# **Connected Operations (COps) Platform**

# Developer’s Guide

# 

# 

# **Skyward Federal**

# **CSC 492: Team 32**

# Jonathan Balliet

# Caleb Boswell

# Daniel Mills

# Jeen Shaji

# Spencer Yoder

## Content

1. Creating SELinux Security Policy Module
2. Integrating SELinux with COps
3. Configuring PostgreSQL for Course Manager and Docker on CentOS 7
4. Generating PostgreSQL Tables, Labels, and Data for Course Manager
5. Creating the Docker Image for Course Manager
6. Running Course Manager
7. Running Container Runtime
8. Installing and Updating Python Dependencies
9. Navigating COps Project Structure
10. Creating Database Models
11. Creating REST API Calls
12. Testing APIs with Curl Commands
13. Running Unit Tests
14. Running Coverage Reports
15. Accessing Remote Server With Firefox

## 

## Creating SELinux Security Policy Module

|  |  |
| --- | --- |
| Primary Author | Spencer Yoder |
| Secondary Author | Daniel Mills |
| Editor | Jeen Shaji |

### What is a SELinux Policy?

An [SELinux](https://www.redhat.com/en/topics/linux/what-is-selinux) policy, at its core, is a set of rules which define which processes (domains) can do which actions (permissions) to which objects (types). These concepts will be expanded on in a second. Every process and file in an SELinux system is assigned a [security context](https://fedoraproject.org/wiki/Security_context), which by itself is just a label. Policies define how contexts can interact in certain ways. By default, all actions which aren’t defined in the policy are not allowed to occur on the system.

### Definitions:

* *Type Enforcement* - This is what SELinux does. Type enforcement refers to fine-grained access control where processes of certain types can only interact with other types in certain ways.
* *Targeted Policy* - In SELinux, this is the default policy. The targeted policy specifies that all actions not defined in the policy are denied *except* for actions taken by processes running with the domain unconfined\_t. This domain is not affected by SELinux. This policy is activated by running sudo setenforce 1 if SELinux is set in targeted mode in /etc/sysconfig/selinux.
* *Permissive Policy* - The permissive policy enforces no selinux rules. *However*, it still logs every action which would be denied. This is a good policy to activate when debugging your policy and is activated by running sudo setenforce 0
* *MLS Policy* - This policy uses the *level* field of *security contexts*. If MLS is activated, it is used in tandem with the other active policy to further constrain data. There’s a little bit more about MLS [here](https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/selinux_users_and_administrators_guide/mls). We did not use MLS for our custom policy.
* *Security Context* - A label assigned to a Linux object (user, process, file, etc.) which is formatted as follows: *user:role:type:level*.
  + *User* - Has no correlation to a Linux user. SELinux users are simply a way to constrict which *roles* an object can be assigned. Certain users can only be assigned certain *roles*.
  + *Role* - Signifies which *types* an object can be assigned.
  + *Type* - For the *targeted policy*, which we used, this is the most important part of the security context. SELinux rules regarding access only apply to types. OS transactions occur between a source and a target (like a process and a file). When it’s applied to a target, it’s just called a type. When a type is applied to a source, it is called a *domain*.
    - *Class* - Every SELinux *type* has a class. A class just lets the policy know what kinds of things can be done to that type. For example, types of the ‘file’ class have actions read, write, execute, etc. associated with them.
    - *Domain* - The type of a running process. Policy rules identify which domains can interact with which types and how. Note that if a process wants to do something to another process, the second process’s domain is its type. A type is a domain only if the object with said type is the one performing the action.
  + *Level* - The last field of the *security context* only has meaning in the *MLS policy*. A level is comprised of a sensitivity range and a category range. There’s more about levels [here](https://selinuxproject.org/page/NB_MLS).
* *AV (Access Vector) Rules* - These are the rules which determine what happens when a particular domain tries to interact with a particular type in a certain way. When you create a policy, these are usually the meat of said policy. More about them [here](https://selinuxproject.org/page/AVCRules).
* *Audit Logs* - In SELinux, every transaction is logged. Audit logs are stored in /var/log/audit/audit.log but can be searched with *ausearch*.
* *Ausearch* - [This command](https://linux.die.net/man/8/ausearch) lets you search through the audit logs for particular elements.

We used *Ausearch* to debug our security policy. A strategy which worked really well was the sequence of commands where date gives the current system date and time, the command is some command we want to debug, the -i flag interprets certain parts of the log to convert them to ascii text, and the -ts flag signals ausearch to only search after the time we enter after it.

$ date

$ <command I want to debug>

$ sudo ausearch -i -ts <time specified from date> | grep <command name>

Figure 1 gives an example of what this looks like:

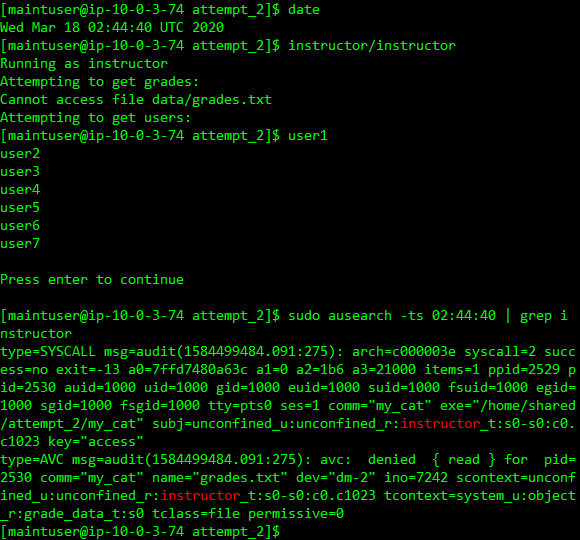


Figure 1: Example of Ausearch for debugging [[1]](#footnote-0)

### Tools:

Before you can create your own security policy, you need the tools.

* sudo yum install [libselinux-utils](https://www.mankier.com/package/libselinux-utils)

We installed this at a previous step, but a really helpful associated tool is:

* [setenforce](https://linux.die.net/man/8/setenforce)
  + Changes SELinux between enforcing (targeted) and permissive mode
* sudo yum install [setools-console](https://www.mankier.com/package/setools-console)

Pertinent tools:

* [seinfo](https://linux.die.net/man/1/seinfo)
  + Very helpful for finding out existing types, rules, etc. in your current policy.
* [sesearch](https://linux.die.net/man/1/sesearch)
  + Helpful for finding AV rules which apply to certain types/domains
* [sestatus](https://linux.die.net/man/8/sestatus)
  + Shows broad details about the policy SELinux is currently enforcing
* sudo yum install [policycoreutils-python](https://centos.pkgs.org/7/centos-x86_64/policycoreutils-python-2.5-33.el7.x86_64.rpm.html)

Pertinent tools:

* [semanage](https://linux.die.net/man/8/semanage)
  + Has the ability to add certain (not all) policy rules from the command line
* [semodule](http://man7.org/linux/man-pages/man8/semodule.8.html)
  + The tools which adds modules to the current policy and can be used to rebuild the current policy
* [sepolicy](http://man7.org/linux/man-pages/man8/sepolicy.8.html)
  + sepolicy generate is what helps us create the starter files for our new policy.
* We already had [coreutils](https://www.gnu.org/software/coreutils/) installed which contains
* [chcon](https://linux.die.net/man/1/chcon)
  + Which allows you to change the context of files much like chmod (if the policy permits it)

Before continuing, it’s important to make sure you have all these commands. If you’re missing any, you can google them to find out how to install them.

### SEPolicy Generate:

When you want to create a new policy, this is the first command you will run: [**sepolicy generate**](http://man7.org/linux/man-pages/man8/sepolicy-generate.8.html). Here’s what this will and won’t provide to you and how to run it.

Our team created a test policy to help us figure this stuff out. We created a very simple system: a program called my\_cat, written in C, which prints out the contents of a file to the terminal, two text files, one with grades, and one with usernames, and the following requirements:

* processes running as coordinator\_t should be able to access files with type grade\_data\_t and files with type user\_data\_t
* processes running as instructor\_t should only be able to access user\_data\_t
* processes running as student\_t should only be able to access grade\_data\_t

What we ended up doing was running *sepolicy generate* three times, one on an executable file for each domain. Here’s how to run *sepolicy generate*:

sepolicy generate --application <path to binary>

Ideally, your working directory should be an empty one, set aside for your policy files, *sepolicy generate* creates quite a few:

* application\_name.te

A type enforcement file. All of our changes ended up being in these kinds of files. This is where you write the rules for your new policy

* application\_name.if

An interface file. As per our knowledge, this file tells selinux how it interacts with certain types (like opening a new type of files).

* application\_name.fc

A ‘file context’ file. Handy in theory, it should assign new contexts to certain files, however we never got this to work as it should and ended up assigning file contexts with chcon

* application\_name.selinux.spec

This file seems to help with installing the new policy

* application\_name.sh

This is the file you run when you’re ready to compile and add your new module. Every time it’s run, it rebuilds the policy.

At the start, these files do very little. They create new types, application\_name\_exec\_t and application\_name\_t. Building the policy changes the context of your application binary file to application\_name\_exec\_t. The rest of the rules help give the file basic permissions (like the ability to read its own contents). Remember that selinux disallows all behavior not specified in the policy. Adding new types and rules will be up to you.

