

Picard Cloud Environment Software Validation and Verification

Software Requirements	2
Verification of Software Requirements	3
Hosting Configuration, Security and Encryption	4
Audit Trail Logs	4
EC2 Configurations	4
Flask Application and API Testing	4
Picard Web Console	5
Software Architecture and Detailed Design	5
Hosting Configuration, Security and Encryption	5
Architectural Diagram	5
AWS Account Security Features	7
AWS Console Security	7
AWS Token Security	7
AWS Server SSH-KEY Security	7
AWS Account Usage Alarms	7
Audit Trail Logs	8
API Overview Schematic	8
Audit Trail: API Request Logs	9
EC2 Configurations	11
NGINX	11
Fail2ban	14
POSTFIX	14
WSGI	15
Service	15
Flask and Celery	16
Cron	16
Flask App	17
API Basics	17
API Request Headers	18
App Configs	18
Custom API Error Codes	20
User Accounts	21
Identity (Email) Verification	22
Session Management	22

Cookies	22
Api Keys	23
Session Protection	23
Data	23
Doctypes	24
Mappings	24
Permissions	24
Usergroups and System Roles	25
Groupings	26
Creating Documents	27
Obtaining Documents	29
Updating Documents	30
Obtaining Previous Versions of Documents	32
Attaching Files to Documents	35
Deleting data	36
Electronic Signatures	37

Software Requirements

The Picard Cloud Environment and API meets the following two high-level requirements for managing clinical trial data.

1. Capture electronic signatures, records, and the audit trail to satisfy the CFR 21 Part 11 Requirement.
2. Ensure compliance with the security and privacy requirements of the HIPAA guidelines.

For our purposes, the high-level requirements are broken down into the following.

1. System validation to ensure accuracy, reliability, consistent intended performance, and the ability to discern invalid or altered records.
2. Generate human readable records.
3. Ensure the protection of records when being transmitted between applications and the cloud via the API.
4. Ensure the protection of records when stored in the cloud.
5. Limit cloud system and API access to authorized individuals.
6. Create computer-generated, time-stamped audit trails that show who changed what and when.
7. Operational system checks to ensure that only the permitted sequencing of steps and events is enforced for managing electronic records and signatures.
8. Authority checks to ensure that only authorized users can use the system to electronically generate and sign documents and access trial data through the API.

9. Authority checks to ensure that only authorized users can access the cloud environment.
10. Peripherals check to ensure that the inputs and outputs are only being accessed via the API and cloud environment by approved applications.
11. Training of the people who work with the system or develop it.
12. Prevention of electronic record and signature falsification so that people are liable in writing for what they sign.
13. System documentation e.g. on who has access to the system, how this access is granted, whether it be for the use or maintenance of the system, and on who changed what in the system and when.
14. Ability to capture digital signatures with the following data:
 - a. The name of the signatory
 - b. The date and time of the signature and
 - c. The meaning of the signature (e.g. review, approval, author).
15. It must not be possible to falsify the digital signature or corresponding electronic records.
16. The signature must be linked to the document in such a way that it cannot be used on other documents.
17. It must be possible to assign the signature to a specific individual.
18. Electronic signature must be regulated in such a way that any attempted misuse of someone else's electronic signature requires the collaboration of two or more individuals.
19. The duplicate assignment of codes and passwords must not be possible.
20. Both codes and passwords must be regularly checked to ensure that they are still sufficiently secure.
21. In the event that codes, passwords, cards, etc. are lost, there must be a procedure that permits "deauthorization".
22. Suitable measures must be in place to protect against and detect unauthorized access attempts.
23. Ensure data is securely backed up
24. Ensure data is encrypted when transferring into and out of the VPC, and while at rest in the VPC.

Verification of Software Requirements

We break the software verification process into 4 distinct protocols. Protocols are presented to testers in the form of a google sheet and are carried prior to a trial being launched. Testers will edit the google sheet with the testing results.

Changes to anything covered by the testing plan during a trial will create updated versions of the protocols. Depending on the changes, testers will opt to either redo all or parts of the protocols. The rationale for not testing all protocols will be added as a note to the updated testing plan.

The testing must first be carried out against the development portals. Once the tests pass there the code changes will be pushed to the production server.

Hosting Configuration, Security and Encryption

The purpose of this protocol is to ensure the Hosting Environment is configured per the product requirements. The tests are executed via a combination of automation and manual inspection of the AWS environments used to create the AMI. The key requirements being tested here relate to the following:

- Security and Encryption
- Backups and Disaster Recover
- Fault Tolerance
- Application Support

Audit Trail Logs

The purpose of this protocol is to ensure the API requests are logged and handled per the product requirements. The tests are executed via a combination of automation and manual inspection of the AWS environments used to manage API requests. The key requirements being tested here relate to the following:

- Audit Trails
- API request headers
- API Request Flow
- API Request Logging

EC2 Configurations

The purpose of this protocol is to ensure the EC2 servers are configured per the product requirements. The tests are executed via inspection of the EC2 instance used to create the AMI. The key requirements being tested here relate to the following:

- AMI
- AUTO SCALING
- NGINX
- POSTFIX
- FLASK APP
- CELERY
- CRON
- FAIL2BAN

Flask Application and API Testing

The purpose of this protocol is to ensure the Flask App meets the product requirements. The tests are executed via a python script. Note we run more than 1,000 distinct tests, but only

include relevant tests in the protocol. The key requirements being tested here relate to the following:

- Signing Electronic Records with Audit Trail
- User Accounts, Passwords and Sessions
- Roles, endpoints and data permissions

Picard Web Console

The purpose of this protocol is to ensure the Picard Web Console meets the product requirements. The tests are executed via a user manually. The key requirements being tested here relate to the following:

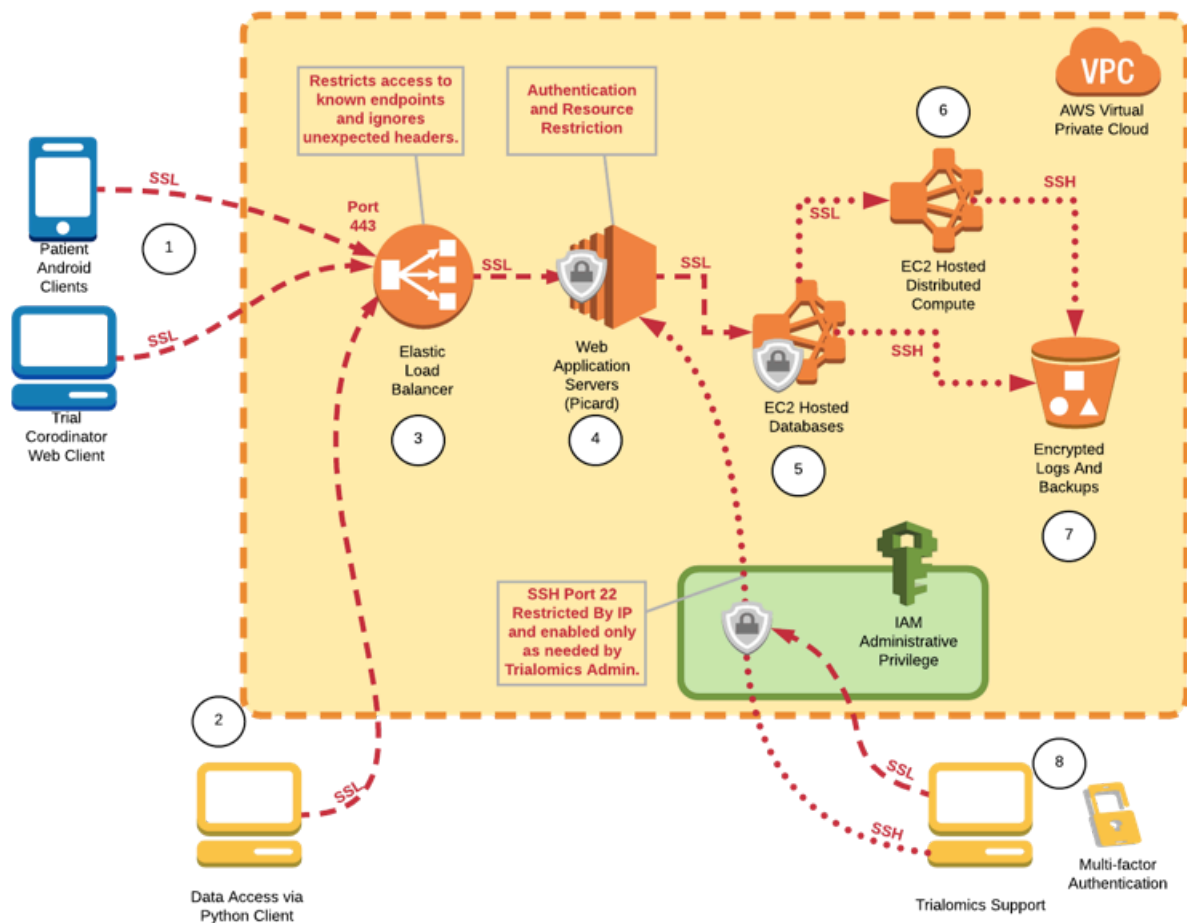
- Can you log in and out of the web console and does the application manage sessions to ensure they expire due to inactivity.
- Can you manage users and data permissions, roles, user groups, and other features related to securing data.
- Can you explore system logs to identify issues.

