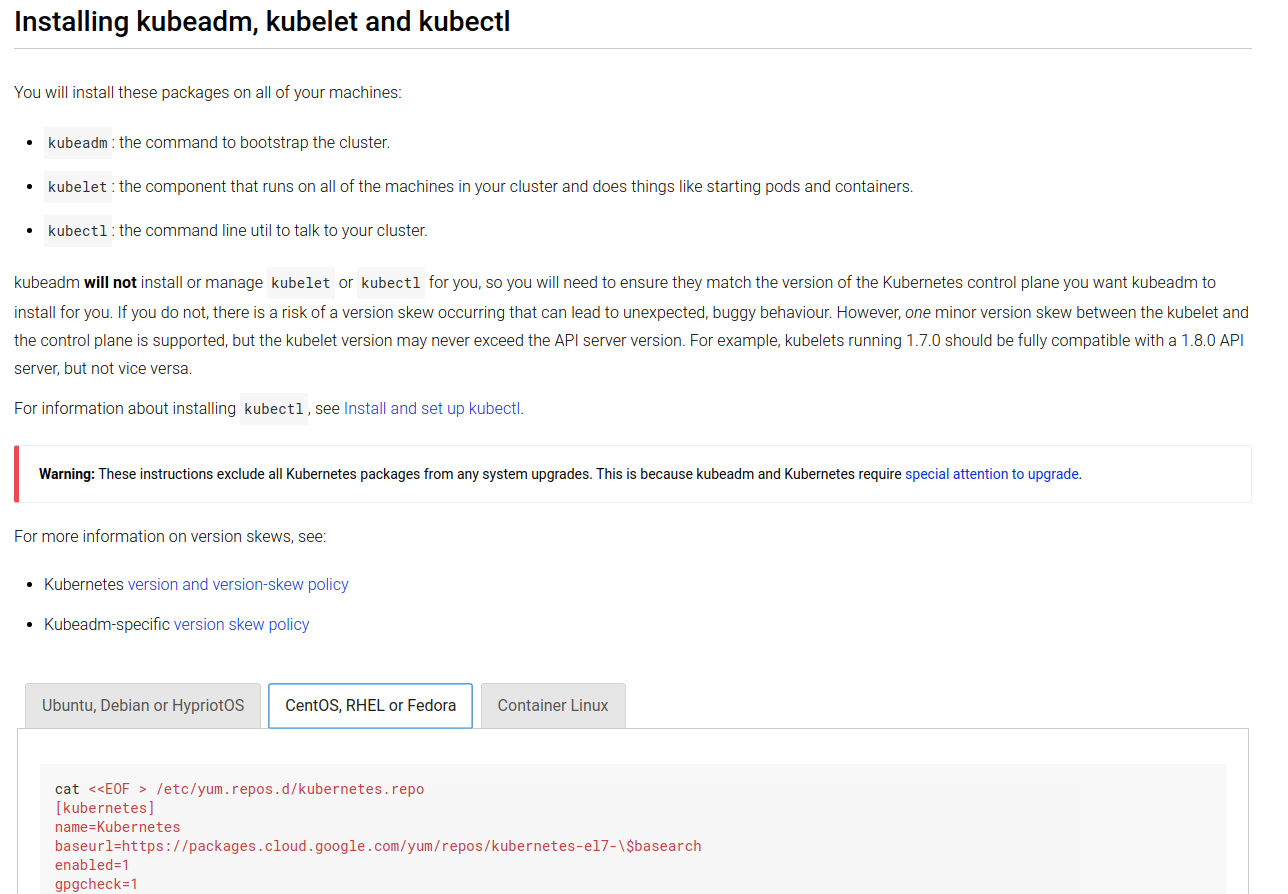
*EOF*

*$ sudo sysctl –system*

*Загружаем модуль br\_netfilter*

*$ sudo modprobe br\_netfilter*

Дальше устанавливаем бинарники кубика:



Добавляем ссылку на репозиторий, выключаем selinux, ставим пакеты:

*$ sudo cat <<EOF > /etc/yum.repos.d/kubernetes.repo*

*[kubernetes]*

*name=Kubernetes*

*baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-\$basearch*

*enabled=1*

*gpgcheck=1*

*repo\_gpgcheck=1*

*gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg*

*exclude=kubelet kubeadm kubectl*

*EOF*

*# Set SELinux in permissive mode (effectively disabling it)*

*$ setenforce 0*

*$ sed -i 's/^SELINUX=enforcing$/SELINUX=permissive/' /etc/selinux/config*

*$ yum install -y kubelet kubeadm kubectl --disableexcludes=kubernetes*

*$ systemctl enable --now kubelet*

И… не работает, что-то не так с метадатой репозитория. Странно, до этого работало, может попробовать в предыдущей версии документации посмотреть?

Да, там другой baseurl указан, попробуем его:

*$ cat <<EOF > /etc/yum.repos.d/kubernetes.repo*

*[kubernetes]*

*name=Kubernetes*

*baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86\_64*

*enabled=1*

*gpgcheck=1*

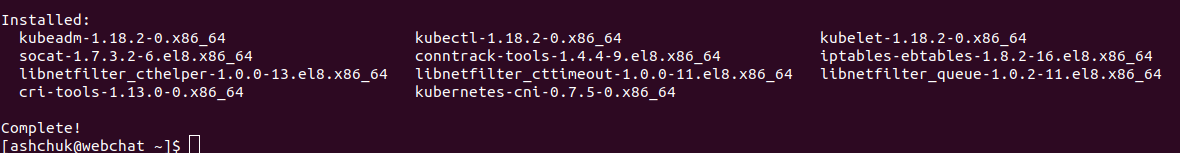
*repo\_gpgcheck=1*

*gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg*

*EOF*

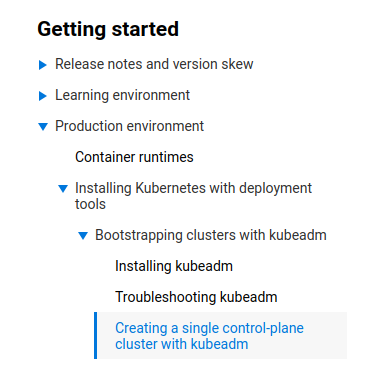
*$ sudo yum install -y kubelet kubeadm kubectl --disableexcludes=kubernetes*

Во, теперь пошло дело, и последняя версия подтянулась, значит всё правильно сделали



Поехали дальше.

Здесь же в документации вижу интсрукцию по установке.



Чтобы развернуть kubernetes ноду нам нужна утилита kubeadm, я сейчас сразу заострю внимание на одном флаге, который указывается при инициализации ноды.

Смотрим документацию, видим здесь вызов команды:

*$ kubeadm init <args>*

Вот этой командой мы и поднимаем ноду, и важно указать флаг **--pod-network-cidr=192.168.0.0/16**, об этом здесь же в документации говорится ниже.

Поехали, инициализируем мастер ноду

*$ kubeadm init --pod-network-cidr=192.168.0.0/16*

**И… не работает,** потому что мы забыли запустить docker-daemon

Запускаем его и повторяем попытку

*$ sudo systemctl enable docker*

*$ sudo systemctl start docker*

*$ sudo systemctl status docker*

*$ kubeadm init --pod-network-cidr=192.168.0.0/16*

Вот теперь пошло дело.