Depending on what you want to do, --application might not be the best option. We plan to create a policy for the postgres server, which right now is set up to run as a daemon. In that case, we would use --init.

Remember that *sepolicy generate,* makes template files. You will not be restricted based on the option you choose, but you may be denied certain basic accesses if you pick the wrong type. In that case, you would have to add them by hand.

Why did we create three new applications with three new policy modules? Initially, we just had the one policy module for one application. However, we found out about context inheritance and context transitions and decided that the best way to run one application from three different contexts would be to start each new application from a different context and then run our original application (my\_cat) so that it would inherit the context of these separate processes. The easiest way to achieve this was to generate three new policy modules, one for each application, to guarantee that a lot of the messy stuff was handled by these template files.

### Policy Language:

Since we’ve got a policy created, we can add our rules. On the [**PolicyLanguage**](https://selinuxproject.org/page/PolicyLanguage) section of the SELinux wiki, the only useful pages seem to be under these [definition links](https://selinuxproject.org/page/PolicyLanguage#Kernel_Policy_Language_Definition_Links). General syntax refers to .te files. To figure out what should go in the other files, you will need to do some googling. As per our current knowledge, a “.te” file can have the following structures

# This is a comment.

# These are declarations:

<type> <name>;

<type> <name> <modifier type> <mod>;

<type> <name> <modifier type> {<mod 1> <mod 2>};

# Declarations can define new SELinux context objects:

role coordinator\_r;

type coordinator\_t;

# Modify existing current objects:

role coordinator\_r types {coordinator\_t unconfined\_t};

# Or both at one time:

user coordinator\_r roles coordinator\_r;

# These are policy rules. We only ever used two of these

# in our test policies:

<rule> <domain> <type>:<class> <action>;

<rule> <domain> <type>:<class> {<action 1> <action 2>};

# This is an “allow” rule. Processes with domain coordinator\_t

# can read, write, open, and get attributes of files with type

# user\_data\_t

allow coordinator\_t user\_data\_t:file {read write open getattr};

# Imagine this line is in 12-point font

# This is a type transition rule. When processes with domain

# unconfined\_t execute files with type coordinator\_exec\_t, they

# transition to domain coordinator\_t

type\_transition unconfined\_t coordinator\_exec\_t:process coordinator\_t;

# Declarations inside a require block aren’t actually created

# in this policy. Pre-existing objects go here. This allows you

# to reference these objects without creating duplicates.

require {

<type> <name>;

}

# Example:

require {

role object\_r;

type unconfined\_t;

}

# object\_r and unconfined\_t already exist in the policy

# TE policy language allows the usage of macros.

macro\_name(<parameter>)

# Anything structured like this with parentheses and no

# semicolon is a macro and expands into multiple lines.

# Some macros are [here](https://selinuxproject.org/page/RefpolicyWriteModule) but I’ve not been able to find a

# comprehensive list

# We used the macro files\_type to assign our new types to

# the file class:

files\_type(grade\_data\_t)

Here is what our “.te” files for our new policies looked like: [coordinator.te](https://drive.google.com/file/d/1k4ZU7SxneCLf7PdEoEkr7gG8iSJcQqCx/view), [instructor.te](https://drive.google.com/file/d/1YOZrpF8p-Lt6Du1qYkcX2zf2tQls4gmM/view), [student.te](https://drive.google.com/file/d/1fki9NoW_ydNPEyS0_s1g58KgS4n17mOI/view). This is about all we can share about writing the policy, which is probably the hardest part. Rely heavily on the policy language wiki and whenever you get an SELinux denial, google it to find out what your policy is missing.

### Developing Your Policy:

We found that developing this policy according to the following strategy was very helpful:

1. Make a small number of changes to the policy files.
   1. An example would be to add a new type: type grade\_data\_t
2. Compile the policy with sudo sh policy\_name.sh
   1. If the policy compiles and builds successfully, proceed to step 4
   2. If not, proceed to step 3
3. Fix your compilation/build errors and then return to step 2
   1. The compilation error messages for these policies are usually straightforward. The only confusions we encountered were:
      1. Compilation errors for lines in macros. In this case, both the line number for the expanded policy and the line number for the unexpanded policy are displayed.
      2. For some reason, the policy would not compile unless user declarations were at the very end of the file. If you declare new selinux user types, make sure to do this, the compiler will not tell you what you did wrong.
   2. You may need to hunt for build errors. After compiling, the script prints out a pretty lengthy series of messages. It will mark build errors, though.
      1. The most common build error we got was for duplicate users/types.
4. Look for evidence that your changes have applied
   1. If you’ve built the policy for the first time, you can run  
      ls -Z <path to executable>  
      The -Z option shows the security context. If your executable was modified by the policy, it should have the type executable\_name\_exec\_t.
   2. One of the things we did was to add waiting for user input in the my\_cat executable. This way, we could launch another terminal, run our program, and run  
      ps -uZ on the first terminal to see the context for all running processes.
   3. The best evidence that changes have applied is your program not being able to run. If you run sudo setenforce 0 and your program can run, SELinux is preventing your program from running. Run sudo setenforce 1 to re-enable SELinux and continue to step 5.
5. Debug your policy
   1. The most common type of error we encountered was SELinux not allowing us to do things after building the policy
      1. To find out what’s being denied, you can run the sequence of commands laid out in the *ausearch* section of the definitions above.
      2. If you can’t find the problem, or grep doesn’t print anything, you can remove the pipe to grep and manually look through the [audit log entries](https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/security_guide/sec-understanding_audit_log_files), there shouldn’t be too many to look through.
      3. If you still can’t find the problem, you can run [sudo semodule -DB](https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/security-enhanced_linux/sect-security-enhanced_linux-fixing_problems-possible_causes_of_silent_denials)  
         to allow every single audit to be printed to the log. By default, [some messages do not get logged](https://selinuxproject.org/page/AVCRules#dontaudit). After you’re done debugging, make sure to run sudo semodule -B to allow the policy to ignore messages again. Logging every single message can affect the performance of your system.
      4. The prior three steps helped us debug all of our problems. If you’re looking through the whole audit log with --dontaudit disabled and still can’t locate the issue, this guide can no longer help you, good luck.
   2. The second most common type of error was our program being allowed to do things it shouldn't
      1. The first thing you should check is the program’s behavior when the line  
         permissive policy\_name\_t; is commented out in your program. This line in a .te file runs a permissive policy only for this domain.
      2. If your program is still doing things it shouldn’t be able to, run it in another terminal and then run ps -uZ to check its security context. Programs running as unconfined\_t were the prime cause of these kinds of error for our group. We made our my\_cat executable wait for user input in order to be able to check its running context before it terminated.
      3. Remember that, for new domains (like the one defined in your new policy), SELinux denies every operation. As mentioned previously, the template policy generated gives this new context some very basic permissions, like the ability to run as a context at all. If your program is accessing things it shouldn’t, it’s either:
         1. Running in a different domain
         2. Running as permissive
   3. In general, finding comprehensive documentation for SELinux policies has been very difficult, but finding fixes by googling specific errors was much easier. If you’ve managed to debug your specific change, start back again from step 1 until your policy is complete.
6. If you ever need to remove your policy, you can do so with sudo semodule -r <policy\_name>

### Resources:

* Our C files and TE files for our practice policy:
  + [coordinator.c](https://drive.google.com/file/d/1unZOuz20zTezfIbglp1trMf2m8DqXA_3/view)
  + [instructor.c](https://drive.google.com/file/d/1Uv0IaScgB5yYj99azSJTiv3SI0iOo3Ha/view)
  + [student.c](https://drive.google.com/file/d/1mZZN8_De71scHr__LWNf63vl5deNCjjV/view)
  + [my\_cat.c](https://drive.google.com/file/d/1kIsk_GuMFPqQSMQsAB8Ncm243ha0fFph/view)
  + [coordinator.te](https://drive.google.com/file/d/1k4ZU7SxneCLf7PdEoEkr7gG8iSJcQqCx/view)
  + [instructor.te](https://drive.google.com/file/d/1YOZrpF8p-Lt6Du1qYkcX2zf2tQls4gmM/view)
  + [student.te](https://drive.google.com/file/d/1fki9NoW_ydNPEyS0_s1g58KgS4n17mOI/view)
* A video showing how to build a simple policy
  + [example\_policy\_creation\_debug.mp4](https://drive.google.com/file/d/1tqFlnilLuthafD7JjkTGHSZfJfoFcqXX/view)

## Integrating SELinux with COps

|  |  |
| --- | --- |
| Primary Author | Spencer Yoder |

### Problem

## After learning to create an SELinux security policy, the next challenge was integrating it with COps. Unfortunately this is a more complicated problem than can be solved by the above process. Creating a policy with *sepolicy generate* is useful when you have a single process you want to label with a single security context. For multiple contexts, we had to use the multiple parent process trick with context inheritance. However, there’s not much room for such a pattern in a Flask application like CourseManager. Furthermore, CourseManager communicates with Postgres through unknown means. At a high level, it’s not easy to know what process should be labeled.