Software Architecture and Detailed Design

Hosting Configuration, Security and Encryption

Architectural Diagram

The Trialomics technical platform has been developed to facilitate compliance with the potential requirements of the Privacy and Security Rules, specifically regarding the encryption of PHI in transmission (“in-flight”) and in storage (“at-rest”). Furthermore, all servers storing PHI are hosted on encrypted instances of the AWS platform. The figure below depicts the key transfer points of data within Trialomics’ platform and describes the purpose of each asset.



1. **User Clients** – Utilizing industry standard bank grade Secure Sockets Layer (SSL) encryption, Trialomics helps protect internet traffic (data "in-flight") from snooping (breach of privacy) as it travels between the apps and our secured virtual private cloud instance in AWS.
2. **Data Access** – If necessary the servers and data can be accessed via a python SDK. Only members of the Workforce role have privileges to access data in this way.
3. **Elastic Load Balancer (ELB)** – A high performance load-balancer provides scalable performance and redundancy, routing traffic to the application web servers.
4. **Web Application Services** – Multiple web application servers are utilized to respond and process requests, acting as a gatekeeper which authenticates and restricts access to the databases.
5. **EC2 Hosted Databases with Encrypted Disk** – A collection of data warehousing servers provide data querying, utilizing 256-bit Advanced Encryption Standards (AES) encryption to protect the data at a disk level (data "at-rest").
6. **EC2 Hosted Distributed Compute** – A collection of data warehousing servers provide compute capability, utilizing 256-bit Advanced Encryption Standards (AES) encryption to protect the data at a disk level (data "at-rest").

7. **Encrypted Logs and Backups** – Following best practices, all of Trialomics' development and administrative servers use the same encryption techniques using a Secure Shell (SSH). Secured and encrypted backups of all logs and data associated with this process provide a data trail strong enough to pass most audits, and recover from almost any disaster.
8. **Admin Access** – If necessary the servers and data can be accessed via the Amazon Web Services console or through ssh via port 22. By default this port is closed, but can be made available to a request from specific IP addresses if necessary. Accessing the AWS console requires Multi-Factor authentication, restricting access to users who can pass this rigorous requirement.

AWS Account Security Features

AWS Console Security

Accessing our AWS account via the web console requires both an MFA token and an AWS password that is updated quarterly. Accessing the MFA tokens requires the attacker to gain access to a phone that is password protected. There are only two accounts which have access to the AWS web console.

AWS Token Security

Accessing our AWS account via the security tokens requires access to the tokens, which is only possible if they access the Web Console to generate new tokens or access a server on which the tokens are stored.

AWS Server SSH-KEY Security

Access to the API servers requires an ssh key that is stored in a password protected database with the password stored in a separate location. As yet another security feature we turn off access to these servers via all ports, meaning that even with the ssh key an attacker would not be able to gain access unless they accessed our AWS account.

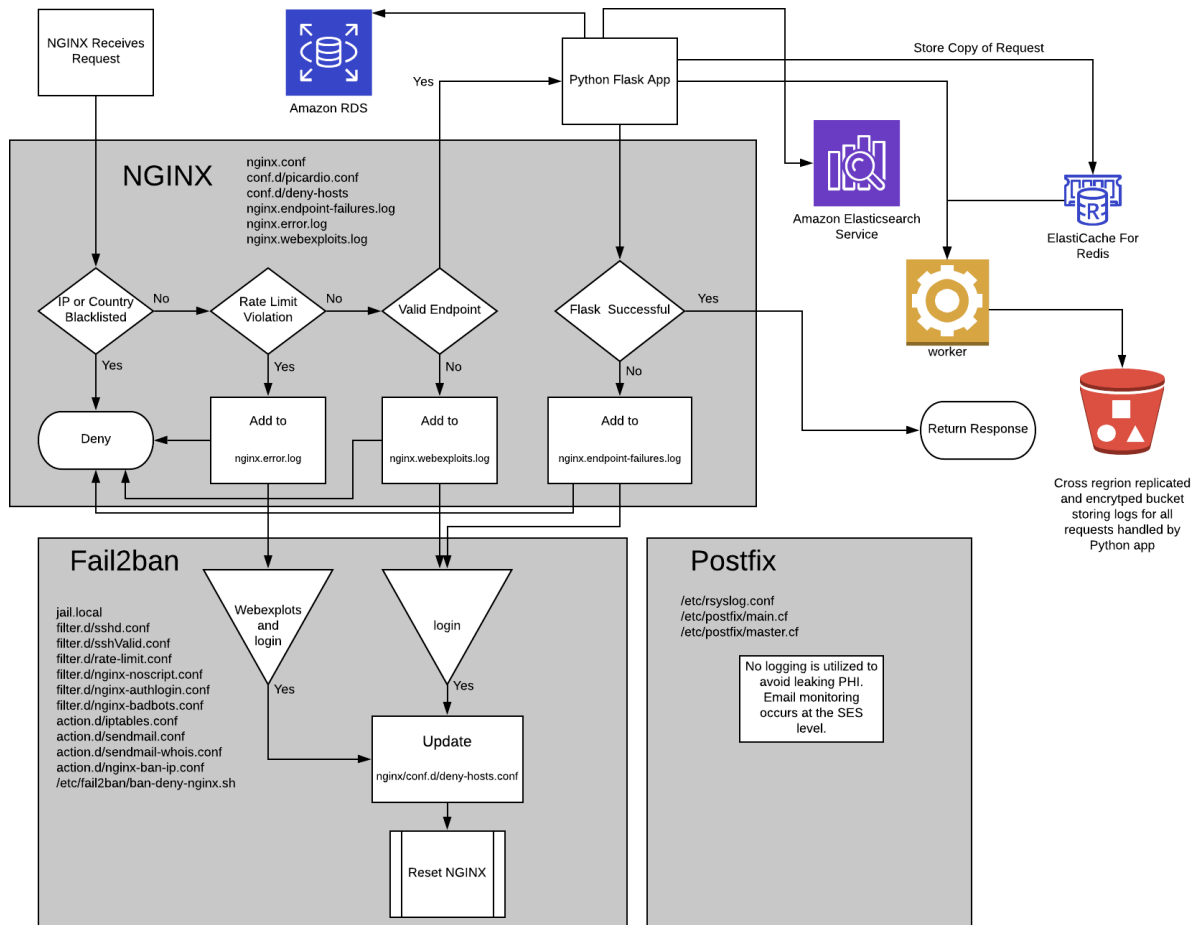
AWS Account Usage Alarms

Any time someone logs into our AWS account an SMS message is sent to two administrators who can determine the threat level based on a simple conversation (ie did you just login? No, then..). If a server is accessed via ssh, an email is sent to our system administrator. This ensures that if someone does somehow gain access, the System Administrator will have immediate knowledge and can reach out to AWS to shut down the account in a prompt manner.