Дальше ждем, пока запустятся системный контейнеры.

*$ kubectl get pods --all-namespaces*

Видим, что dns контейнеры подвисли в состоянии Pending. Это всё почему? Правильно, потому что мы cni не добавили.

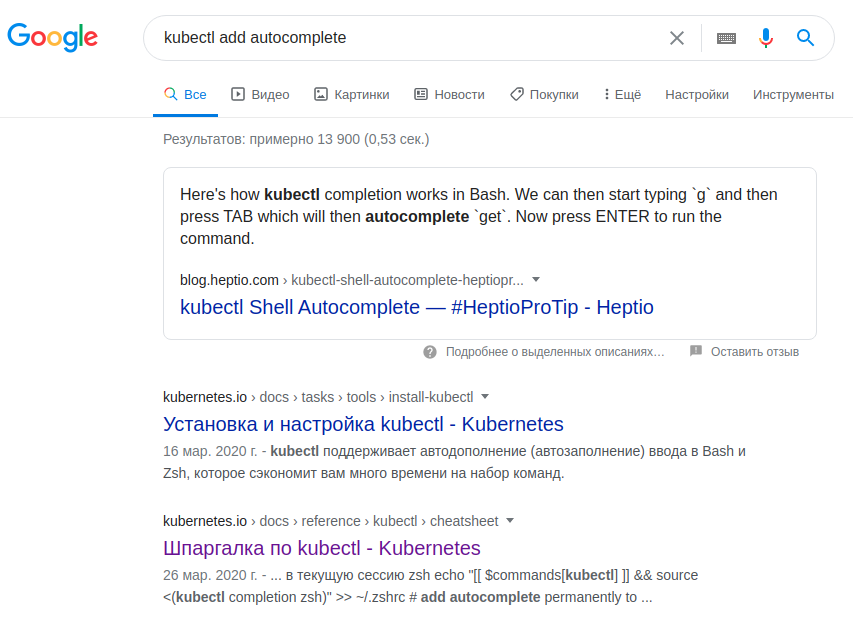
Добавляем cni:

*$ kubectl apply -f https://docs.projectcalico.org/v3.11/manifests/calico.yaml*

И еще для того, чтобы у нас консоль умела автодополнение ввода, сделаем вот что:

**Идем в Google, пишем там**

**kubectl add autocomplete**



Идем в шпаргалки, находим там такие команды

*$ sudo yum install bash-completion*

*$ source <(kubectl completion bash)*

*$ echo "source <(kubectl completion bash)" >> ~/.bashrc*

Теперь по нажатию на Tab у нас будут автоматически дополняться команды в консоли. Красота, пальцы об клавиатуру теперь не стирать.

Что мы делаем дальше. Дальше нам нужно вытащить наше приложение наружу, если мы сейчас попробуем обратиться на публичный адрес виртуальной машины в браузере, то мы ничего в ответ не получим. А нам надо бы.

Запуститься-то оно запустилось, а вот работает ли?

Посмотрим в dns, опа, а там ошибки. Погуглим про это.

**Идем в Google, пишем там:**

**kubernetes HINFO: read udp i/o timeout**

Четвертая ссылка.

<https://github.com/kubernetes-sigs/kubespray/issues/4674>

Советуют сделать так:

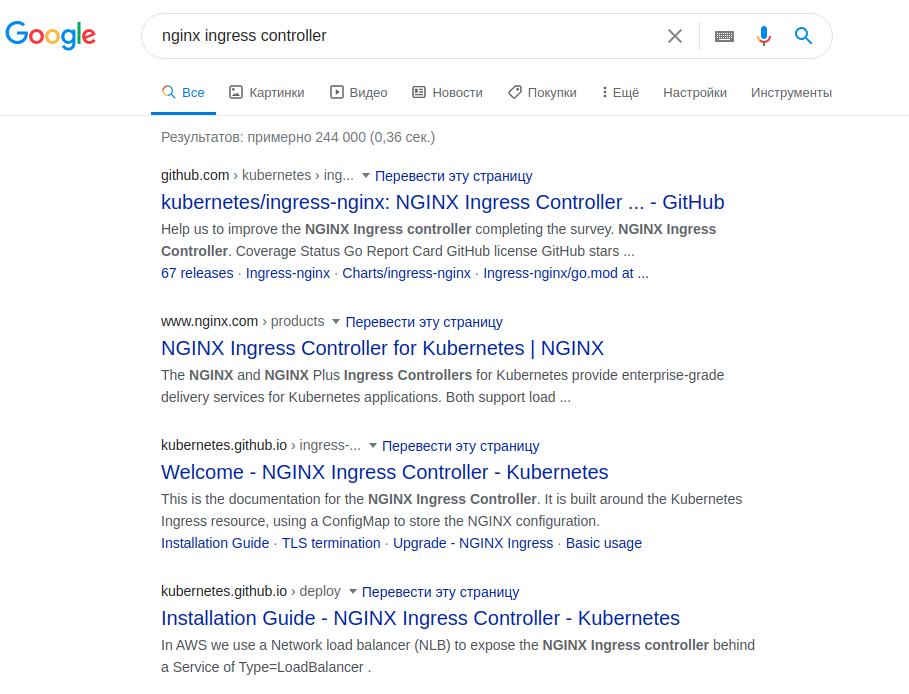
*$ sudo iptables -P FORWARD ACCEPT*

Сделали, перезапускаем контейнеры. Починилось! Можно идти дальше.

Для этого ставим nginx, точнее nginx ingress controller.

**Идем в Google, пишем там:**

**nginx ingress controller**



Третья ссылка – на официальную документацию. Идем туда, смотрим раздел deployment, ищем там bare-metal

*$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-0.32.0/deploy/static/provider/baremetal/deploy.yaml*

Теперь выполняем следующее, для того чтобы забиндить 80 и 443 порт виртуалки:

*$ sudo vi /etc/kubernetes/manifests/kube-apiserver.yaml*

Поправляем там конфигурацию, добавляем флаг --service-node-port-range=80-32767

*$ kubectl edit service -n ingress-nginx ingress-nginx-controller*

Поправляем номера **nodePort** портов на 80 и 443, сохраняем изменения

Теперь по ip адресу виртуальной машины нам вернется дефолтная страница nginx

**И… не работает.**

Всё потому что по умолчанию на мастер ноде нельзя запускать никаких компонентов, кроме системных. Но это можно поправить. Выполняем в консоли команду:

*$ kubectl taint nodes --all node-role.kubernetes.io/master-*

для наглядности можно посмотреть как контейнеры создаются

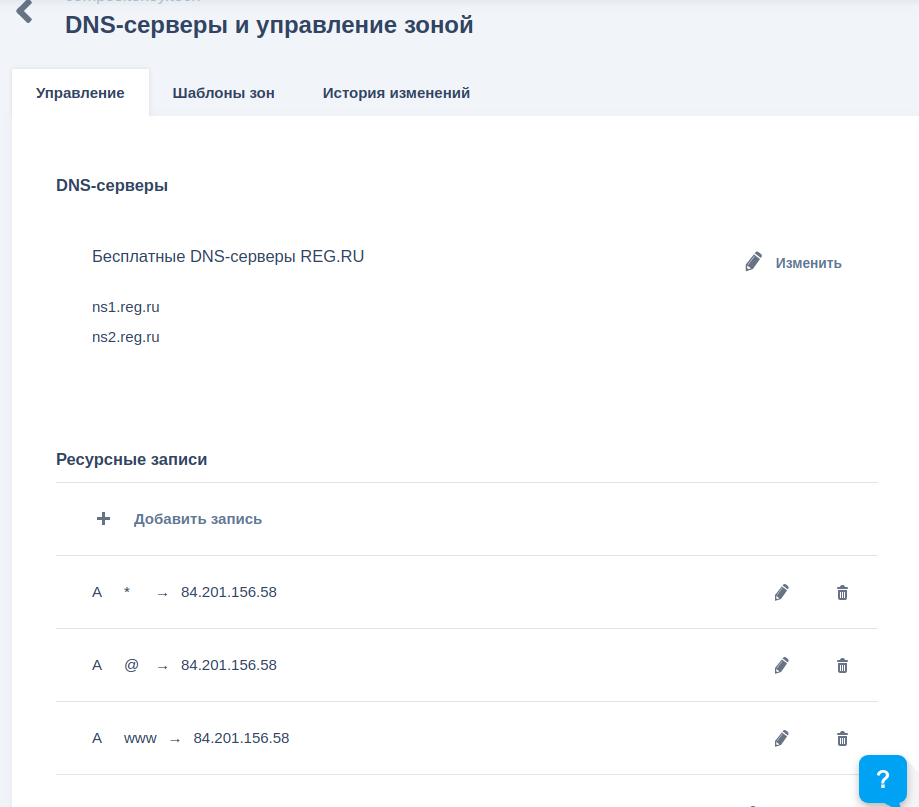
*$ kubectl get pods -n ingress-nginx -l app.kubernetes.io/name=ingress-nginx --watch*

Ждем немного и вуаля – теперь возвращается “404 Not Found”



И еще я бы сразу подключил доменное имя к этому ip адресу, у меня после хакатона осталось одно, вот его и будем использовать.

У меня домен зарегестирован на reg.ru, идем туда, открываем настройки домена и вписываем туда ip адрес нашей виртуальной машины.



Так, теперь по идее если мы пойдем по адресу **compositekey.tech**, то мы увидим то же самое, что и при обращении к ip адресу виртуалки.

**И… работает.**

Так, отлично, теперь нам нужно запихнуть наш видео чат сюда, на виртуальную машину.

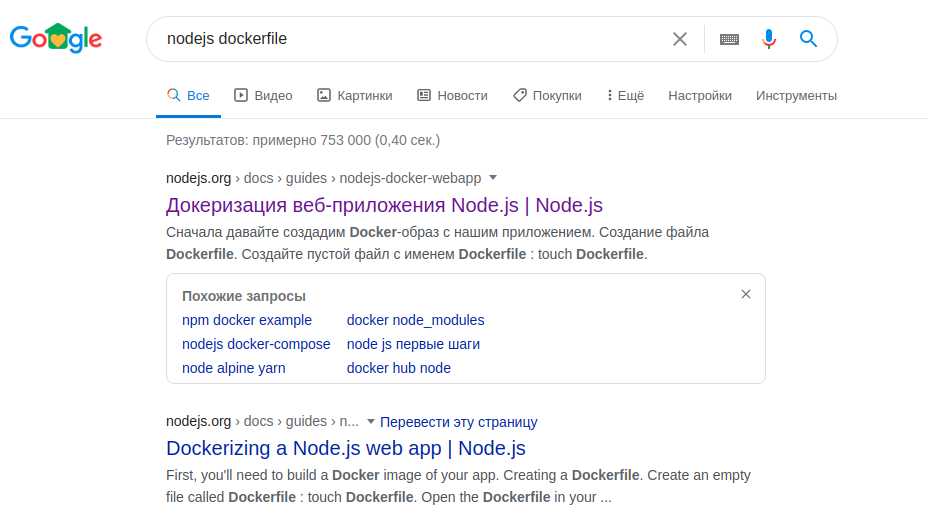
Для начала нам нужно запаковать наш фронтенд и бекенд в контейнеры.

Бекенд, я так понимаю, у нас на nodejs, а фронтенд на react.

Значит так и запаковываем в docker, начнем с бекенда.

**Идем в Google, пишем там:**

**nodejs dockerfile**



Первая ссылка – на официальный пример. Идем туда.

https://nodejs.org/ru/docs/guides/nodejs-docker-webapp/

И идем так же, по шагам.

Создаем файл Dockerfile, только добавим к нему server, потому что нам для фронтенда нужно будет тоже составить такой файл.

*FROM node:12*

*# создание директории приложения*

*WORKDIR /usr/src/app*

*# установка зависимостей*

*# символ астериск ("\*") используется для того чтобы по возможности*

*# скопировать оба файла: package.json и package-lock.json*

*COPY package\*.json ./*

*RUN npm install*

*# Если вы создаете сборку для продакшн*

*# RUN npm ci --only=production*

*# копируем исходный код*

*COPY . .*

*EXPOSE 8080*

*CMD [ "node", "server.js" ]*

Так, здесь нам нужно обратить внимание на директиву EXPOSE, это указание порта, который будет слушать контейнер и нам нужно, чтобы этот порт совпадал с тем, который указан в коде приложения.

Идем в server.js, смотрим что там у нас указано.

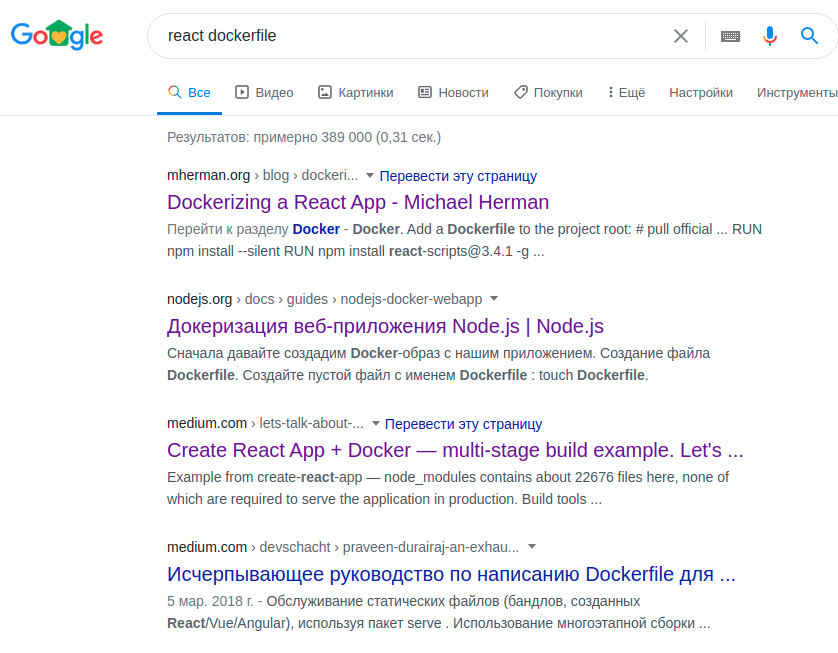
Видим, что порт указан в переменной DEFAULT\_PORT и выставляется в зависимости от переменной среды NODE\_ENV. Давай сделаем так, чтобы здесь был один порт 8080, для простоты.