### Overview

## Integrating the rest of our application with SELinux involved the following steps:

## Creating an SELinux policy

## Labeling the data in the Postgres database

## Facilitating labeled communication between CourseManager and Postgres

## Performing system testing to debug and refine the security policy/labeling

### Creating an SELinux Policy

## CourseManager is an application which was designed to tackle as many different cases of controlled database access as possible, while still being simple to implement. Figure 2 shows, at a high level, how access was meant to be granted:

## 

## Figure 2: Figure explaining how access is granted

## (Dotted lines refer to read access, solid lines to read/write access)

## At a high level, coordinators are our system administrators. They can read/write to any object in the database. Instructors and students are two user-types who can each see some data and are denied visibility to some other data. In addition, neither can write anything to the database with the exception that instructors can modify student grades. To create an SELinux policy, these categorizations were broken into SELinux types:

## coordinator\_t

* + The type for a user accessing data as a coordinator
* instructor\_t
  + The type for a user accessing data as an instructor
* student\_t
  + The type for a user accessing data as a student

## coordinator\_data\_t

## Coordinator account data

## instructor\_data\_t

## Private instructor data

## student\_data\_t

## Private student data

## course\_data\_t

## Course data

## Public instructor data

## Public student data

## grade\_data\_t

## Student grades

## Access vector rules were written in our policy to enforce the following set of access rights:

## 

## Figure 3: read access rights for CourseManager’s security policy

## 

## Figure 4: write access rights for CourseManager’s security policy

## The creation of the policy for CourseManager differed from the process described in the ‘Creating SELinux Security Policy Module’ section of the document because that process described using sepolicy generate to create a template for a policy designed to apply to a single running process. We had no need for such a template since our rules were meant to be general and apply to any access made by certain domains to certain object types.

## This policy was created using a more general process for compiling and building policy modules, and only required the creation of a .te (type enforcement) file to serve as the file containing the policy rules. The commands for compiling and building a policy module are located in the same packages described in the ‘Creating SELinux Security Policy Module’ section and the specific commands are located in the compile\_alias.sh file, which if loaded via

## $ source compile\_alias.sh

## allows users to compile the CourseManager policy with

## $ secompile

## and add it to the system with

## $ sebuild

## See the installation guide for more information about these commands.

## The process for developing this policy is the same as the one described in ‘Creating SELinux Security Policy Module.’ With a major exception: since denials are occurring via SEPostgres, they are logged the Postgres’s logging destination (which on our system could be traced with the journalctl command. So the process for checking for denials by SEPostgres is as follows:

## $ date

## $ <command>

## $ journalctl -S <time specified by date>

## The -S argument is the equivalent to ausearch’s -ts argument. Of course, this only shows denials done by SEPostgres. If the Linux kernel is doing any denials directly, they still must be parsed with the *ausearch* command. A good way to determine which command to run is to see whether the process accessing the database is throwing any exceptions. If it runs into trouble during a database access, use journalctl, if not, use ausearch.

## A final, minor difference to point out about the CourseManager policy is the definition of classes in the *require* block:

## require { ... # SEPostgres classes

## class db\_table {search select drop create insert lock delete update};

## class db\_column {select drop create insert delete update};

## class db\_schema {search};

## class db\_procedure {execute};

## }

## These are the relevant SEPostgres classes which are not natively defined in SELinux. In order to refer to them in access vector rules, they, along with the permissions used, must be defined in the *require* block.

### Labeling the Data in the Postgres Database

## In order for the rules of the policy to be enforced, the database objects must be labeled. Postgres adds a query to facilitate this:

## SECURITY LABEL ON <TABLE, COLUMN, etc.> <name> IS '<label>';

## Database labels can be viewed with the following query:

## SELECT \* FROM pg\_seclabels;

## Initially, we used the security label query with the psql command to label all our data; however, we later automated it in our data generator:

## cops\_platform/services/course\_manager/tests/db/generate\_db.py

## with the following function:

## 

## As mentioned, we couldn’t find a way to parametrize this query, but the primary difficulty was figuring out how to apply it to the database. Initially, db.engine.execute() was used, which caused commands to be audited by SELinux, but no labels were actually set. We later figured out that we needed to use db.session.execute() followed by db.session.commit(). After designing the policy, we labeled the database as follows:

## 

## Figure 5: database labels

## This differed from our original plan in that much of the student and instructor data is marked as course\_data\_t. Instructors and students ended up needing to access this data to be able to view information about their courses and modify grades. Also, something we learned is that accesses in Postgres are hierarchical as far as SEPostgres is concerned. What this means is that if I want to read data from the student id column, I need permission to search the schema, select from the student table, and select from the id column. Our initial assumption was that all database objects allow accesses allowed by their parent, with the exception of those labeled differently than their parent. However, each object requires a label and a permission to be accessed.

## Finally, it’s worth noting that every database object not included in this diagram is labeled as unlabeled\_t. This means a lot of our policy rules deal with accesses for unlabeled data. This isn’t an issue for us, since the only application we enforce with SELinux is CourseManager. However, if we had multiple applications, it could be problematic to give a domain blanket access to all unlabeled database tables.

### Facilitating labeled communication between CourseManager and Postgres

## This was the process we had the most difficulty in completing. Our initial plan of attack was to figure out if there was some intermediate process between CourseManager and the Postgres server daemon which could be labeled at launch. What we found is that this is not the case. CourseManager communicates with Postgres directly via a TCP socket. We discovered this because, after moving CourseManager to our EC2 machines, the application would crash whenever attempting to communicate with Postgres. After extensive research, we found a similar problem on [this webpage](http://www.2done.org/44.html) (note: the page is in Japanese, it requires translation to read unless you can read Japanese). Detailing that SEPostgres checks for the context of its connections using the SELinux system call: [getpeercon](https://linux.die.net/man/3/getpeercon). If the connection is not labeled, the application will be unable to connect to the server. The solution is to use the following command (all on one line):

## sudo netlabelctl unlbl add default address:<IP address> label:<SELinux label>

## which labels communications from the given IP with the given SELinux label. (See the [netlabelctl man page](https://linux.die.net/man/8/netlabelctl) for further details).

## Because of this revelation, we stopped wondering what process to label and started attempting to figure out how to automatically label connections to the database based on the usertype. At this time, the container runtime had been integrated with the system on the EC2 machines, so, since each container is unique to a user, we decided to use the container runtime to assign a label to each container. Containers also have a unique IP for communication, so each IP could be labeled separately. Our final system for labeling connections looks like the following:

## 

## Figure 6: IP labeling system

## In this system, the container runtime uses a bash script: cops\_platform/container\_runtime/iplabel to label the connections of new containers based on their associated user and another script: cops\_platform/container\_runtime/clearlabel to remove the labels for containers which have shut down. To see exactly how this is done, read lines 20-24, 133, and 195-200 in cops\_platform/container\_runtime/app/controllers/service\_controller.py

## This method has some limitations, and was not our first choice for labeling containers. We wanted initially to use the [--security-opt argument](https://docs.docker.com/engine/reference/run/#security-configuration) for launching Docker containers but kept getting errors when attempting to use this method.

### System Testing Via the Terminal

## At the time of the creation of our policy, we had not created a front-end interface for our system. In order to test and develop our policy, we had to send API calls to the system through [curl commands](https://curl.haxx.se/docs/). To make this easy, a collection of bash functions was created and stored in

## cops\_platform/container\_runtime/curl\_shortcuts.sh running

## $ source curl\_shortcuts.sh

## will load these functions into the current bash environment and print out a description of each function. A sample execution would look like the following:

## $ source curl\_shortcuts.sh

## $ flask run &

## $ connect\_as coordinator

## $ add\_user smyoder "Spencer Yoder" student

## $ kill\_flask

## These functions were used in the script cops\_platform/container\_runtime/system\_tests.sh

## To create a full suite of system tests to determine whether SELinux was correctly controlling access to the system. This suite of tests can be run with

## $ sh system\_tests.sh

## and the output should (mostly) match the text in

## cops\_platform/container\_runtime/system\_test\_output.txt

## They will not be entirely identical, since not all output could be piped to system\_test\_output.txt.

## Configuring PostgreSQL for Course Manager and Docker on CentOS 7

|  |  |
| --- | --- |
| Primary Author | Jonathan Balliet |
| Editor | Jeen Shaji |

### Setup the host’s Postgres Database Permissions

### Make your postgresql listen to an external ip address.

### First find the location of postgresql.conf:

sudo find / -type f -name postgresql.conf

In our case, this was at the following location:

/usr/local/pgsql/data/postgresql.conf

* Now, open this file (with elevated permission) with a text editor to edit it. We will be using the nano editor:

sudo nano /usr/local/pgsql/data/postgresql.conf

Look for the line:#listen\_addresses = 'localhost' # what IP address(es) to listen on; Uncomment and set the external ip address/es that'll be accessing your DB. Since we are just having the Docker Container interface (at address 172.17.0.1) access this ip address and our localhost, we will change it to the following:

listen\_addresses = “172.17.0.1, localhost”

If you don't know it, or want to free all ips to access it (not safe), set it to '\*':

listen\_addresses = '\*' # what IP address(es) to listen on;

* Restart postgres

sudo systemctl restart postgres

You can check if this worked out with this command:

netstat -nlt

It will output something like this:

Proto Recv-Q Send-Q Local Address Foreign Address State

tcp 0 0 172.17.0.1:5432 0.0.0.0:\* LISTEN

In the example above, the PostgreSQL database is listening on the 172.17.0.1 address at the port 5432. Port 5432 is the default port used for PostgreSQL and indicates that PostgreSQL is listening at this address.