Audit Trail Logs

API Overview Schematic

The following schematic illustrates how Picard handles API requests in the cloud. The various components outlined here are designed to ensure security and auditability of users interacting with client applications that connect to the Picard API.



At a high level this diagram states the following:

- If an API request fails due to an IP address being blocked, a rate limit violation, improperly formatted request, or request failure then the request fails and returns an HTTP response with a status greater than 400 and a message potentially indicating what went wrong.
- If the call is successful and creates or updates data, then the data is stored in the following locations:
 - Encrypted, redundant Postgres server that is backed up nightly. Backups are encrypted to ensure HIPAA compliance.

- Encrypted, redundant Elasticsearch servers that are backed up nightly. Backups are encrypted to ensure HIPAA compliance.
 - Encrypted, Cross-Region Replicated Amazon S3 object with versioning turned on to enable data auditability.
- A log of the call is stored in the following locations (see next section for log details):
 - Encrypted, redundant Postgres server that is backed up nightly. Backups are encrypted to ensure HIPAA compliance.
 - Encrypted, redundant Elasticsearch servers that are backed up nightly. Backups are encrypted to ensure HIPAA compliance.
 - Encrypted Amazon S3 object with versioning turned on to enable data auditability.

Audit Trail: API Request Logs

On each API call Picard stores a record of the api call as documents in Postgres, ElasticSearch, and Amazon S3. Log documents cannot be deleted, ensuring complete auditability of which user modified which data when. Each log document contains a plethora of information about the calling user, their location, device, endpoint, parameters, etc.

The images below contain all the information gathered from a specific API call. This includes the endpoint, params masked to exclude account credentials and other sensitive data, where the call was made from, what type of browser or application, the response status, and exception information in case of an error.

These logs can be accessed through the API, but only by a trial administrator.

Timestamp	"2017-12-14T09:22:40.230Z"
Thread	140476320065600
Thread Name	uWSGIWorker1Core0
Logger Name	MainProcess
Method	log_call
Host Name	

ID	
Message	home-flask-1511302253-10-1-5-106 GET /admin/publicConfigList returned 200 OK to user admin from 109.147.148.5 (0.051s)
Level	INFO
Module	request_handler
File Name	request_handler.py
Line Number	378
Hostip	

User

username	admin
authtype	session
Provider	picard
uuid	6239b416-a6d1-52dd-8ae4-079074a4df72
fresh	true

Request

User Platform	android
User Browser	chrome
Browser Version	61.0.3163.100
Path	/admin/publicConfigList

Is PC	false
Is Bot	false
Is Touch Capable	true
Is Tablet	false
Is Mobile	true

Browser

Version String	61.0.3163
Family	Chrome Mobile

City	Tempe
Country	United States
Code	753
Latitude	33.4148

Host	home.picard.io
Remote Address	109.147.148.5
Scheme	https
Method	GET

Device

Brand	LG
Model	Nexus 5
Family	Nexus 5

State	Arizona
Continent	North America
Postal	85287
Longitude	-111.9093

params	<div></div>
--------	-------------

Response

Status	200 OK	Execution Time	0.051
Status Code	200	Content Length	1294

Exception

Type		Value	
params			

EC2 Configurations

NGINX

We configure nginx to run as a linux service. Following the recommended nginx configuration pattern, we use the nginx.conf file and a separate conf file for the Picard API. We use these files to do the following:

- Specify log file formats
- Ensure any request uses one of the following headers: (GET|HEAD|POST|DELETE|PUT|OPTIONS)
- Don't log requests from the load balancer
- Log failed and successful requests to their own files
- Log requests attempting to access invalid endpoints and add them to the deny hosts file used by fail2ban.
- Pass any valid request to a socket.

Here are example files running on a production server.

```
[root@newcust3-flask-1582585783-10-1-5-199 ec2-user]# cat /etc/nginx/nginx.conf
# For more information on configuration, see:
# * Official English Documentation: http://nginx.org/en/docs/
# * Official Russian Documentation: http://nginx.org/ru/docs/

user nginx;
worker_processes auto;
error_log /var/log/nginx/error.log;
pid /var/run/nginx.pid;

# Load dynamic modules. See /usr/share/nginx/README.dynamic.
include /usr/share/nginx/modules/*.conf;

events {
```

```

worker_connections 1024;
}

http {

    real_ip_header    X-Forwarded-For;
    set_real_ip_from  0.0.0.0/0;

    geoip_country /usr/share/GeoIP/GeoIP.dat;
    map $geoip_country_code $allowed_country {
        default yes;
        CN no;
    }

    #log_format webexploits
    '$http_x_forwarded_for$request_method$uri$time_local$body_bytes_sent$http_referer$http_user_agent$request_time$upstream_connect_time$upstream_header_time$upstream_response_time$geoip_country_code';
    log_format webexploits
    '$http_x_forwarded_for$request_method$uri$time_local$http_user_agent$geoip_country_code';
    log_format web401s '$request_method$uri$time_local$http_user_agent$geoip_country_code';
    map $http_user_agent $elb {
        ~*HealthChecker 1;
        default 0;
    }
    map $status $abnormal {
        ~^2 0;
        default 1;
    }
    map $status $normal {
        ~^2 1;
        ~^3 1;
        default 0;
    }

    # Don't store load balancer requests that are 200
    #access_log /home/picard/logs/nginx.access.log webexploits if=$normal;
    access_log /home/picard/logs/nginx.endpoint-failures.log web401s if=$abnormal;

    error_log /home/picard/logs/nginx.error.log;

    sendfile        on;
    tcp_nopush       on;
    tcp_nodelay       on;
    keepalive_timeout 65;
    types_hash_max_size 2048;

    include          /etc/nginx/mime.types;
    default_type      application/octet-stream;

    # Load modular configuration files from the /etc/nginx/conf.d directory.
    # See http://nginx.org/en/docs/nginx_core_module.html#include
    # for more information.
    include /etc/nginx/conf.d/*.conf;
    include /etc/nginx/conf.d/deny-hosts;
}