И вижу еще в index.js такой же код, который в зависимости от NODE\_ENV устанавлявает адрес бекенда, поменяем здесь адрес на тот, который у нас будет использоваться в интернете, но чуть позже.

Такс, теперь делаем такой же файл для фронтенда.

**Идем в Google, пишем там:**

**react dockerfile**



Открываем первую и третью ссылку.

В первой есть что-то похожее на правду, попробуем так.

Создаем файл Dockerfile.client, в нем пишем следующее:

*# build environment*

*FROM node:13.12.0-alpine as build*

*WORKDIR /app*

*ENV PATH /app/node\_modules/.bin:$PATH*

*COPY package.json ./*

*COPY package-lock.json ./*

*RUN npm ci --silent*

*RUN npm install react-scripts@3.4.1 -g --silent*

*COPY . ./*

*RUN npm run build*

*# production environment*

*FROM nginx:stable-alpine*

*COPY --from=build /app/build /usr/share/nginx/html*

*EXPOSE 80*

*CMD ["nginx", "-g", "daemon off;"]*

Отлично, теперь давай попробуем собрать серверный и клиентский контейнер.

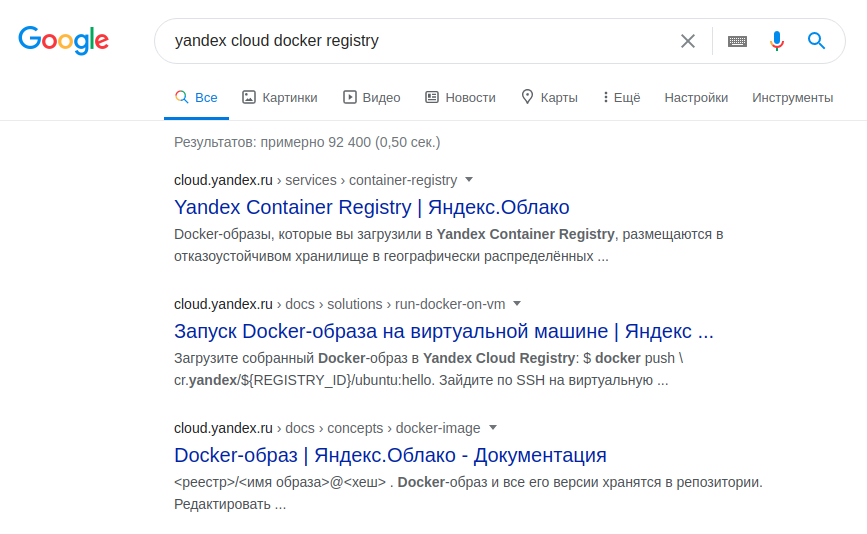
*$ docker build -f Dockerfile.client .*

*$ docker build -f Dockerfile.server .*

Так, и пока контейнеры собираются, пойдем посмотрим, куда их можно будет сложить, чтобы доставить до нашей виртуальной машины. Для этого нам нужен docker registry яндекса.

**Идем в Google, пишем там:**

**yandex cloud docker registry**



Первая ссылка, идем туда.

Блаблабла, сервис для хранения, идем в документацию.

Так, начало работы. Пишут, что понадобится yandex cli и docker. Ну, docker у меня уже установлен, а вот yandex cli еще нет, давай его поставим,

*$ curl https://storage.yandexcloud.net/yandexcloud-yc/install.sh | bash*

потом делаем

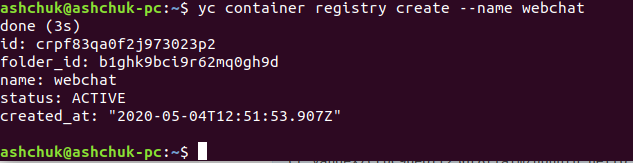
*$ yc init*

Идем по ссылке, получаем токен, авторизуемся

Дальше создаем себе container registry

*$ yc container registry create --name webchat*

Получаем id и folder\_id, они нам, видимо, понадобятся чтобы отправлять наши контейнеры в regestry.



Дальше сконфигурируем docker для использования только что созданного registry

*$ yc container registry configure-docker*

Давай теперь отправим наши собранные контейнеры в яндекс.

Получаем список наших образов, которые мы предварительно собрали

*$ docker images -a*

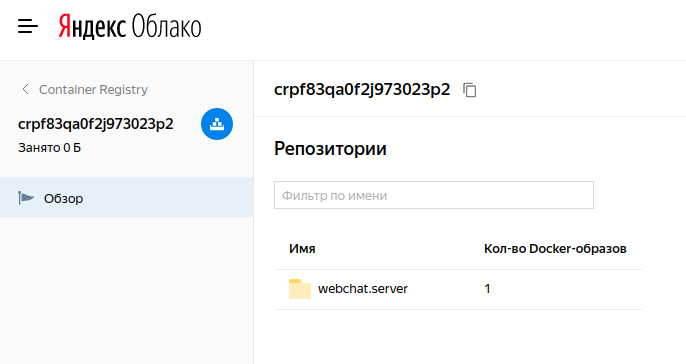
помечаем их id-шниками, которые получили при регистрации в registry

*$ docker tag 9edd80668a61 cr.yandex/crpf83qa0f2j973023p2/webchat.server:1*

*$ docker push cr.yandex/crpf83qa0f2j973023p2/webchat.server:1*

Пушим, смотрим в консоли яндекса, появилось у нас что-нибудь или нет.

Появилось!



Делаем теперь то же самое с фронтенд контейнером.

**И… не работает.**

Потому что у нас не собрался контейнер с клиентским кодом, почему? Потому что у нас в контейнере не оказалось питона

**python Error: not found: python**

Штош, думаю надо его поставить туда, или найти другой пример dockerfile.

Или давай попробуем другой базовый image использовать для сборки фронтенда, давай попробуем поменять

***node:13.12.0-alpine as build***

на

***node:12 as build***

В alpine образы не добавляют ничего лишнего, чтобы размер был поменьше.

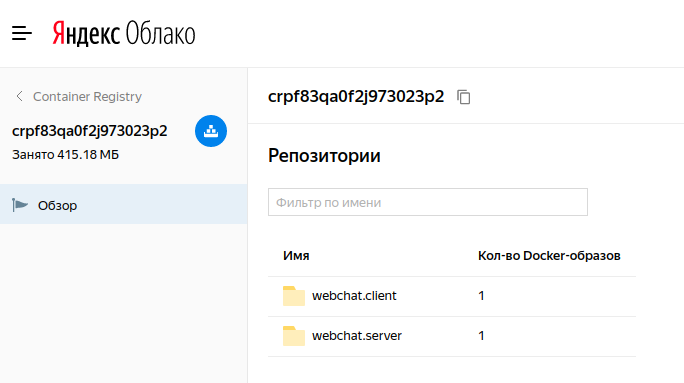
Меняем, пробуем собрать еще раз.