### Let your container access your postgresql database with a given user

### Find your pg\_hba.conf file

### $ sudo find / -type f -name pg\_hba.conf # => /etc/postgresql/9.5/main/pg\_hba.conf

In our case, this is at the following location:

/usr/local/pgsql/data/pg\_hba.conf

* Now, open this file (with elevated permission) with a text editor to edit it. We will be using the nano editor:

$ sudo nano /usr/local/pgsql/data/pg\_hba.conf #

or the path you found before

* Add the following line to the table of access permissions for this database:

host johndb john 172.17.0.0/16 trust

This allows the “john” user to connect to the “johndb” database from Docker containers (the ip addresses for Docker Containers consist of the range of addresses within 172.17.0.0/16).

* The *trust* method means that we will allow this connection without a password. In reality you would want to use the *md5* or *password* methods for security reasons.
* As noted previously, we are using *“johndb”* for the database and *“john”* for the user throughout these installation instructions for consistency. Make sure to replace it with the specific names for the database and user you are using.
* An additional line needs to be added to the table of access permissions. This is so the Container Runtime can access PostgreSQL through this specific database and user (in our case *johndb* and *john,* respectively). This is also necessary for Course Manager to run normally (not inside a Docker Container) if using a PostgreSQL connection and so it can generate and label the tables of this database via a script.
* Add the following line to the table of access permissions for this database:

host johndb john 127.0.0.1 trust

Instructions on how to set Postgres config files to accept connections with Docker:

<https://gist.github.com/MauricioMoraes/87d76577babd4e084cba70f63c04b07d>

### Change firewall rules to allow access

* On our CentOS 7 development machines, the following firewall changes were necessary to allow the Docker Container to connect to the database running on the host:

firewall-cmd --permanent --zone=trusted --change-interface=docker0

firewall-cmd --permanent --zone=trusted --add-port=4243/tcp

firewall-cmd --reload

* After entering these commands, restart the Docker service:

sudo systemctl restart docker

## Generating PostgreSQL Tables, Labels, and Data for Course Manager

|  |  |
| --- | --- |
| Primary Author | Jonathan Balliet |
| Editor | Jeen Shaji |

Successfully connecting the PostgreSQL database with the Course Manager application, requires a few additional configuration steps. In order for SEPostgreSQL to accept connections from a specific client, these connections must be labeled with SELinux on the host machine. Running the Course Manager application normally (in other words, not through Container Runtime), requires the localhost connection (at address 127.0.0.1) to be labeled with SELinux.

First install net-label tools on your machine:

sudo yum -y install netlabel\_tools

Now use the following command to label the 127.0.0.1 address:

sudo netlabelctl unlbl add default address:127.0.0.1 label:unconfined\_u:unconfined\_r:unconfined\_t:s0

You can confirm this was successful but using the following command:

netlabelctl unlbl list

The above added label for the address 127.0.0.1 should be listed after *“*accept on:*”*.

Next we are going to create the tables, along with some mock data, for the PostgreSQL database that Course Manager is going to use. To do this we will run a script that resets the state of the database. This script also labels all of the columns and tables of this database according to our SELinux policy. As a precondition, all the previous installation sections of this guide must be completed.

***NOTE:*** Whether you want to label the database columns and tables is based on whether the environment variable *ENFORCING* has been set.

Use the following command to set this environment variable:

export ENFORCING=true

This environment variable can be set to any arbitrary value, it simply must exist.

If you do not wish to label the database, you can use the following to unset this environment variable if it does contain a value:

unset ENFORCING

Change to following directory:

2020SpringTeam32/cops\_platform/services/course\_manager/tests/db/

Now use the following command:

python generate\_db.py

If no error was thrown, then the tables were successfully generated for the database. It’s important to note that everytime this script is run, the database is destroyed (the tables are dropped) and then recreated. These tables are created based on the model classes defined inside Course Manager. Some mock data for the database is also generated via this script file.

### Potential Issues

This script will not run if SELinux is in *enforcing* mode, but it will run if SELinux is in *permissive* mode. This is because we did not end up creating a SELinux Policy that allows all of the SQL commands running within this script to run on the labeled local host connection.

To check SELinux’s status, enter the following command:

sudo sestatus

Check to make sure permissive is the value of “Current mode”

Current mode: permissive

If not, enter the following to change to permissive mode:

sudo setenforce 0

If you are still encountering issues after doing the above, make sure all the *preconditions* have met (all previous installation steps have completed) and refer to the potential issues section for ***Confirming Successful Course Manager Installation***. It’s very likely it is either a missing dependency or environment error, which would need to be fixed in the same manner as listed in this section.

## Creating the Docker Image for Course Manager

|  |  |
| --- | --- |
| Primary Author | Jonathan Balliet |
| Editor | Jeen Shaji |

### *NOTE:* Whenever changes are made to Course Manager make sure to rebuild the Docker Image for Course Manager. This is the specific Course Manager application that will be launched in the Docker Container with the Container Runtime.

Make sure all of the previous installation steps have been completed.

### Build the Docker Image for Course Manager

* Change directory to *2020SpringTeam32/cops\_platform/services/course\_manager.* Enter the following command to build the Docker image for Course Manager:

sudo docker build -t course\_manager\_test:latest

This creates the docker image from the Dockerfile located in the *course\_manager* directory. The name of this docker image is *“*course\_manager\_test:latest*”*. Once a Docker image is created you can then run this image at any time using the “docker run” command.

### Common Docker Commands

* Test out if the docker container can connect to the Postgres Database. This command runs the Course Manager docker image as a background process (in detached mode)

docker run --name test -d -p 8000:5000 -e POSTGRES\_URL=172.17.0.1 -e POSTGRES\_USER=john -e POSTGRES\_DB=johndb -e POSTGRES\_PW=password -e ROLE=student course\_manager\_test:latest

The -p option designates which ports this container will run on.The docker container will be running on port 5000, which is the default port for FLASK applications, but this is linked to the host’s 8000 port. Therefore, you contact this running course manager application by sending a request to the host’s 8000 port on localhost.

The --name option designates the name of the container. If no name is provided, then the container will be assigned a randomized name instead. In this example we are using “*test*” as the name of our container.

The -e option is used for providing environment variables to the running docker container. These are vital so the running Course Manager application can connect to the database. All of the POSTGRES\_\*\* related environment variables are required in order for the application to contact the host’s database. The ROLE environment variable is optional and designates what the role of the user is that will be accessing this application. If none is given, it will default to “student” in the Course manager application.

course\_manager\_test:latest is the current name of this docker image that we built previously.

* The following command executes a command inside a running container. This allows us to connect to this container and open up a shell on this container (in other words, kind of sshing into this container)

docker exec -it test /bin/bash

This command requires the name of the container. In this case, we are using the container we created in the previous command that had the name of the *test*.

* This command lists all running docker containers

docker ps

Use the -a option with the above command list all containers (even ones that are stopped).

* This command deletes all stopped docker containers and anything in the docker cache

docker system prune

* This command deletes (and forces delete if it’s still running) the container with the name *‘test’*. Insert the name of the container you would like to delete instead.

docker rm -f test

## Running Course Manager

|  |  |
| --- | --- |
| Primary Author | Jonathan Balliet |
| Editor | Jeen Shaji |

### Precondition:

It is assumed all of the steps in the *Installation Guide* for installing COps, the PostgreSQL database, generating the tables for this database, and confirming the installation of Course Manager, have been completed.

*NOTE:*CourseManager is best run in Permissive Mode if using with the SELinux module enabled. This is because the Container Runtime automates enabling certain permissions, via labeling, when launching a Docker Container to run CourseManager with the authorization permissions of the user. You may have denials if running Course Manager normally with SELinux in enforcing mode.

A bash script, located in the root directory, 2020SpringTeam32, titled envs\_vars.sh, automates setting the envs (environment variables) that are needed to successfully run the flask Course Manager application on a CentOS 7 machine.

To run this script enter the following from the 2020SpringTeam32 directory:

source env\_vars.sh

However, in the following sub-section, we will go into detail about each one of these environment variables. You may wish to make changes to these for when different config settings are needed for running Course Manager. It is also important you understand the purpose of each one; some of these envs are required, while others are optional. They will each be indicated as such.