```

```
[root@newcust3-flask-1582585783-10-1-5-199 ec2-user]# cat /etc/nginx/conf.d/picard.conf
```

```
limit_req_zone $http_x_forwarded_for zone=req_zone:10m rate=60r/m;
limit_req_zone $http_x_forwarded_for zone=req_zone_auth:10m rate=60r/m;
limit_req_zone $http_x_forwarded_for zone=one:1m rate=1r/m;
```

```
server {
    listen      80;
    server_name localhost;

    if ($allowed_country = no) {
        return 444;
    }

    set $logging 1;
    set $logtest "";

    if ($elb = 1) {
        set $logtest "${logtest}1";
    }

    if ($normal = 1){
        set $logtest "${logtest}1";
    }

    if ($logtest = "11"){
        set $normal 0;
    }

    if ($request_method !~ ^(GET|HEAD|POST|DELETE|PUT|OPTIONS)$ ) {
        return 444;
    }

    location ~
^(/custom/dukeSms/custom/sendSms/auth/verifyIdentity/custom/electronicSignature/auth/login/geoiptool/targetCloseb
yname/geoiptool/targetClosebyname/geoiptool/targetGuess/geoiptool/randomList/geoiptool/targetList/geoiptool/dbRelo
ad/hubspot/query/custom/documentBulkDelete/custom/documentBulkCreate/custom/documentSearch/custom/doctype
Reindex/custom/doctypeCheck/custom/bulkloadList/custom/doctypeClone/custom/doctypeList/custom/fieldUpdate/cus
tom/endpoint/custom/document/custom/document/custom/document/custom/document/custom/grouping/custom/grou
ping/custom/mapping/custom/mapping/custom/doctype/custom/doctype/custom/doctype/custom/doctype/custom/load
/custom/load/custom/load/custom/load/custom/git/admin/getClusterHealth/admin/controlPanelList/admin/publicConfig
List/admin/templateCatList/admin/templateSend/admin/templateTest/admin/templateList/files/deleteFiles/terms/tosfor
mList/admin/dblogQuery/admin/apitestRun/admin/configList/admin/dblogEvent/admin/dblogEvent/admin/dblogList/ad
min/cacheTest/admin/template/admin/template/admin/template/admin/template/admin/stackId/admin/sitemap/terms/to
sform/terms/tosform/terms/tosform/terms/usertos/admin/config/admin/config/files/file/files/file/files/file/auth/oauthprov
iderListunconfigured/auth/oauthproviderUserlist/auth/passwordUpdatebyToken/auth/userPasswordChangeSet/auth/em
ailValidationReset/auth/emailValidationList/auth/oauthproviderList/auth/regAuthChecksTest/auth/userIdentityTest/auth/u
serActiveUpdate/auth/emailValidation/auth/emailValidation/auth/emailValidation/auth/usergroupMember/auth/usergrou
pMember/auth/usergroupMember/auth/passwordUpdate/auth/passwordReset/auth/passwordReset/auth/oauthprovider
/auth/oauthprovider/auth/oauthprovider/auth/oauthprovider/auth/usergroupList/auth/apikeyTest/auth/apikeyList/auth/u
sergroup/auth/usergroup/auth/usergroup/auth/usergroup/auth/oauthUser/auth/oauthUser/auth/emailpref/auth/emailpr
ef/auth/roleUser/auth/roleUser/auth/roleList/auth/userList/auth/pgpkey/auth/pgpkey/auth/pgpkey/auth/apikey/auth/ap
ikey/auth/apikey/auth/logout/auth/user/auth/user/auth/user/auth/user/auth/user/auth/role/auth/role/auth/role/auth/role/geoiptoo
l/static/hubspot/hubspot/custom/static/admin/static/terms/static/files/files/auth/oauthAuthorize/auth/oauthRedirect/aut
h/oauthCallback/auth/oauthResource/auth/oauthRevoke/auth/static/favicon.ico/static/healthCheck) {
        client_max_body_size 0;
    }
}
```

```

        include uwsgi_params;
        uwsgi_pass unix:/tmp/uwsgi.minnetronix.sock;
    }

    location = /admin/stackId {
        limit_req zone=req_zone_auth burst=1;
        client_max_body_size 0;
        include uwsgi_params;
        uwsgi_pass unix:/tmp/uwsgi.minnetronix.sock;
    }

    location = / {
        client_max_body_size 0;
        include uwsgi_params;
        uwsgi_pass unix:/tmp/uwsgi.minnetronix.sock;
    }

    location ~* / {
        limit_req zone=one;
        access_log /home/picard/logs/nginx.webexploits.log webexploits;
        deny all;
    }
}

```

Fail2ban

We configure fail2ban to run as a linux service. Following the recommended fail2ban configuration pattern, we update the jail.local file to specify the following:

- Ban anyone who fails logging in via ssh for 10 seconds
- Ban anyone who fails logging in
- Send an email on successful ssh login
- Ban anyone who accesses an invalid endpoint for 60 seconds
- Ban anyone who accesses an endpoint known to be commonly exploited by hackers for 86,400 seconds

POSTFIX

We configure postfix as the mail server and configure it to send emails to Amazon S3. We have an IAM user who has permissions to SES. The credentials are stored in the sasl_password file in the postfix directory. If you need to update the credentials or change IAM users you simply modify the ACCESS_KEY and ACCESS_SECRET values accordingly, then follow the standard Postfix documentation to reset the service.

```

[root@newcust3-flask-1582585783-10-1-5-199 fail2ban]# cat /etc/postfix/sasl_passwd
[email-smtp.us-west-2.amazonaws.com]:25 ACCESS_KEY:ACCESS_SECRET

```

WSGI

We use a WSGI container to connect messages passed from nginx to the socket to the Flask application. Here are the files from a production server:

```
[uwsgi]
project = minnetronix
projectdir = /home/picard/

socket = /tmp/uwsgi.%(project).sock
chmod-socket = 666
chown-socket = picard:picard
uid = picard
gid = picard

processes = 1
gevent = 100
idle = 3600
buffer-size = 32768
disable-logging = true
stats = :1717
listen=500

module = wsgi
chdir = %(projectdir)/app/
virtualenv = /home/picard/venv/
pythonpath = %(virtualenv)

logto = %(projectdir)/logs/uwsgi.log
pidfile = %(projectdir)/.%(project).pid
[root@newcust3-flask-1582585783-10-1-5-199 picard]# cat wsgi.py
import sys
from subprocess import call
import logging
import os

activate_this = '/home/picard/venv/bin/activate_this.py'
execfile(activate_this, dict(__file__=activate_this))

sys.path.insert(0, '/home/picard/app/')
from picardio import app as application
```

Service

We use the linux service manager to manage the Flask App and Celery Worker. The configuration files are shown below. Note this allows us to run 'service supervisor restart/start/stop' to start and stop the Flask App and Celery Worker.

```
[root@minnetronix-flask-1518596015-10-1-5-134 picard]# cat /etc/supervisor/conf.d/picard.conf
[program:picard]
```

```

environment = HOME=/home/picard
command = /usr/bin/uwsgi --ini /home/picard/wsgi.ini
directory=/home/picard/app/
user=picard
stopasgroup=true
redirect_stderr=true
stdout_logfile=/home/picard/logs/supervisord.stdout.log
autostart=true
autorestart=false
[root@minnetronix-flask-1518596015-10-1-5-134 picard]# cat /etc/supervisor/conf.d/celeryd.conf
; =====
; celery worker supervisor
; =====

[program:celery]
command=/home/picard/venv/bin/celery -A worker worker --loglevel=INFO
directory=/home/picard/app

user=picard
numprocs=1
stdout_logfile=/home/picard/logs/celery-stdout-worker.log
stderr_logfile=/home/picard/logs/celery-stderr-worker.log
autostart=true
autorestart=true
startsecs=10

; Need to wait for currently executing tasks to finish at shutdown.
; Increase this if you have very long running tasks.
stopwaitsecs = 600

; Causes supervisor to send the termination signal (SIGTERM) to the whole process group.
stopasgroup=true

; Set Celery priority higher than default (999)
; so, if rabbitmq is supervised, it will start first.
priority=1000

```

Flask and Celery

The Flask app runs the API and has an entire section dedicated to its architecture and how the API works. For purposes of the EC2 configuration, the Flask App runs inside a virtual environment and is managed as a linux service. The virtual environment is accessible in the picard user home directory.

```

[root@minnetronix-flask-1518596015-10-1-5-134 picard]# cat /etc/supervisor/conf.d/picard.conf
root@minnetronix-flask-1518596015-10-1-5-134 picard]# ls /home/picard/venv/
bin include lib lib64 LICENSE local man pip-selfcheck.json shapely

```

Cron

We use the cron scheduler to restart the fail2ban and supervisor services.

```
[root@newcust3-flask-1582585783-10-1-5-199 picard]# cat /etc/crontab
SHELL=/bin/bash
PATH=/sbin:/bin:/usr/sbin:/usr/bin
MAILTO=root
HOME=/

# For details see man 4 crontabs

# Example of job definition:
# .----- minute (0 - 59)
# | .----- hour (0 - 23)
# | | .----- day of month (1 - 31)
# | | | .----- month (1 - 12) OR jan,feb,mar,apr ...
# | | | | .---- day of week (0 - 6) (Sunday=0 or 7) OR sun,mon,tue,wed,thu,fri,sat
# | | | | |
# * * * * * user-name command to be executed
0 0 * * * root service fail2ban restart
42 0 * * * root service supervisor restart
```

Flask App

The Flask App is the Picard API and is broken down into four main modules (see below). Developers should consult the unit tests to truly understand how the API works. The remainder of this section is designed to illustrate the main endpoints used to manage users, passwords, logs, and data.