*$ docker build -f Dockerfile.client .*

Пошло дело! Ждем, пока соберется, и закидываем его в registry.

*$ docker tag 0e8a6090b24e cr.yandex/crpf83qa0f2j973023p2/webchat.client:1*

*$ docker push cr.yandex/crpf83qa0f2j973023p2/webchat.client:1*



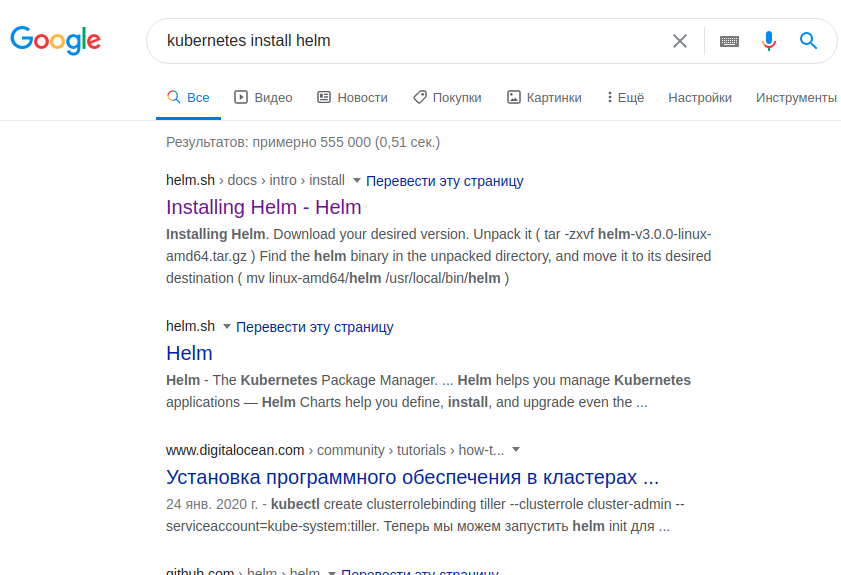
Видим, что в registry появился наш фронтенд. Чудненько, теперь давайте публиковать всё это дело.

Мне нужно быстро написать yaml файлы для нашего приложения, я помню что helm-ом можно сгенерировать шаблон для деплоя, так мы сможем быстро слепить деплой для нашего приложения.

Для этого ставим на нашу виртуалку helm.

**Идем в Google, пишем там:**

**kubernetes install helm**



Первая ссылка.

Скачиваем последнюю версию, распаковываем архив, переносим в папку с бинарными файлами, проверяем.

*$ wget https://get.helm.sh/helm-v3.2.0-linux-amd64.tar.gz*

*$ tar -zxf helm-v3.0.0-linux-amd64.tar.gz*

*$ mv linux-amd64/helm /usr/local/bin/helm*

*$ helm*

Дальше создаем дефолтный чарт, его будем редактировать под наши нужды.

*$ helm create webchat*

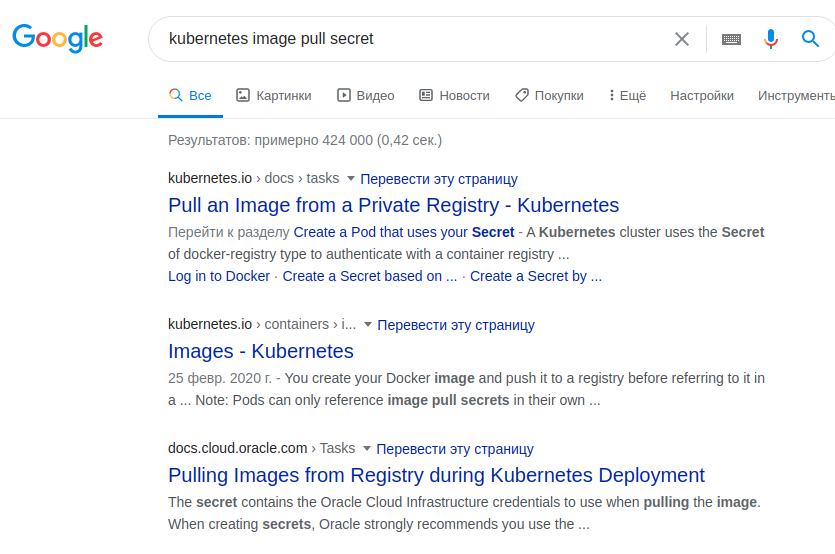
Идем в папку webchat и редактируем файл **values.yaml**.

Так, мне здесь сразу в глаза бросается **imagePullSecrets**, нам нужно его создать, чтобы у нас появилась авторизация в docker registry яндекса.

Как его создать не помню, нужно подсмотреть в гугле.

**Идем в Google, пишем там:**

**kubernetes image pull secret**



Идем по первой ссылке, находим там следующее:

*$ kubectl create secret docker-registry regcred --docker-server=<your-registry-server> --docker-username=<your-name> --docker-password=<your-pword> --docker-email=<your-email>*

Это нам подходит, поменяем поля на те, которые нам нужны для яндекса, и создаем секрет в том неймспейсе, в котором у нас будет приложение опубликовано. Пусть имя неймспейса будет webchat, будем его везде использовать.

*$ kubectl create secret docker-registry regcred --docker-server= cr.yandex/crpf83qa0f2j973023p2 –docker-username=oauth --docker-password=AgAAAAA0x0123123123cdf7ULjkW11SD4-xuk –docker-email=<your-email>-n webchat*

так, не работает, потому что неймспейса такого нет, создадим его и попробуем еще раз.

*$ kubectl create namespace webchat*

Вот, теперь работает.

Теперь нам есть что указать в values.yaml, возвращаемся туда и указываем там:

*imagePullSecrets:*

*- name: regcred*

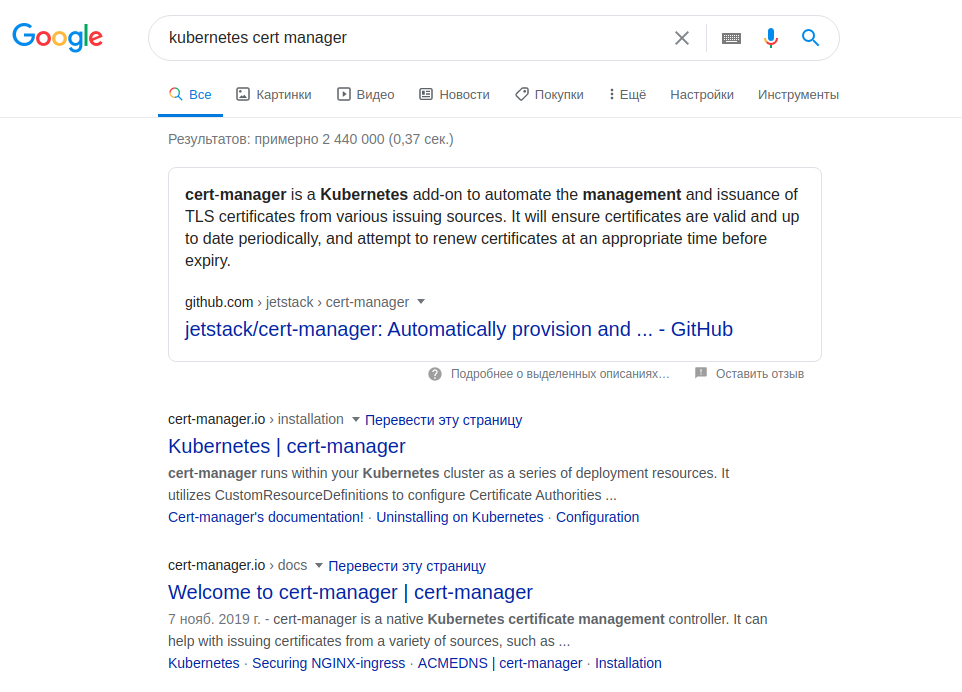
Дальше вижу секцию **ingress**, **annotations**, вижу там **tls**.