### Required Course Manager Envs:

* FLASK\_APP
  + This is needed in order for Flask to know which Python application to run.
  + Set to the value of “course\_manager.py”
* PYTHONPATH
  + This is needed for Python to be able to know the PATH for specific modules we are running.
  + Set to the value of path/to/root/directory/of/repo/Spring2020Team32
  + Example: PYTHONPATH=/home/maintuser/2020SpringTeam32
* LC\_ALL
  + This was required on our CentOS 7 development machines.
  + Set to the following: LC\_ALL=en\_US.UTF-8

### Optional Course Manager Envs:

* FLASK\_CONFIG
  + This is used to determine what database connection runs when Course Manager launches. Course Manager supports two database connections: SQLite and PostgreSQL. SQLite is built into Python and will be created by default after running the *db\_generate.py* Python script. Reference the section,***Generating PostgreSQL Tables, Labels, and Data for Course Manager,*** for running this script. It is required to run no matter which database connection you are using.
  + Not setting this env, or setting to the value of ‘testing’, will use the SQLite database.
  + Setting this env to ‘development’ will instead attempt to use the PostgreSQL database. If any of the required PostgreSQL envs have not been set, it uses the SQLite database connection instead. The config.py class, located in the course\_manager directory documents all of these environment variables in more detail.
* POSTGRES\_URL, POSTGRES\_PW, POSTGRES\_USER, POSTGRES\_DB
  + Setting all of these is required in order to connect to the PostgreSQL database running on the host machine.
  + Make sure the correct values are set for each. The ones we are using for this database connection are defined in the env\_vars.sh bash script.
  + For more details about what each is setting, check out the config.py class, located in the course\_manager directory.
* ROLE
  + This determines what the role the user is when launching this isolated instance of Course Manager.
  + By default this sets the env to the least privileged role of a student.
  + Set to one of the following values:
    - coordinator
    - instructor
    - student

### Launch Course Manager application

Change directory to the *course\_manager* directory in our repo. Example of the command to use starting from the root directory of our repo:

cd 2020SpringTeam32/cops\_platform/services/course\_manager/

Now run the following command to start Course Manager:

flask run

You should be greeted with a series of printed messages saying course\_manager.py is running on the local host at port 5000 (This is the default URL and port when running Flask). You can then stop running the Course Manager Flask application at any time by pressing “Ctrl-C” on the terminal.

To manually test the REST API endpoints for Course Manager, please consult the ***Testing APIs with Curl Commands*** section of this Developer Guide.

If you are encountering any errors with running Course Manager, please consult the ***Confirming Successful Course Manager Installation*** section of the Installation Guide. This contains common troubleshooting information for issues that could potentially occur.

## Running Container Runtime

|  |  |
| --- | --- |
| Primary Author | Jonathan Balliet |
| Editor | Jeen Shaji |

### Precondition:

It is assumed all of the steps in the *Installation Guide* are completed. Everything will need to be installed correctly in order for the Container Runtime application to successfully run.

***NOTE:*** Always make sure the following steps are true before attempting to run Container Runtime or test a Docker Container connection to the host PostgreSQL database depending on whether running in Permissive or Enforcing mode with SELinux.

Check SELinux enforcing status:

sudo sestatus

The current mode will be listed by the value of “Current mode”. In this example, it is running in permissive mode:

Current mode: permissive

The following command runs SELinux in permissive mode:

sudo setenforce 0

The following command runs SELinux in enforced mode:

sudo setenforce 1

### Required regardless of SELinux Mode:

The environment variables for the PostgreSQL connection running on the host machine must be set in order for the Container Runtime to be able to authenticate and authorize a user after a service request is sent. These are automatically set after running the env\_vars.sh script that is detailed in the previous ***Running Course Manager*** section.

### Running in Permissive Mode:

1. Make sure both the localhost interface (127.0.0.1) and the range of Docker Container IP addresses (172.17.0.1/16) are labeled with SELinux. You can check if these interfaces are labeled by using the following command:

sudo netlabelctl unlbl list

This command should have the following output:

accept:on interface:DEFAULT,address:172.17.0.0/16,label:"unconfined\_u:unconfined\_r:unconfined\_t:s0, interface:DEFAULT,address:127.0.0.1,label:"unconfined\_u:unconfined\_r:unconfined\_t:s0 "

If not, run the following commands:

sudo netlabelctl unlbl add default address:127.0.0.1 label:unconfined\_u:unconfined\_r:unconfined\_t:s0

sudo netlabelctl unlbl add default address:172.17.0.0/16 label:unconfined\_u:unconfined\_r:unconfined\_t:s0

1. Postgres is listening on both address 172.17.0.1 and 127.0.0.1 ***OR*** on the address 0.0.0.0

For some reason, when restarting our virtual machines, PostgreSQL does not automatically listen to these addresses. You can check if it is listening by entering the following command:

netstat -nlt

It will output something like this:

Proto Recv-Q Send-Q Local Address Foreign Address State

tcp 0 0 172.17.0.1:5432 0.0.0.0:\* LISTEN

tcp 0 0 0.0.0.0:5432 0.0.0.0:\* LISTEN

If the Local Address it is listening on is either 0.0.0.0:5432 (listening on all network interfaces at port 5432) ***OR*** 172.17.0.1:5432 (listening on the docker0 network interface at port 5432) and on 127.0.0.1:5432 (listening on localhost); then this connection should work. If it does not display this in the table of results, try restarting postgres. This would always work for us. Use the following command to restart Postgres:

sudo systemctl restart postgres

If this is still not working make sure you have added these to connect correctly in the Postgres configuration files as detailed in the previous ***Configuring PostgreSQL for Course Manager and Docker on CentOS 7*** section of this Installation guide.

1. The environment variable FLASK\_APP is set to container\_runtime.py and the environment variable ENFORCING is unset.

Set FLASK\_APP:

export FLASK\_APP=container\_runtime.py

Unset ENFORCING:

Unset ENFORCING

### Running in Enforcing Mode:

1. Make sure the localhost interface (127.0.0.1) is labeled with SELinux. You can check if this interfaces is labeled by using the following command:

sudo netlabelctl unlbl list

This command should have the following output:

accept:on interface:DEFAULT,address:127.0.0.1,label:"unconfined\_u:unconfined\_r:unconfined\_t:s0 "

If not, run the following command:

sudo netlabelctl unlbl add default address:127.0.0.1 label:unconfined\_u:unconfined\_r:unconfined\_t:s0

1. Postgres is listening on both address 172.17.0.1 and 127.0.0.1 ***OR*** on the address 0.0.0.0

For some reason, when restarting our virtual machines, PostgreSQL does not automatically listen to these addresses. You can check if it is listening by entering the following command:

netstat -nlt

It will output something like this:

Proto Recv-Q Send-Q Local Address Foreign Address State

tcp 0 0 172.17.0.1:5432 0.0.0.0:\* LISTEN

tcp 0 0 0.0.0.0:5432 0.0.0.0:\* LISTEN

If the Local Address it is listening on is either 0.0.0.0:5432 (listening on all network interfaces at port 5432) ***OR*** 172.17.0.1:5432 (listening on the docker0 network interface at port 5432) and on 127.0.0.1:5432 (listening on localhost); then this connection should work. If it does not display this in the table of results, try restarting postgres. This would always work for us. Use the following command to restart Postgres:

sudo systemctl restart postgres

If this is still not working make sure you have added these to connect correctly in the Postgres configuration files as detailed in the previous ***Configuring PostgreSQL for Course Manager and Docker on CentOS 7*** section of this Installation guide.

3. The flask\_app.sh bash script has been run. This automatically sets the ENFORCING environment variable and the FLASK\_APP environment variable to container\_runtime.py. This script is located in the *2020SpringTeam32/cops\_platform* directory and can be run with the following command:

source flask\_app.sh

### Launch Container Runtime flask application

Make sure the docker service is running. Change directory to 2020SpringTeam32

* Start running the virtual environment:

source env\_vars.sh

* Change directory to cops\_platform/container\_runtime
* If you want to run the application with SELinux enforced:

source flask\_app.sh

This script sets the ENFORCED env to being true and the FLASK\_APP env to being the container\_runtime.py.

* Otherwise, manually set the flask app env to the container\_runtime application:

export FLASK\_APP=container\_runtime.py

* Change directory to cops\_platform/container\_runtime

flask run

The application will print out whether it is running in SELinux enforced mode when it starts. By default it will be running on port 5000. If you want to run this application with SELinux enforced, set the ENFORCED env to any value. If not, unset the ENFORCED env to delete it.

* Ctrl-C will stop the running Container Runtime if running it in the foreground.
* If you want to run Container Runtime in the background, run with

flask run &

* Stop it with

kill -SIGINT <procid>

To manually test the REST API endpoints for Container Runtime and Course Manager, please consult the ***Testing APIs with Curl Commands*** section of this Developer Guide. Be sure to contact the url that is sent back from the service request for Container Runtime to test the Course Manager application running within that specific Docker Container.

If you are encountering any errors with running Container Runtime, please consult the ***Confirming Successful Container Runtime Installation*** section of the Installation Guide. This contains common troubleshooting information for issues that could potentially occur Also, always make sure the following preconditions and steps are true, depending on what SELinux mode you are in, that is detailed at the beginning of this guide.

## Installing and Updating Python Dependencies

|  |  |
| --- | --- |
| Primary Author | Jonathan Balliet |
| Editor | Jeen Shaji |

The requirements.txt file, located in the top level directory (*2020SpringTeam32*), contains all the dependencies that are needed for running the Python applications, along with their automated testing, for this project.

As development for the project continues, you will want to update the requirements.txt file with any new libraries you have installed with the pip command.

This can be easily be done by using the following command:

pip freeze > requirements.txt

The *freeze* command obtains the current list of all installed python modules. This command then pipes this output to the requirements.txt file. If you are running in a virtual environment, as is recommended and detailed both in our Installation and Development guide, this command will update the requirements.txt file with all Python modules you have installed in your virtual environment.

Now anyone on the team, or working on a different development machine, can update their own dependencies by using the following command:

pip install --user --requirement requirements.txt

This installs and/or updates all the libraries for the Python environment you are running in.