Flask App			
<u>Auth</u> User accounts Passwords Sessions API-KEYS Roles and Users	<u>Admin</u> App Configs Logs	<u>Custom</u> Doctypes Documents Mappings Groupings E-Signatures E-Records	<u>Files</u> File management

API Basics

The Picard API allows an app to programmatically interact with a Picard stack via simple HTTP requests. Our API design follows pure REST principles when possible. As is the standard for REST apis, each endpoint is designed to associate with a specific CRUD operation which maps to a http method: Create is Post, Retrieve is Get, Update is Put, Delete is Delete. Some endpoints, known as resources, accept more than one method, as they represent atomic data

resources in the system and require all four operations to manage. Endpoints, as opposed to resources, tend to support a single method tied to the purpose of the endpoint.

The most common endpoint request type is GET, used for retrieving data from the system. Both GET and DELETE use query string params which are inputting using the param 'p' which should contain the json request.

```
curl -i 'https://sana.picard.io/admin/stackId?p=%7B%7D' -H "X-Picard: X-Picard"

curl -d 'p={"test":"test"}' -G -v https://sana.picard.io/admin/stackId -H "X-Picard: X-Picard"
```

POST and PUT requests send their parameters using the payload body of the request. In this request use curl's verbose mode to show all request and response headers, including how the cookie is set.

```
curl -d '{"name": "brig@trialomics.com", "password": "secure_password"}' -H "Content-Type: application/json" -H "X-Picard: X-Picard" -v https://sana.picard.io/auth/login
```

API Request Headers

Each API request to Picard needs to contain the following request headers:

```
Unauthenticated Request Header:
  Referrer: https://sana.picard.io
  User-Agent: Custom for each agent and trial
  X-Picard-API: Custom for each trial
```

API requests that require authentication will do so via an additional header named X-API-Key that is set equal to an API Key retrieved from the server:

```
Authenticated Request Header:
  Referrer: https://sana.picard.io
  User-Agent: Custom for each agent and trial/app
  X-Picard-API: X-Picard-API
  X-API-Key: api key goes here
```

App Configs

Picard has a set of default app configuration variables. These variables are used to configure your application, including how users are created, how long sessions last, etc. See below for a description of the default configuration variables.

- **USER_CREATION_METHOD** - Method by which user accounts are created.
 - admin (default) - only administrators are able to create users. The admin control panel facilitates this process by allowing the admin to either create the user with a preset password, or to generate a random human memorable password and send a welcome email to the user with the password. On first login, the users are forced to change their password to proceed. Users in the 'admin' group are exempt from this check and should be reminded to change the password before use.
 - user - users are able to create their own accounts. They enter a username and password and receive an email acknowledging their account has been created and welcoming them to use the site. The content in the welcome email can be modified via the admin console or via the API. In this situation admins can restrict resource access for new users until they have validated their emails or signed a terms of use.
- **TERMS_ENFORCE**
 - Boolean flag used to determine if system requires Terms of Service acceptance for use. If True, then users must have agreed to the most recent terms of use before accessing resources. Default value is False. If True and a user has not agreed to the most recent terms of use then they will receive 423 response codes to all endpoints requiring valid sessions.
- **SITE_NAME**
 - Name of instance, used on outgoing emails and can be fed to web frontend. Default is Picard.
- **SESSION_PROTECTION**
 - Method by which sessions are protected, must be one of the following:
 - none: No session protection, cookies are accepted from anywhere to identify user. Least secure option, but ideal for mobile devices where the user is moving between networks often.
 - basic (default): If session secure hash does not match, session is marked as non-fresh, and user must log-in again to proceed.
 - strong: If session secure hash does not match the entire session is invalidated requiring new login.
- **SESSION_LIFETIME**
 - Life span of a normal session in seconds. Default value is 259,200 seconds which is equal to 3 days.
- **REMEMBER_LIFETIME**
 - Life span of session when user specifies remember=True during login.
- **SECRET_KEY**
 - Secret key used for cryptographic seeding, changing invalidates existing sessions.

- **REQUIRE_EMAIL_VALIDATION**
 - Flag to require email validation for users on first login (True/False). If set to True and user has not validated email then they will receive 423 response codes on all calls to endpoints requiring valid sessions.
- **PASSWORD_MIN_LENGTH**
 - Minimum password length. Default value is 6.
- **FRONTEND_BASE**
 - Frontend url for embedding in mail and other link redirects. Default value is 127.0.0.1.
- **ACCEPT_APIKEY**
 - Flag to determine if system accepts apikeys as well as sessions (True/False).

Custom API Error Codes

Picard throws the following HTTP error codes. If you identify situations where an error code is incorrectly thrown please let us know!

HTTP Status Code	Message	Explanation
400	Mail sending error	Attempt to send a mail fails
400	Invalid JSON	Invalid json is sent in the p
409	Document Conflict	Attempt to create or update an entity violates key constraints
404	Document Not Found	Document is not found
401	Permission Denied	User is not sufficiently authorized
500	Unable to execute due to database error	A db operation fails in a try catch
400	Invalid field	User tries to add new field to mapping with dynamic equal to False
400	Failed creating doctype	Failed creating new doctype
400	Invalid Parameter	Invalid or missing parameter exception
423	Email validation required	User must validate their email before they can access resources.

423	User password change required	User needs to change their password before they can access resources.
423	Terms of service review required	User must agree to the most recent terms of service before they can access resources.

User Accounts

Picard enables user account creation, identity verification (through validating user emails), and account management. The `USER_CREATION_METHOD` config variable controls how accounts are created. If it is set equal to 'user', then unauthenticated users can create accounts by submitting POST calls to the `/auth/user` resource. If the `USER_CREATION_METHOD` config is set to admin, then only authenticated users with admin permissions can create accounts.

When admins create an account they can chose to define the users password or have it defined by the server via the "random_password" parameter. When the password is defined on the server the user automatically receives an email that includes their password in cleartext. When the user logs in for the first time they will be forced to update their password, ensuring the password is only used once.

Internally Picard has a model for each user. The attributes of the model cannot be changed as its sole purpose is to model the auth status of the user. Apps that require other types of user data, like gender, age, or billing info, should do so via a custom doctype.

The user model consists of the following fields:

```
{
  active: boolean value indicating whether user account is active. Accounts can only be
deactivated by users in the admin system role.
  created: date account was created.
  current_login: date current session started.
  current_login_ip: IP address of current session.
  email: email address provided during login.
  id: UUID identifier used throughout the application for linking documents and other
data to this user.
  last_login: last logn date
  last_login_ip: last IP address used to login
  login_count: number of sessions created
  modified: last date user model was updated.
  name: unique user name defined at account creation.
  password_change_next_login: boolean value indicating if user needs to change
```

```
password on next login.
```

```
  validated: boolean value indicating if the users email address has been verified. If the  
  app config variable is set to true then this value must be true.
```

```
  extras: json object used to store arbitrary data
```

```
}
```

After the user model has been created the only attribute that can be modified is the user's email and the extras json blob. Only admins can delete user accounts. We highly recommend accounts are never deleted, rather admins should disable the user account, which disables their ability to access any resources until the admin reenables their account. If you do decide to delete the account then be aware that any custom data generated by this user (in logs or doctypes) will be untraceable.

Identity (Email) Verification

Certain applications need users to validate their email address. The global config variable `REQUIRE_EMAIL_VALIDATION` can be set equal to `True` in order force users to validate their email after first login. If a user has been asked to verify their email but yet to do so they will receive 423 error codes on each request. To validate their emails users need to use the `emailValidation` resource. Read below for more details on how this works.