Из той статьи Яндекса я помню, что WebRTC работает только по HTTPS. Значит нам нужен будет валидный сертификат. Есть такой замечательный проект, называется LetsEncrypt, они выдают бесплатные сертификаты, причем валидные. Есть проект cert-manager, который автоматизирует процесс выдачи сертификатов kubernetes, его и будем использовать.

Посмотрим, как нам организовать это дело у нас.

**Идем в Google, пишем там:**

**kubernetes cert manager**



Идем в официальную документацию, третья ссылка.

Во, устанавливается одной командой.

*$ kubectl apply --validate=false -f https://github.com/jetstack/cert-manager/releases/download/v0.14.3/cert-manager.yaml*

Ставим, переходим в раздел verify installation.

*$ kubectl get pods --namespace cert-manager*

Дальше создаем тестовый ресурс и проверяем, что он успешно подписан сертификатом.

*cat <<EOF > test-resources.yaml*

*apiVersion: v1*

*kind: Namespace*

*metadata:*

*name: cert-manager-test*

*---*

*apiVersion: cert-manager.io/v1alpha2*

*kind: Issuer*

*metadata:*

*name: test-selfsigned*

*namespace: cert-manager-test*

*spec:*

*selfSigned: {}*

*---*

*apiVersion: cert-manager.io/v1alpha2*

*kind: Certificate*

*metadata:*

*name: selfsigned-cert*

*namespace: cert-manager-test*

*spec:*

*dnsNames:*

*- example.com*

*secretName: selfsigned-cert-tls*

*issuerRef:*

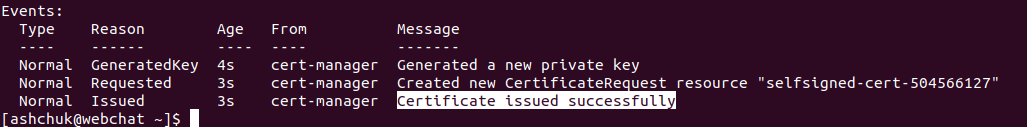
*name: test-selfsigned*

*EOF*

*$ kubectl apply -f test-resources.yaml*

*$ kubectl describe certificate -n cert-manager-test*

Видим запись Certificate issued successful, значит всё работает.



Удаляем тестовые ресурсы и идем устанавливать наше приложение

*$ kubectl delete -f test-resources.yaml*

Дальше нам нужен issuer для нашего приложения.

Переходим в секцию Configuring your first Issuer, Configure, ACME, там находим Creating a Basic ACME Issuer.

https://cert-manager.io/docs/installation/kubernetes/#configuring-your-first-issuer

*apiVersion: cert-manager.io/v1alpha2*

*kind: ClusterIssuer*

*metadata:*

*name: letsencrypt-staging*

*spec:*

*acme:*

*# You must replace this email address with your own.*

*# Let's Encrypt will use this to contact you about expiring*

*# certificates, and issues related to your account.*

*email: user@example.com*

*server: https://acme-staging-v02.api.letsencrypt.org/directory*

*privateKeySecretRef:*

*# Secret resource that will be used to store the account's private key.*

*name: example-issuer-account-key*

*# Add a single challenge solver, HTTP01 using nginx*

*solvers:*

*- http01:*

*ingress:*

*class: nginx*

В поле email указываем наш почтовый адрес.

И поскольку мы идем “в прод”, то нам нужен не промежуточный тестовый сертификат, а доверенный, поэтому меняем адрес ACME сервера на такой:

https://acme-v02.api.letsencrypt.org/directory

Итого получаем

*apiVersion: cert-manager.io/v1alpha2*

*kind: ClusterIssuer*

*metadata:*

*name: letsencrypt-staging*

*spec:*

*acme:*

*# You must replace this email address with your own.*

*# Let's Encrypt will use this to contact you about expiring*

*# certificates, and issues related to your account.*

*email: user@example.com*

*server: https://acme-v02.api.letsencrypt.org/directory*

*privateKeySecretRef:*

*# Secret resource that will be used to store the account's private key.*

*name: webchat-issuer-account-key*

*# Add a single challenge solver, HTTP01 using nginx*

*solvers:*

*- http01:*

*ingress:*

*class: nginx*

Создаем у себя эту штуку.

*$ kubectl create -f ./issuer.yaml*

Так, и теперь нам нужно указать эту штуку в ingress, чтобы он подтянул нужный нам сертификат для нужного доменного имени.

Здесь же, в документации cert manager пишем в поиске слово ingress, переходим на страницу Securing Ingress Resources.

Там находим аннотацию:

*cert-manager.io/cluster-issuer: nameOfClusterIssuer*

и то, что нужно будет в секции tls указать имя секрета, в которое будет помещен сертификат.

*apiVersion: extensions/v1beta1*

*kind: Ingress*

*metadata:*

*annotations:*

*# add an annotation indicating the issuer to use.*

*cert-manager.io/cluster-issuer: nameOfClusterIssuer*

*name: myIngress*

*namespace: myIngress*

*spec:*

*rules:*

*- host: myingress.com*

*http:*

*paths:*

*- backend:*

*serviceName: myservice*

*servicePort: 80*

*path: /*

*tls: # < placing a host in the TLS config will indicate a certificate should be created*

*- hosts:*

*- myingress.com*

*secretName: myingress-cert # < cert-manager will store the created certificate in this secret.*

Возвращаемся к нашему приложению.

Идем в **values.yaml,** в секции ingress пишем, во первых, что он у нас enabled: true, потом нужные аннотации:

*kubernetes.io/ingress.class: nginx*

*cert-manager.io/cluster-issuer: letsencrypt-staging*

В секции hosts пишем доменное имя, по которому будет доступно наше приложение.

Пусть будет **webchat.compositekey.tech.**

Дальше в секции tls как и длоговаривались, пишем имя секрета, в котором будет храниться сертификат, и имя доменное имя.