## 

## 

## Navigating COps Project Structure

|  |  |
| --- | --- |
| Primary Author | Jonathan Balliet |
| Secondary Authors | Daniel Mills, Spencer Yoder |
| Editors | Caleb Boswell |

### Top-Level

The Top-Level directory (2020SpringTeam32) contains the following contents:

* env\_vars.sh
  + Script to start the Python virtual environment and to set all the environment variables (envs) needed to connect to the PostgreSQL database running on the host machine. These envs are also required in order to successfully run both the Course Manager and Container Runtime Flask application, and for their respective automated tests.
* requirements.txt
  + Contains all of the Python dependencies needed in order to run the Course Manager and Container Runtime Flask applications.
* README.md
  + The README for our project contains information for how to run our application, generate coverage reports, and best practices we were following as a team with our development in Python and the Pycharm IDE. There are also linked resources for guides on working with Flask and SQLAlchemy.
* .gitignore
  + All the files we are ignoring when committing and pushing with git to our remote git repository on GitHub.
* cops\_platform
  + This directory is home to the four main sub-directories of this application: container\_runtime, services. policies, and front-end.

The services directory (2020SpringTeam32/cops\_platform/services) contains the following contents:

* course\_manager directory
  + For now, Course Manager is the only service we support for the University related system we designed our system around. In the future, other related services could be added to this application. A new service would have it’s root directory folder added here and follow similar conventions to the directory structure of course\_manager that is detailed in the **Course Manager** sub-section.

### Container Runtime

This contains all of the contents for the Container Runtime application, it’s automated tests, and bash scripts that our application calls in order to label our containers with SELinux. It is located in the directory (2020SpringTeam32/cops\_platform/container\_runtime). The directory container\_runtime contains the following contents:

* \_\_init\_\_.py
  + This is necessary for Python in order to recognize this directory as a module it can import. For this reason, it will be contained in every directory in this project. Most of \_\_init\_\_.py files will be completely empty and contain no code. However, there are few exceptions to this, and this will be clearly detailed in this guide when that is the case.
* clearlabel
  + Similar to iplabel. A shell script which takes an IP address as a parameter and removes the label associated with it using netlabelctl.
* config.py
  + This class is responsible for configuring all of the static settings for the Container Runtime application when it launches. These are configured via the envs that are detailed in the***Running Container Runtime*** section of this guide.
* container\_runtime.py
  + This is the main runner of the Container Runtime application and what the FLASK\_APP env must be set to in order for this application to launch with the “flask run” command.
* curl\_shortcuts.sh
  + A shell script containing bash functions which serve as a backend interface for the COps system. See Integrating SELinux with COps: System Testing Via the Terminal.
* flask\_app.sh
  + Contains two environment variables: the first defines the Flask app, to set the Container Runtime application as the target for Flask, the second sets an environment variable which several of our Python programs (including Container Runtime) use to identify whether the system they are running on has SELinux installed.
* get\_port
  + A binary executable used by curl\_shortcuts.sh to get the port from the JSON response returned after a service request to the Container Runtime application.
* get\_port.c
  + The source code for get\_port.
* iplabel
  + A shell script which takes two arguments: an IP address and a user type. Uses netlabelctl to label the IP with the context associated with the given user type.
* system\_test\_output.txt
  + The output of system\_tests.sh piped to a text file. Represents correct system test results, though is not exactly equal to correct system test output because some output could not be piped to the file.
* system\_tests.sh
  + A shell script which goes through all of the API endpoints to CourseManager for each user type. Displays the JSON responses of these calls to the terminal.
* app directory
  + Contains all of the code files and sub-directories for the Container Runtime application.
* tests directory
  + Contains all of the test code files for running the Container Runtime automated tests.

The Container Runtime app directory is located at the following location:

(2020SpringTeam32/cops\_platform/container\_runtime/app) and contains the following contents:

* \_\_init\_\_.py
  + This contains all the code for the app class that Flask uses to run and configure any settings for this application before it launches it. Sticking to typical Flask conventions, this code is put inside the \_\_init\_\_.py class because this will be initialized before Flask even begins running itself. There are some important global fields contained within this class, such as the docker\_client, which is used to communicate with the Docker Service. The db global field is used to communicate with the PostgreSQL database running on the host machine.
* controllers directory
  + The directory containing all of the controller classes that handle the REST API endpoints for this application.
* utils directory
  + The directory containing all of the utility classes and functions for this application.
* extensions directory
  + Contains statically defined fields that must be declared inside of another class due to conflicts they cause with the unittest module.

The Container Runtime controller directory is located at the following location

(2020SpringTeam32/cops\_platform/container\_runtime/app/controller)and contains the following contents:

* service\_controller
  + This class contains all of the major functionality of this application. It handles the service request (as a POST request), authenticates the user, determines their authorization level, and creates a container to enforce the authorization level of the user in the form of a SELinux label. This Python file also contains the HealthCheckThread class. This inner class extends the Python thread class and is responsible for monitoring every container after it is launched.

The Container Runtime utils directory is located at the following location,

(2020SpringTeam32/cops\_platform/container\_runtime/app/controller)and contains the following contents:

* logging\_util
  + This is currently unimplemented as it was a stretch goal for this project. However, the intent of this application was to contain all static codes for each log message and handle all logging for this application via defined functions.
* service\_util
  + This contains the ServiceConfig and CourseManagerServiceConfig classes. ServiceConfig is an abstract class containing fields which a service needs to work with the Container Runtime. The CourseManagerServiceConfig extends this class and contains statically defined values for the fields.

The Container Runtime extension directory is located at the following location,

(2020SpringTeam32/cops\_platform/container\_runtime/app/extensions)and contains the following contents:

* \_\_init\_\_.py
  + This contains the api field for the Flask Restful library that will be used to link all of our controller classes to specific endpoints. This is defined in a separate Python file due to conflicts caused by unittest.

The Container Runtime tests directory is located at the following location

(2020SpringTeam32/cops\_platform/container\_runtime/tests)

and contains the following contents:

* .env
  + This contains environment variables that are needed when running the automated tests. Due to the test Flask client executing in its own environment; it’s necessary to define these in its own file and load these envs using the Python dot-env library.
* container\_runtime\_tests.py
  + This class contains all of the unit tests for the container runtime class.
* safety\_test\_util.py
  + This was copied from an online source as a potential solution for exception issues that were occuring when running the automated tests for Container Runtime. However, this does not seem to have much impact on the Container Runtime testing issues and can likely be removed.

### Course Manager

This directory includes all of the contents for the Course Manager application, it’s automated tests, and the db\_generate script that is used to create and label the PostgreSQL database.

It is located at the following location

(2020SpringTeam32/cops\_platform/service/course\_manager) and contains the following contents:

* Dockerfile
  + The Dockerfile that is used to build a Docker Image for the Course Manager application. This basically contains all of the dependencies and files needed to run the Course Manager application in a CentOS environment.
* boot.sh
  + The bash script that is run when the Docker Image for the Container Runtime is launched.
* config.py
  + This class is responsible for configuring all of the static settings for the Course Manager application when it launches. These are configured via the envs that are detailed in the***Running Course Manager*** section of this guide. It also supports two different SQL database connections: SQLite and PostgreSQL. These correspond to the FLASK\_CONFIG env.
* course\_manager.py
  + This is the main runner of the Course Manager application and what the FLASK\_APP env must be set to in order for this application to launch with the “flask run” command.
* requirements.txt
  + Contains all of the Python dependencies needed to run the Course Manager application inside the Docker Container.
* app directory
  + Contains all of the code files and sub-directories for the Container Runtime application.
* tests directory
  + Contains all of the test code files for running the Container Runtime automated tests.

The Course Manager app directory is located at the following location:

(2020SpringTeam32/cops\_platform/services/course\_manager/app)

and contains the following contents:

* \_\_init\_\_.py
  + This Python file contains all the code for the app class that Flask uses to run and configure any settings for this application before it launches it. Sticking to typical Flask conventions, this code is put inside the \_\_init\_\_.py class because this will be initialized before Flask even begins running itself.
* controllers directory
  + This directory contains all the controllers for Course Manager. By controllers, we are referring to the classes that handle the REST API endpoints for our application.
* extensions directory
  + This directory contains statically defined fields that must be declared inside of another Python file due to conflicts they cause with the unittest module.
* models directory
  + This directory contains all the models classes for Course Manager. These correspond to the database objects that make up our database and we will be interacting with in our application
* utils directory
  + This directory contains all utility classes and functions our application will be using.

The Course Manager controllers directory is located at the following location:

(2020SpringTeam32/cops\_platform/services/course\_manager/app/controller)and contains the following contents:

* course\_controller.py
  + Handles all of the REST API endpoints related to courses.
* health\_check\_controller.py
  + Handles the special health check endpoint that the Container Runtime uses to see if a user’s session is still active.
* user\_controller.py
  + Handles all of the REST API endpoints related to users.

The Course Manager extensions directory is located at the following location:

(2020SpringTeam32/cops\_platform/services/course\_manager/app/extensions)and contains the following contents:

* \_\_init\_\_.py
  + Contains statically defined fields that must be declared inside of another class due to conflicts they cause with the unittest module.