Session Management

Picard has two methods of session management named cookies and api keys. The former is always available, while the latter can be activated or deactivated via the `ACCEPT_APIKEY` global config.

Cookies

By using Session Auth, we eliminate the need to send passwords or api-keys on each individual request. To start a new session a user needs to send a request to the login endpoint with the username and password. The auth token is returned in the header of the response in the form of a cookie. This auth token can then be used for authentication in subsequent requests. The auth token is cleared from the server when a user logs out, rendering it useless in subsequent requests.

The users device will use the session response header value in subsequent api requests. Each session has an expiration date or lifetime. As such, the session cookie obtained from a POST call to the `/auth/login` endpoint expires. Afterwards the user would have to re-login to obtain a valid session. The expiration duration of session cookies can be modified through the `SESSION_LIFETIME` global configuration variable.

The `/auth/login` can also accept a third parameter called `remember`. When this value is set to `true` the session cookie returned in the header response will be valid for an additional amount of

time. The exact amount is defined through the REMEMBER_COOKIE_DURATION global configuration.

Api Keys

Picard also offers API key based authentication. While this method is less secure than the session based auth described above, it's useful for certain applications in which it's inconvenient for users or devices to login. By default, each Picard instance does not allow API key based access. Admins interested in enabling this method can do so by setting the ACCEPT_APIKEY global configuration variable equal to true.

Session Protection

Picard allows for varying levels of session protection, per your application needs. Session protection prevents other individuals from stealing the session information and using it themselves to hijack the valid users account. Session protection can be configured by setting SESSION_PROTECTION in the config control panel to one of the following:

- None: No session protection, cookies are accepted from anywhere to identify the user. Least secure option, but ideal for mobile devices where the user is moving between networks often.
- Basic (default): If session secure hash does not match, session is marked as non-fresh, and user must log-in again to proceed.
- Strong: If session secure hash does not match the entire session is invalidated requiring new login.

At a high level, all sessions are originally marked as fresh. A secure hash of IP address and user agent is kept in the server side session, which is loaded when the user presents a valid cookie. If a user changes browsers or computers with a valid session cookie, their session is marked non 'fresh'. Certain functions, like changing a users password or access administrator functions, require a 'fresh' session. Attempts to access these functions will result in a 401 PERMISSION DENIED stating that a session is not fresh. This error should be caught by your application and force the user to reauthenticate.

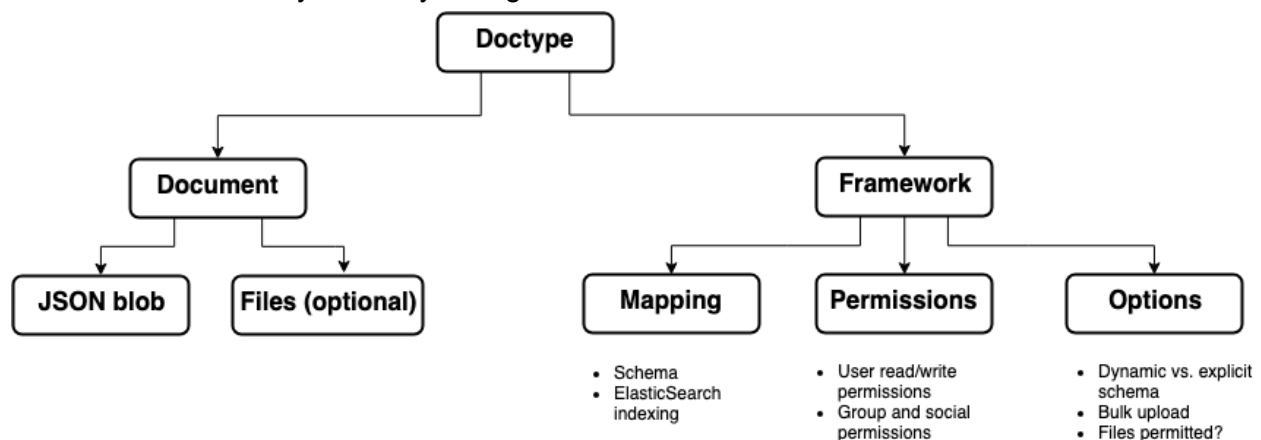
Data

Picard provides a flexible and powerful service for working with data. Under the hood Picard leverages Postgres and ElasticSearch, the former for its reliability and security and the latter for its search and compute capabilities. Picard also has the ability to act as an authentication gateway to the massive, high performance file store AWS Simple Storage Service, aka Amazon S3. While this service provides cheap storage, it lacks easily integrated authenticated services. Picard provides the underlying glue to allow you to take advantage of this complex technology in a secure fashion.

Doctypes

Picard organizes data into Doctypes. Here is a brief introduction to Doctypes.

- Doctypes are conceptually similar to tables in relational databases or collections in MongoDB in that they provide a means to logically group documents based on their contents.
- Each Doctype has a framework, which is a mapping or schema with extra features to facilitate complex search and analysis, permissions to control access, and options that define rules on how data can be added.
- Each Doctype has one or more Documents, each of which is a unique instance of data that consists of field:value pairs and/or binary files. Note every field in a Document exists in the Doctype's mapping. And mappings can be configured to enforce an explicit schema or to dynamically change when new Documents are added with new fields.



Mappings

As mentioned earlier, each Doctype has a Mapping. Mappings enable Admins to customize how data are stored and indexed for search and analysis. The power of Mappings, and their complexity, lie in how they define properties for fields that contain strings, objects, and lists of objects. Users should consult [Elasticsearch's Mapping documentation](#) for more information.

Permissions

Picard has a permission scheme which enables fine grained control over who can read and write data to a given Doctype.

- The read permission defines which users can read data. Reading data happens when a user is searching or analyzing.
- The write permission grants the ability to write data. Note, this includes creating, updating, and deleting data.

Admins set permissions on Doctypes at creation. The permissions can be subsequently updated if the need arises. The table below defines the different permission settings for read and write.

PERMISSION	READ	WRITE
Public	Unauthenticated users can read all documents of this doctype	Not allowed
Owner	Users can only read their own documents	Any authenticated user can create new documents, but only the user who created the data can update or delete it.
User	Any authenticated user can read all documents	Any authenticated user can create new data, and any authenticated user can update, or delete all the data.
Grouped	Users in approved usergroups or system roles can read. See below for more information.	Users in approved usergroups or system roles can create, update or delete data. See below for more information.

While public, owner, and user permission types are self-explanatory, the grouped permission is a bit more complex. In the next section we introduce usergroups and system roles before finishing with a thorough overview of grouping based permissions.

Usergroups and System Roles

Usergroups are a powerful tool for developing community within your app, allowing users to define access to the resources they create and own. Usergroups can only be created by Admins. Users make use of usergroups by creating an instance of the usergroup and then adding users.

Once a usergroup has been created every user has an instance of that usergroup that does not contain any other users. For example, if an admin creates a friends group every user has an empty friends group. Users can then manage their usergroup instance by making calls to the `/auth/usergroupmember` endpoint resource.

The most basic example of System Roles is the Administrators role. Being a member of this role allows access to the endpoints and resources that power the administrator console. Any user empowered with this Role can act as a super user: reading, writing, and deleting all resources in the system.

At a more atomic level, System Roles can be used to define access control to documents. If a document type has grouped write access limited to a specific Role, only members of this Role can create, update, and delete docs from that doctype. Similarly, in terms of read access, a

grouped read access would only allow group members to read documents on the merit of being in that group.

Groupings

Grouped permissions on doctypes are very common in modern applications. For example, they underlie many familiar concepts like shared permissions for google docs, friends ability to access photos on facebook, and ability of your professional network on linked-in to endorse skills. They also allow admins to restrict doctypes to the internal workforce, for example internal BI dashboards or results generated to model and understand KPIs.