Получается как-то так:

*ingress:*

*enabled: true*

*annotations:*

*kubernetes.io/ingress.class: nginx*

*cert-manager.io/cluster-issuer: letsencrypt-staging*

*# kubernetes.io/tls-acme: "true"*

*hosts:*

*- host: webchat.compositekey.tech*

*paths: []*

*tls:*

*- secretName: webchat-tls*

*hosts:*

*- webchat.compositekey.tech*

Дальше нам нужно прописать здесь образы нашего приложения.

И поскольку у нас их два, сервер и фронтенд, то нам нужно два деплоя.

И два сервиса.

Давай сделаем.

Во первых, здесь же, в файле **values.yam**l в секции **images** укажем два наших образа, тег им укажем одинаковый – единичка. И добавим еще два параметра, мы ими будем помечать сущности, чтобы имена не совпадали.

Подсмотрим имена образов в консоли, а то я уже забыл, что там было.

*$ docker images*

*выведет что-то такое:*

*cr.yandex/crpf83qa0f2j973023p2/webchat.client*

*cr.yandex/crpf83qa0f2j973023p2/webchat.server*

добавим еще два поля:

*labels:*

*backend: backend*

*frontend: frontend*

и секция images будет выглядеть так:

*image:*

*backend: cr.yandex/crpf83qa0f2j973023p2/webchat.server*

*frontend: cr.yandex/crpf83qa0f2j973023p2/webchat.client*

*pullPolicy: IfNotPresent*

*# Overrides the image tag whose default is the chart version.*

*tag: "1"*

Дальше мы идем в папку templates, в ней лежит описание деплойментов, сервисов и всё остальное содержимое чарта.

Идем редактировать deployment.yaml

Для начала дублируем всё содержимое файла и разделяем его на две части тремя тире.

Первая часть будет для сервера, вторая – для фронтенда.

Потом добавляем к селекторам и именам наш label, который мы указали в values.yaml

*{{ .Values.labels.backend }} и {{ .Values.labels.frontend }} соответственно*

Закомментируем секции **livenessProbe** и **readinessProbe**, поскольку их мы не реализуем, как нибудь в следующий раз.

И в ports в поле **containerPort** указываем те порты, которые мы указывали, когда собирали контейнеры.

То же самое делаем с **service.yaml**

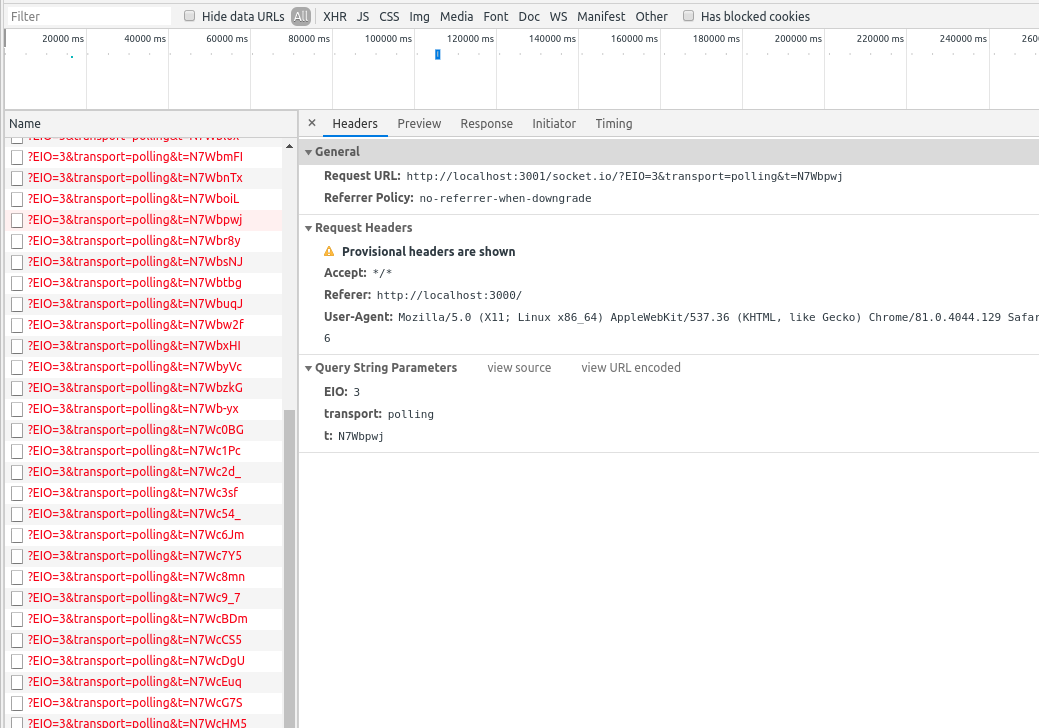
Дальше уже поинтереснее, идем в **ingress.yaml.**

В этом файле нам надо указать, по каким url какие сервисы будут принимать трафик.

Как нам узнать, куда заворачивать трафик?

Для WebRTC нам по идее не нужно придумывать отдельный машрут, а вот для вебсокета надо.

Когда мы запускали на localhost приложение без бекенда, то там в консоли были ошибки типа такой.



Вот тут и видно, что запросы в бекенл идут по пути /socket.io

Значит в ingress так и будем направлять трафик, /socket.io – в бекенд, всё остальное – в фронтенд.

Чтобы сильно не задерживаться на этом шаге, соберем paths так:

*paths:*

*- path: /*

*backend:*

*serviceName: {{ $fullname }}-{{ .Values.labels.frontend }}*

*servicePort: {{ $svcPort }}*

*- path: /socket.io*

*backend:*

*serviceName: {{ include "webchat.fullname" . }}-{{ .Values.labels.backend }}*

*servicePort: {{ $svcPort }}*

И в самом верху добавим две новые переменные

*{{- $backendLabel := .Values.labels.backend -}}*

*{{- $frontendLabel := .Values.labels.frontend -}}*

Теперь проверяем, не накосячили ли мы где-нибудь.

*$ helm template ./*

Если ошибок нет, то пробуем всё это запустить.

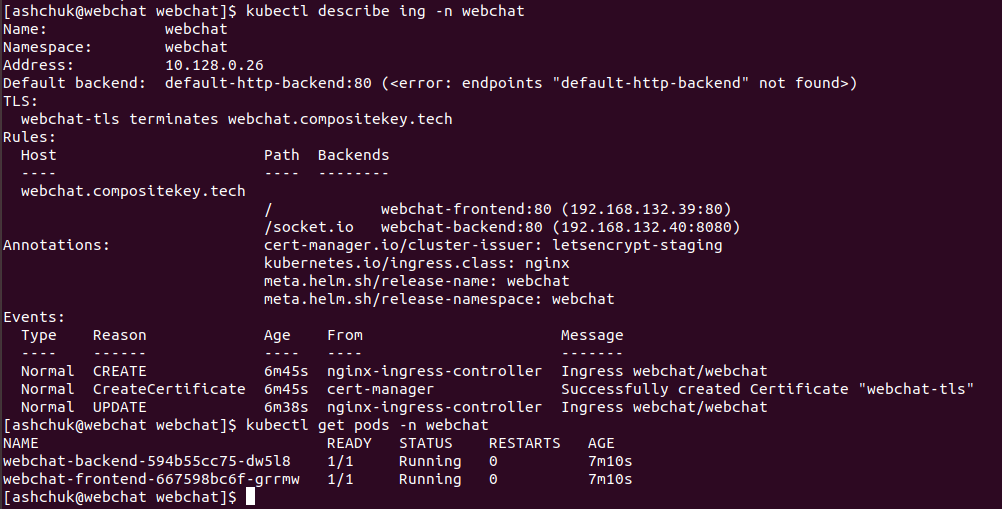
Идем в консоль и делаем следующее:

*$ helm install ./ --name webchat –namespace webchat*

И смотрим, насколько удачно прошла публикация

*$ kubectl describe ing -n webchat*

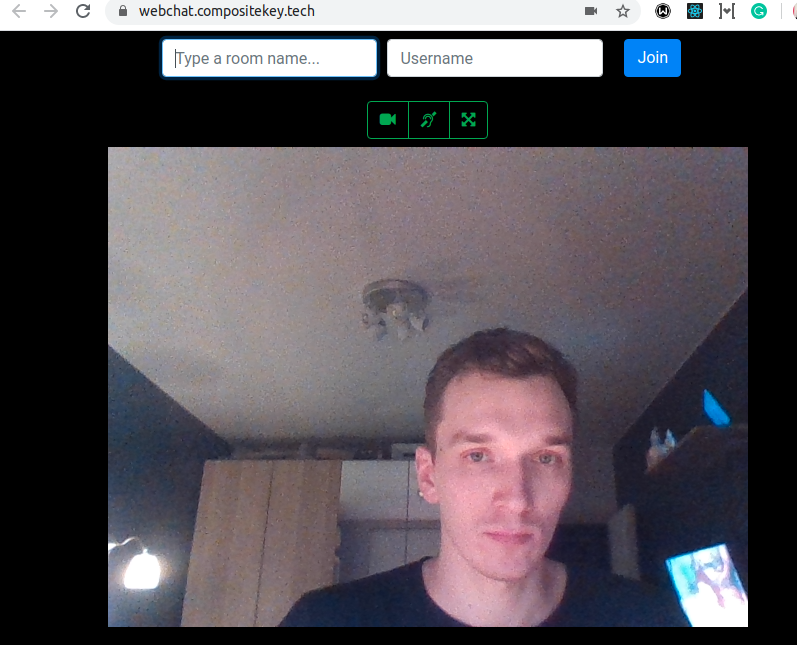
*$ kubectl get pods -n webchat*



Ну, и в конечном счете, идем по адресу

https://webchat.compositekey.tech

Должно получиться что-то такое



**И… не работает.**

В консоли ошибок нет, но что-то не работает, пользователи друг друга не видят.

Давайте еще раз заглянем в код.

Во первых, мы так и не поменяли в index.js тот код, который ходил в localhost в зависимости от переменной среды.

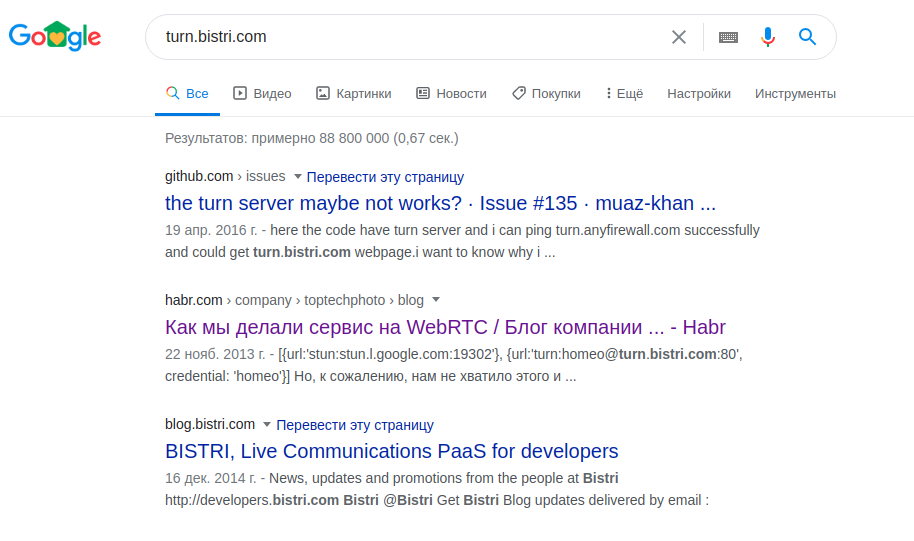
И второй момент, который мне не нравится – это количество ice server-ов здесь же, в index.js.

Возможно они не функционируют и поэтому у нас ничего не работает.

Давайте расширим список, пойдем с вопросом в гугл

**Идем в Google, пишем там:**

**turn.bistri.com**



Вторая ссылка на habr, статья, в которой есть список stun turn серверов, давай добавим их к себе и попробуем запустить приложение еще раз.

*[{url:'stun:stun01.sipphone.com'},*

*{url:'stun:stun.ekiga.net'},*

*{url:'stun:stun.fwdnet.net'},*

*{url:'stun:stun.ideasip.com'},*

*{url:'stun:stun.iptel.org'},*

*{url:'stun:stun.rixtelecom.se'},*

*{url:'stun:stun.schlund.de'},*

*{url:'stun:stun.l.google.com:19302'},*

*{url:'stun:stun1.l.google.com:19302'},*

*{url:'stun:stun2.l.google.com:19302'},*

*{url:'stun:stun3.l.google.com:19302'},*

*{url:'stun:stun4.l.google.com:19302'},*

*{url:'stun:stunserver.org'},*

*{url:'stun:stun.softjoys.com'},*

*{url:'stun:stun.voiparound.com'},*

*{url:'stun:stun.voipbuster.com'},*

*{url:'stun:stun.voipstunt.com'},*

*{url:'stun:stun.voxgratia.org'},*

*{url:'stun:stun.xten.com'},*

*{*

*url: 'turn:numb.viagenie.ca',*

*credential: 'muazkh',*

*username: 'webrtc@live.com'*

*},*

*{*

*url: 'turn:192.158.29.39:3478?transport=udp',*

*credential: 'JZEOEt2V3Qb0y27GRntt2u2PAYA=',*

*username: '28224511:1379330808'*

*},*

*{*

*url: 'turn:192.158.29.39:3478?transport=tcp',*

*credential: 'JZEOEt2V3Qb0y27GRntt2u2PAYA=',*

*username: '28224511:1379330808'*

*}]*

*И заодно прописываем адрес https://webchat.compositekey.tech в переменную signaling компонента PeerDataProvider здесь же, в index.js.*

Собираем новую версию, публикуем

*$ docker build -t cr.yandex/crpf83qa0f2j973023p2/webchat.client:2 -f Dockerfile.client .*

*$ docker push cr.yandex/crpf83qa0f2j973023p2/webchat.client:2*

*$ kubectl edit -n webchat deploy webchat-frontend*

*меняем - image: cr.yandex/crpf83qa0f2j973023p2/webchat.client:1 на - image: cr.yandex/crpf83qa0f2j973023p2/webchat.client:2, сохраняем, выходим.*

Ждем, пока обновится версия, и пробуем снова.

Работает! И даже с телефона работает!