The Course Manager models directory is located at the following location:

(2020SpringTeam32/cops\_platform/services/course\_manager/app/models)and contains the following contents:

* coordinator.py
* course.py
* course\_student\_mapping.py
* instructor.py
* student.py

Each of these classes extend the SQLAlchemy Model class and correspond to a specific database table and its attributes.

The Course Manager utils directory is located at the following location:

(2020SpringTeam32/cops\_platform/services/course\_manager/app/utils)and contains the following contents:

* controller\_util.py
  + Includes common utility functions that are shared among all the controller classes.
* logging\_util.py
  + This is currently not implemented since this was a stretch goal. However, the plan was to include the static logging codes from the Use Cases in this class, along with a function to accomplish all the logging of this application.

The Course Manager tests directory is located at the following location:

(2020SpringTeam32/cops\_platform/services/course\_manager/tests)

and contains the following contents:

* controllers directory
  + The directory containing the unit tests for each controller class.
* db directory
  + The directory containing scripts used for creating the database and labeling data.
* models directory
  + The directory containing the unit tests for each model class.

The Course Manager tests/models directory is located at the following location:

(2020SpringTeam32/cops\_platform/services/course\_manager/tests/models)and contains the following contents:

* coordinator\_test.py
* course\_test.py
* instructor\_test.py
* mapping\_test.py
* student\_test.py

Each of these tests classes contain unit tests for testing that specific model class.

The Course Manager tests/controllers directory is located at the following location:

(2020SpringTeam32/cops\_platform/services/course\_manager/tests/controllers)and contains the following contents:

* course\_controller\_test.py
* health\_controller\_test.py
* User\_controller\_test.py

Each of these tests classes contain unit tests for testing the specific controller class, and the REST APIs that are contained within. The Course Manager tests/db directory is located at the following location:

(2020SpringTeam32/cops\_platform/services/course\_manager/tests/db)

and contains the following contents:

* generate\_db.py
  + A python script that is used for resetting the state of the database. It drops all tables, creates tables based on the defined model classes, and labels the tables and columns if used with SELinux enforced on a PostgreSQL database.

### SELinux Policies

2020SpringTeam32/cops\_platform/policies/course\_manager contains all the files related to the CourseManager SELinux policy module. Those files are described as follows:

* tmp
  + This directory is used in the compilation/build process of the policy. The files within are generated by this process.
* compile\_alias.sh
  + A shell script that, when loaded using ‘source compile\_alias.sh,’ adds two aliases for use in compiling and building the policy. ‘secompile’ which compiles course\_manager.te to course\_manager.pp and ‘sebuild’ which builds course\_manager.pp onto the system’s policy.
* course\_manager.fc
  + This file is generated during the compile process for the policy module. It’s a piece of source code which determines the default label for certain files in the system. It’s unused in our policy
* course\_manager.if
  + This file is generated during the compile process for the policy module. It’s a piece of source code which defines the interfaces (macros) used in course\_manager.te. It’s unused in our policy.
* course\_manager.pp
  + Contains the binary information which is built on to the system’s SELinux policy.
* course\_manager.te
  + Contains the relevant source code for CourseManager’s SELinux policy. Written using SELinux’s Kernel Policy Language.
* journal\_log.txt
  + The output of a journalctl call which shows denials logged by SEPostgres. This data has no bearing on the system, but is a good example of what such denials look like.

### Front-end

The front\_end directory (2020SprintTeam32/cops\_platform/front\_end) contains the following contents:

* html - This directory contains all of the HTML files that make up the front-end interface. The notable files and directories are listed below:
  + Index.html
    - The directory listing for the COps Platform which directs the user to the single sign-on page.
  + Sso.html
    - The single sign-on page for the COps platform and Course Manager. Here, registered users can enter their username and password (which is not used currently) to log in to the system.
  + home.html
    - The homepage for Course Manager. Here, the user is presented with a list of all the different actions they can take in the system. Each item in the list corresponds to one API call. Each link takes the user to an HTML page in which they can fill in any necessary information to make the API call.
  + js
    - The directory with all of the Javascript files for the front-end. Contains the necessary AngularJS files, as well as util.js. util.js has helper functions for making API calls that are used by the HTML files.
  + xss
    - Contains style.css, which is used by the HTML files for styling tables the correct way.
* mockups
  + The directory containing images of each Balsamiq mockup from the front-end design. This is helpful when developing HTML pages, since it allows you to open the mockup and HTML file in the same IDE and cross-reference the image while developing.

## 

## Creating Database Models

|  |  |
| --- | --- |
| Primary Author | Caleb Boswell |
| Editor | Jonathan balliet |

From the 2020SpringTeam32 root directory, navigate to services/course\_manager/app/models. The models are used so objects can store and access data in the database and to control what format the data must be in.

Let us look at an example of how these models store the data using the course model. In course.py we first add the from ...app import db statement to allow the model to interact with the database. Then we define the class using class Course(db.Model):, db.Model signifies the extension of the SQLAlchemy model class to use in this particular model.

Next we declare what data this model will have, in course the variables are id, name, days, start\_time, end\_time, instructor\_id, and course\_mapping. Once the variables are declared they must be defined using a format specific to use with the database. For simple variables, like name and day, this is done using db.Column() which creates a column in the database table to store data for that particular attribute. For variables that hold object relationships as their data, such as course\_mapping which maps student objects to course objects to signify enrollment, we use db.relationship() to signify this variable holds a one-to-many relationship value.

After using the appropriate parameter for each variable you then must define their attributes so that data can be stored correctly. Here is an example statement from the name variable, name = db.Column(db.String, nullable=False) , Since it stores a value it uses db.Column for the database. Inside the parenthesis we first declare what data type will be stored in the database, in this case it is a string, and pass it in using the appropriate command db.String. After which we declare any other attributes we need to define such as whether the value can be nullable or empty when the object is created or if the data input must be unique to that object when compared to other objects of the same type. For variables like id, it is necessary to add the attribute primary\_key=True to the variable to indicate that it is a value that is unique to the object and that it is used to parse through the database to find that object. For the name variable it does not need to be unique so that attribute is not set, also the variable requires the value not be null or empty so we add the parameter nullable=False inside the parenthesis, to indicate the value must be filled when creating the object or it will not be passed into the database. Most of the other variables are defined similarly except instructor\_id. Since ids are unique to objects and since instructor is a separate object from the course object it is necessary to insert db.ForeignKey(“instructor.id”). This indicates that the value is listed in a separate column than is currently being used and prevents the use of values in this variable that are not from that column.

For the course\_mapping object, as it is declaring a relationship instead of a column the attributes must also change accordingly. Here in the variable statement it shows the necessary input for the variable course\_mappings = db.relationship('CourseStudentMapping', backref='mapping', lazy=True), which differs greatly from db.Column. The first part ‘CourseStudentMapping’ declares what class will be used for this relationship, in this case it is the CourseStudentMapping class. The method then uses the backref attribute to determine what variable within that class will be stored, and the lazy attribute determines when the database will load this data. In the case of lazy=True, it means the data will only be loaded as it is necessary.

Following these rules the rest of the variables can then be declared and thus create the Course model for the database.

## 

## Creating REST API calls

|  |  |
| --- | --- |
| Primary Author | Jeen Shaji |
| Editor | Jonathan Balliet |

From the 2020SpringTeam32 root directory, navigate to services/course\_manager/app/controllers . Let us walk through an example where we created an api request for viewing the course schedule. In course\_controller.py, we create a new class called CourseScheduleView(Resource) and define a request like below:

def get(self):

Inside this method, first make sure that a user is logged in by using the below command before proceeding:

abort\_if\_user\_not\_logged\_in()

We can get the current user by checking credentials in the database by calling the below method:

user = check\_user\_credentials(username, role)

This method will be found in app/utils/controller\_util.py. We will now search the database to find relevant schedules for the current user and append all course fields to a list if there are more than one:

schedule.append(course\_fields)

If there are no exceptions, we return this schedule with successful return code:

return schedule, 200

Now, to connect this new api to a correct url, we will navigate to app/\_init\_.py and inside def \_register\_api(app), we will add a new resource as shown below:

api.add\_resource(CourseScheduleView, "/api/schedule")

This should successfully create the new REST api. If not, make sure all your model classes are created and have methods to support the controller extension.

## 

## Test APIs with curl commands

|  |  |
| --- | --- |
| Primary Author | Jonathan Balliet |
| Editor | Jeen Shaji |

Make sure you are in the course\_manager directory and have set the environmental variable FLASK\_APP=course\_manager. Start up the Course Manager application using the flask run command.

It should be running on your localhost at port 5000. This is the following url: 127.0.0.1:5000 . If running through container runtime, it’s necessary to change the port from 5000 to the port returned from the API call.

* Curl command to make a GET request to a specific endpoint:

curl <http://127.0.0.1:5000>

This will contact the base endpoint (http://127.0.0.1:5000) and send back an authentication error if you are currently not logged in. If you are logged in, a message saying the user you are logged in as, will be sent back.

* Curl command to login as a user and and create a session in Course Manager:

curl -d "username=student" --cookie-jar ./session <http://127.0.0.1:5000/api/login>

The --cookie-jar flag saves the session cookie inside of a designated file ( the ‘session’ file in the example above). The -d argument is used to add data in this request. By default, this signifies a POST request. It places data in the body of the request in the form “key=value”. In this case we are passing in the “student” value for the “username” key.