Groupings enable admins to define how usergroups or system roles can read or write documents of a given doctype. At a low-level groupings consist of four parts:

- A doctype with either read or write permissions set to grouped.
- Either a usergroup or system role that needs access to the doctype.
- A permission which must be one of read or write, with the requirement that the corresponding permission setting for the doctype be equal to grouped. For example, if the doctypes read permission is owner admins cannot grant the workforce read permission.
- A type, which specifies the users which have the permission to access documents of the doctype. This value must be one of usergroup, role, or roleuser.

Admins should be certain to understand the meaning of the different combinations of grouping types and permissions. This is especially true for write permissions as any user granted write access can update or delete documents. For read permissions this becomes critical for products that enable users to manage access to view documents, like images, posts, health data, etc. We present the six combinations of grouped types and permissions as well as their implications on accessing documents in the table below.

	READ	WRITE
Usergroup	User B can read User A's documents if User B is a member of User A's instance of the corresponding usergroup.	Any authenticated user can create documents and User B can update or delete documents created by user A if User B is a member of User A's instance of the corresponding usergroup.
Role	Users in the approved system role can read all documents regardless of their owner.	Only users in the approved system role can create documents, but any user with this role can update and delete all documents regardless of their owner.

Roleuser	Only users in the corresponding system role can read documents with the added restrictions that approved users cannot read each others documents.	Only users in the approved system role can create, update and delete their own documents. Approved users cannot update or delete documents created by other users.
----------	---	--

Creating Documents

There are two methods for creating documents on Picard, with the sole difference being in one case the document is given a name parameter. When a document has a name parameter no other documents can be created for the doctype using the same name. This comes in handy when you want to restrict certain data to a single instance; for example a user profile document could be created with the name set to the user UUID or email.

To create a new document *without specifying a name*, applications should make POST requests to the /custom/endpoint document and only include parameters for the docytype name and a JSON object to store on the document itself.

```
Request Header: Authenticated
Method: POST
Endpoint: https://sana.picard.io/custom/document

REQUEST PARAMETERS:
{
  "doctype": "exampleDocType",
  "document": {
    "key": "version1"
  }
}

STATUS: 201
RESPONSE BODY:
{
  "_id": "5223e07acf28555e90c2c26394f21910",
}
```

To create a new document *while specifying a name*, applications should make POST requests to the /custom/endpoint document and only include parameters for the docytype name, the document name, and a JSON object to store on the document itself.

```
Request Header: Authenticated
Method: POST
Endpoint: https://sana.picard.io/custom/document
```

REQUEST PARAMETERS:

```
{
  "doctype": "exampleDoctype",
  "document": {
    "key": "version1"
  },
  "name": "exampleDocument"
}
```

STATUS: 201

RESPONSE BODY:

```
{
  "_id": "5223e07acf28555e90c2c26394f21910",
}
```

The value of the `_id` key in the response body is the unique UUID for the document. We can use this to obtain the document. We can also use the document name value as well. See the next section for how to do this.

For either method, when a document is created it is stored in an encrypted fashion in three distinct locations: Postgres, ElasticSearch, and Amazon S3. The Amazon S3 copy leverages Cross-Region Replication (CRR) ensuring all trial data can be recovered from a truly catastrophic disaster affecting Amazon's US-East1 availability zone.

A log document also exists capturing who created it, when, from where, etc. As stated earlier these cannot be updated or deleted and are used to ensure traceability.

Here are the messages, status values, and causes of errors returned by the server. If you find one that isn't here please let us know!

Message	Status	Cause
Permission Denied	401	Calling user does not have permissions to write to the doctype provided in the request payload.
Unable to create document object from Document model.	400	Issue with one or more of the values set as attributes on the Document model.

check mapping to find right type for this field	400	If document provided in the request payload contains a field with a value that conflicts with the fields existing type.
strict_dynamic_mapping_exception	400	If the doctype is set to not allow new field types and the document provided in the request payload contains a previously unseen field name.
Unable to add document due to database error	400	Unexplained error. Thrown if issue with writing to database.
Unable to create document, already exists	409	Thrown if document id conflicts with an existing document id.

Obtaining Documents

To get a document via its id value you need to call:

```
Request Header: Authenticated
Method: GET
Endpoint: https://sana.picard.io/custom/document
```

REQUEST PARAMETERS:

```
{
  "id": "5223e07acf28555e90c2c26394f21910"
}
```

If the call is successful the server will respond with the following:

```
STATUS: 200
RESPONSE BODY:
{
  "id": "5223e07acf28555e90c2c26394f21910",
  "body": {
    "key": "version1"
  }
}
```

To get a document using its document name you need to call:

```
Request Header: Authenticated
Method: GET
Endpoint: https://sana.picard.io/custom/document
```

REQUEST PARAMETERS:

```
{
  "doctype": "exampleDoctype",
  "name": "exampleDocument"
}
```

If the call is successful the server will respond with the following:

```
STATUS: 200
RESPONSE BODY:
{
  "id": "5223e07acf28555e90c2c26394f21910",
  "body": {
    "key": "version1"
  }
}
```

If the call fails you will see one of the following status codes and corresponding error messages:

Message	Status	Cause
Permission Denied	401	Calling user does not have permissions to write to the doctype provided in the request payload.
Document not found	404	Document specified by _id or name,doctype combination cannot be location.

Updating Documents

To update a document via its id value you need to call:

```
Request Header: Authenticated
Method: PUT
Endpoint: https://sana.picard.io/custom/document
```

REQUEST PARAMETERS:

```
{
  "id": "5223e07acf28555e90c2c26394f21910",
  "document": {
    "key": "version2"
  }
}
```

If the call is successful the server will respond with the following:

STATUS: 200

RESPONSE BODY:

```
{ }
```

To update a document via its document name you need to call:

Request Header: Authenticated

Method: PUT

Endpoint: <https://sana.picard.io/custom/document>

REQUEST PARAMETERS:

```
{
  "doctype": "exampleDoctype",
  "name": "exampleDocument",
  "document": {
    "key": "version2"
  }
}
```

If the call is successful the server will respond with the following:

STATUS: 200

RESPONSE BODY:

```
{ }
```

If the call fails you will see one of the following status codes and corresponding error messages:

Message	Status	Cause
Permission Denied	401	Calling user does not have permissions to write to the doctype provided in the request payload.
Document not found	404	Document specified by _id or name,doctype,doctype name combination cannot be location.
Unable to create document object from Document model.	400	Issue with one or more of the values set as attributes on the Document model.
check mapping to find the right type for this field	400	If document provided in the request payload contains a field with a value that conflicts with the fields existing type.
strict_dynamic_mapping_exception	400	If the doctype is set to not allow new field types and the document provided in the request payload contains a previously unseen field name.

Obtaining Previous Versions of Documents

Imagine three users have write access to the same Doctype. Let's assume the first user creates an initial document, then the second user updates it, before the third user updates it to the most recent version. For simplicity, we'll continue to use the example from above and assume first sets the value of the key field to version1, the second user updates the value to version2, and the third user updates it to version 3.

We can request a list of version ids of a document by including a field named versions with value equal to true in a GET request to /custom/document endpoint. Note if you want to access the document by its document name you can replace the id field with the doctype name and document name as shown above.

Request Header: Authenticated
Method: GET
Endpoint: <https://sana.picard.io/custom/document>

REQUEST PARAMETERS:

```
{
  "id": "5223e07acf28555e90c2c26394f21910",
  "versions": true
}
```

```
}
```

STATUS: 200

RESPONSE BODY:

```
{  "body": {  "_created": "2019-03-08T22:29:19.284Z",
    "_modified": "2019-03-08T22:29:19.435Z",
    "_modified_by": "531e794efc92504a9351d2d39a5f66b1",
    "_owner": "b25143d1fcf65fa2862526e3b38965e1",
    "key": "version3.txt"},
  "created": "2019-03-08 14:29:19.284000",
  "doctype": "d11dd951da895d888b52b1764fd93664",
  "id": "a5a746fc5aa059d6ac32734a6b70bfba",
  "modified": "2019-03-08 22:29:19.435000",
  "modifiedby": "531e794e-fc92-504a-9351-d2d39a5f66b1",
  "owner": "b25143d1fcf65fa2862526e3b38965e1",
  "versions": {  "IsTruncated": False,
    "KeyMarker": "",
    "MaxKeys": 100,
    "Name": "picardio-redbirdtest-cfs",
    "Prefix": "else/a5a746fc5aa059d6ac3/a5a746fc5aa059d6ac3.json",
    "VersionIdMarker": "",
    "Versions": [  {  "ETag": "\"bb1d36ce61ad2efea2b309d7c7e9bbae-1\"",
      "IsLatest": True,
      "Key": "else/a5a746fc5aa059d6ac3/a5a746fc5aa059d6ac3.json",
      "LastModified": 1552084160,
      "Owner": {  "DisplayName": "awsadmin",
        "ID": "6aa3a1dd5d611f27b"},
      "Size": 202,
      "StorageClass": "STANDARD",
      "VersionId": ".FQ0ZIKZ8UDrj.0i..BUUQD7p8H7AFQO"},
    {  "ETag": "\"a1e8854755ba7cdf38c1978eab75f43e-1\"",
      "IsLatest": False,
      "Key": "else/a5a746fc5aa059/a5a746fc5aa059.json",
      "LastModified": 1552084160,
      "Owner": {  "DisplayName": "awsadmin",
        "ID": "6aa3a1dd5d611f27b52"},
      "Size": 202,
      "StorageClass": "STANDARD",
      "VersionId": "XxoF.zxGCOQjwQF2CfrdCMwYmemnxo9y"},
    {  "ETag": "\"f03ab9b9cfb9b74c340ba27607adf5d9-1\"",
      "IsLatest": False,
      "Key": "else/a5a746fc5aa/a5a746fc5aa.json",
      "LastModified": 1552084160,
```

```
"Owner": { "DisplayName": "awsadmin",
            "ID": "6aa3a1dd5d611f27"},
"Size": 202,
"StorageClass": "STANDARD",
"VersionId": "MuxtBj_EnojZNp4w.E9ZD30z0j4tM36H"}]]}}
```

The response contains a field named `versions` that has a field named `Versions` which contains a list of up to the last 100 versions of this document. The versions are ordered in descending chronological order, so the first version is the most recent.

To retrieve the content for a specific version, the application should use the `/files/file` endpoint and specify the filename as the document id prepended to the string `'.json'`, the id equal to the document id, and `version_id` equal to the specific `VersionId` from the versions list.

```
Request Header: Authenticated
Method: PUT
Endpoint: https://sana.picard.io/custom/document

REQUEST PARAMETERS:
{
  "filename": "5223e07acf28555e90c2c26394f21910.json",
  "Id": "5223e07acf28555e90c2c26394f21910"
  "version_id": "wQLaNJQ4.xayHXdK86Sr81swBgo3vaWY"
}

STATUS: 200
RESPONSE BODY:
{
  "url": "https://picardio-stack-cfs.s3.amazonaws.com....",
}
```

The URL can be used to access the version document. If we perform the previous call three times, once for each element in the **versions.Versions** list returned from the GET call, and retrieve the content by downloading the URL we will obtain the following:

```
// version id: .FQ0ZIKZ8UDrj.0i..BUUQD7p8H7AFQO
{
  "_created": "2019-03-08T22:38:18.371Z",
  "_owner": "b25143d1fc65fa2862526e3b38965e1",
  "_modified_by": "531e794efc92504a9351d2d39a5f66b1",
  "key": "version3",
  "_modified": "2019-03-08T22:38:18.528Z"
}
```

```
// version id: XxoF.zxGCOQjwQF2CfrdCMwYmemnxo9y
{
  "_created": "2019-03-08T22:38:18.371Z",
  "_owner": "b25143d1fcf65fa2862526e3b38965e1",
  "_modified_by": "64430ef7ac5c5779b764ac77c11b3723",
  "key": "version2",
  "_modified": "2019-03-08T22:38:18.451Z"
}

// version id: MuxtBj_EnojZNp4w.E9ZD30z0j4tM36H
{
  "_created": "2019-03-08T22:38:18.371Z",
  "_owner": "b25143d1fcf65fa2862526e3b38965e1",
  "_modified_by": "b25143d1fcf65fa2862526e3b38965e1",
  "key": "version1",
  "_modified": "2019-03-08T22:38:18.371Z"
}
```

Applications can update the current version of a document using an older version by performing the above operations to get the desired version, then simply update the document via a PUT call.

Attaching Files to Documents

To store data in S3 the application needs to first request a prevalidated URL from Picard. This request requires a filename which we set equal to the name of the corresponding documents id appended with the string '.json'. Here's an example for the document returned in the above example.

```
Request Header: Authenticated
Method: POST
Endpoint: https://sana.picard.io/files/file

REQUEST PARAMETERS:
{
  contenttype: "application/json"
  filename: "example.json"
  id: "5223e07acf28555e90c2c26394f21910"
}
```

If the call is successful the response will contain a 201 with a prevalidated Amazon S3 URL.

```
Status: 201
```

REQUEST BODY

```
{
  "url":"https://picardio-sana-cfs.s3.amazonaws.com/ap....
}
```

Applications can use the URL to upload the contents of the example.json file. Applications should support **multipart upload for large files** if they expect to upload large files, such as audio or video content.

Deleting data

While we do not support deleting data for an actual Trial, during testing documents can be deleted via the id value only.

Request Header: Authenticated

Method: DELETE

Endpoint: <https://sana.picard.io/custom/document>

REQUEST PARAMETERS:

```
{
  "id": "5223e07acf28555e90c2c26394f21910"
}
```

If the call is successful the server will respond with the following:

STATUS: 200

REQUEST PARAMETERS:

```
{}
```

If the call fails you will see one of the following status codes and corresponding error messages:

Message	Status	Cause
Permission Denied	401	Calling user does not have permissions to write to the doctype provided in the request payload.
Document not found	404	Document specified by _id or name,doctype,doctype combination cannot be location.

Electronic Signatures

<u>Electronic Signature</u> <ul style="list-style-type: none">• SIG_NAME• SIG_DATE• SIG_MEANING• SIG_ID	<u>Electronic Link</u> <ul style="list-style-type: none">• SIG_ID• RECORD_ID	<u>Electronic Record</u> <ul style="list-style-type: none">• SIG_NAME• SIG_DATE• SIG_MEANING• SIG_ID• RECORD_ID• RECORD_DATA
--	---	---

The Flask App enables electronic signatures to be captured and assigned to custom data. The flow is as follows:

1. A user creates an electronic signature via a POST request to the /custom/electronicSignature endpoint. The request needs to include the current date and time, the calling users username and password, the patient UUID for which they are signing data, the meaning of the signature, and an optional JSON blob containing any extra information. If successful, the response to this request contains an electronic signature id that uniquely represents the electronic signature.
2. Next, the same user using the same session creates the electronic record and passes in the electronic signature id. The server checks the following and if any fail the record is denied:
 - a. Is the same user who created the signature creating the record?
 - b. Was the signature created no more than 60 seconds ago?
 - c. Does the patient UUID match on the electronic record match the patient UUID on the electronic signature?
 - d. Does the meaning of the electronic record match the meaning of the electronic signature?
 - e. Is the calling IP address the same as was used to create the electronic signature?
3. If the checks all pass, then the server creates the electronic record as a custom document, as well as a link document that associates the electronic signature id with the custom document id for the newly created electronic record.
4. The user can create as many electronic records as they want and associate them all to the same electronic signature as long as the above checks have been met.
5. Electronic signatures and records cannot be updated or deleted once they have been created.

Picard Web Console