* The ROLE environment variable must be set to the corresponding role of the user you want to login in as. By default, it is set to ‘student’. For example, if you want to change the ROLE value to ‘coordinator’ use the following command:

export ROLE=coordinator

The user you are attempting to login must already exist in the corresponding table (based on their role) for the database you are trying to contact.

* Curl command to view the account details for a logged in user:

curl --cookie ./session --cookie-jar ./session <http://127.0.0.1:5000/api/user>

This sends a GET request to this API endpoint and will send back a JSON response of this user’s account information. The --cookie flag sends the session cookie in the request from the designated file it is stored (‘session’ in our case). This must be sent with every REST API request or else you will receive an authentication error. The --cookie-jar flag is used to modify the updated session cookie in the designated file (‘session’) from the REST response. In order to track a user’s inactivity the session cookie is updated every time the user makes a REST request, therefore, the stored session cookie must be updated each time.

* Curl command to add a new user to the database:

curl --cookie ./session -d "username=student\_tester&name=Student Test&role=student" --cookie-jar ./session <http://127.0.0.1:5000/api/user>

The -d argument takes in three key/value pairs in the body of the POST request. These consist of the “username”, “name”, and “role” of this user. The ‘&’ symbol is used to separate the key/value pairs. Similar to the last command, you must both send the session cookie and then update this session based on the cookie sent back. You should receive a message in the response that the user was successfully added.

* Curl command to delete an existing user from the database:

curl -X DELETE --cookie ./session -d "username=student\_tester&role=student" --cookie-jar ./session <http://127.0.0.1:5000/api/user>

The -X <Method> argument allows you to make specific method requests using curl. In this case we are making a DELETE request. In this request, we must pass in the username and role of the user in the body of the request using the -d argument. You should receive a message back that the user was successfully deleted.

* Curl command to logout as a logged in user:

curl --cookie ./session <http://127.0.0.1:5000/api/logout>

This will logout the user and end their session. You will receive authentication errors if you try to make a REST request to Course Manager endpoints until you log back in as a user using the ‘/api/login’ endpoint.

* Curl command to add a new course:

curl --cookie ./session -d "name=CSC326&days=MW&start\_time=11:00&end\_time=12:45&instructor\_username=instructor" --cookie-jar ./session <http://127.0.0.1:8000/api/course>

A successful request requires the following fields passed into its body:

* name - the name of this course. Must be a unique course name
* days - the days of the week this course takes place.
* start\_time - the time this course starts. Must be in the format “HH:MM”
* end\_time - the time this course ends. Must be in the format “HH:MM”
* instrutor\_name - the username of the instructor of this course. Must be the username of an existing Instructor in the system.
* Curl command to delete an existing course:

curl -X DELETE --cookie ./session -d "course\_name=CSC326" --cookie-jar ./session <http://127.0.0.1:5000/api/course>

A successful request requires the following fields passed into its body:

* course\_name - the name of this course. Must be the name of a course that exists in this system.
* Curl command to enroll a student in a course:

curl --cookie ./session -d "course\_name=CSC316&username=student" --cookie-jar ./session <http://127.0.0.1:5000/api/mapping>

A successful request requires the following fields passed into its body:

* course\_name - the name of this course. Must be the name of a course that exists in this system.
* username - the username of the student to enroll. Must be the name of a student that exists in this system.
* Curl command to modify the grade in a course for a student:

curl -X PUT --cookie ./session -d "course\_name=CSC316&username=student&grade=3.5" --cookie-jar ./session <http://127.0.0.1:8000/api/mapping/grade>

A successful request requires the following fields passed into its body:

* course\_name - the name of this course. Must be the name of a course that exists in this system.
* username - the username of the student to enroll. Must be the name of a student that exists in this system.
* grade - the grade to be modified. Must be in the format of integer or float value
* Curl command to view the schedule of the logged in user:

curl --cookie ./session --cookie-jar ./session <http://127.0.0.1:5000/api/schedule>

* Testing Container Runtime’s REST APIs using Curl commands:

curl -d "username=student&service=course\_manager" --cookie-jar ./session <http://127.0.0.1:5000/service_request>

A successful request requires the following fields passed into its body:

* username - the username of the user that is logging to be authenticated. This user must exist inside one of the tables (student, coordinator, or instructor) in the PostgreSQL database.
* service - the name of the service to be run. Currently “course\_manager” is the only accepted service name value.

If a successful request is made to the Container Runtime, a response will be sent back containing the URL where Course Manager is currently running. The session cookie that is sent in the response is needed to maintain authentication for the already logged in user when contacting Course Manager. You can then test out Course Manager running inside the Docker Container using the same curl commands detailed above in this section.

## 

## Running Automated Tests

|  |  |
| --- | --- |
| Primary Author | Jonathan Balliet |
| Editor | Jeen Shaji |

We have three suites of unit tests: one for Course Manager, one for Container Runtime, and finally one for System Testing our SELinux policies with the Database and our Applications.

### Running Unit Tests for Course Manager

Unit tests for CourseManager are located within the 2020SpringTeam32/cops\_platform/services/course\_manager/tests directory. Tests relating to the controller classes, which handle all the REST API calls, are located within the /tests/controller directory. Tests relating to testing the model classes, which correspond to our database objects, are located within the */tests/model* directory. Run a unit test by using the following command:

python -m unittest <path/to/python/unit/test/file>

For example, running the following command results in the coordintator\_test.py unit tests being run:

python -m unittest coordinator\_test.py

Output will then be shown on the terminal for the results of these test cases.

### Running Unit Tests for Container Runtime

Unit tests for Container Runtime are located within the

2020SpringTeam32/cops\_platform/container\_runtime/tests

directory. Currently, these tests are contained within one file, titled

container\_runtime\_tests.py. The test cases for Container Run are run using the following command:

python -m unittest path/to/container\_runtime\_tests.py

It is expected that an exception will be thrown when the unit tests call the *tear down* function. Sending a signal to the Container Runtime is necessary in order to clean up all containers so that no port conflicts are caused when starting up the Container Runtime again. It also tears down all running threads as well. However, even with this exception being thrown, the test cases should still report an OK status, that all test cases have successfully passed. If you run into any issues with the testing, ensure that a user exists in the Student table of the PostgreSQL database with the username “student”. The unit tests assume this precondition is true in order for the test cases to successfully pass.

### Running System Tests for SELinux Policies

The System Tests that were developed to make sure our SELinux policy is working correctly in our end-to-end system is available at the following location:

cops\_platform/container\_runtime/system\_tests.sh

This suite of tests can be run using the following command:

$ sh system\_tests.sh

The output should (mostly) match the text contained with the following file:

cops\_platform/container\_runtime/system\_test\_output.txt

They will not be entirely identical, since not all output could be piped to system\_test\_output.txt.

## 

## Running Coverage Reports

|  |  |
| --- | --- |
| Primary Author | Jonathan Balliet |
| Editor | Jeen Shaji |

The nose Python library will be used to locate and run all unit tests that reside in our tests directory. This library uses the coverage.py module, with an additional argument, to run a coverage report for all detected unit tests. Similar to the above Automated Tests section, there are separate coverage reports for the Container Runtime and Course Manager, since these are two different applications.

### Generating Coverage Report for CourseManager:

Run the following command, from the root directory *"*2020SpringTeam32*"*, in a terminal to run all the unit tests and generate a coverage report:

nosetests -w cops\_platform/services/course\_manager/tests/ --with-coverage --cover-package=cops\_platform

After running this command, you will be shown output displaying the statement coverage amount for each Python class contained within the course\_manager directory.

### Potential Issues:

If you receive a complaint that Python cannot import a specific module, make sure you set the PYTHONPATH variable as described within the prior ***Running Course Manager*** section of this guide.

### Generating Coverage Report for ContainerRuntime:

Run the following command, from the root directory *"*2020SpringTeam32*"*, in a terminal to run all the unit tests and generate a coverage report for the Container Runtime:

nosetests -w cops\_platform/container\_runtime/tests --with-coverage --cover-package=cops\_platform

***NOTE:*** You will likely receive an exception when running the above command. This is expected behaviour. A coverage report will still be generated based on the tests, but the code within the HealthCheckThreadclass is currently being ignored with our implementation. We did not have time to fix this before our due date for this project.

### Potential Issues:

Similar to the above sub-section, you must have the root directory of the project available in the PYTHONPATH.

## 

## Accessing remote server from Firefox

|  |  |
| --- | --- |
| Primary Author | Daniel Mills |
| Editor | Jeen Shaji |

This part of the document describes how to access a remote server from the Firefox browser. The application of this for our project is loading the HTML files on a remote CentOS 7 computer so that we can use the front-end interface. These instructions are based on this page: <https://linuxize.com/post/how-to-setup-ssh-socks-tunnel-for-private-browsing/>

1. Open a terminal, and run the following command:

ssh -N -D 9090 <remote-address>

This will open a SOCKS tunnel to the remote computer on port 9090

1. Leave that terminal running, and open another one. This time, ssh into the same computer as you normally would:

ssh <remote-address>

1. Navigate to the directory containing the HTML pages you want to access, and run:

python -m http.server 9000

This will start a server at address 0.0.0.0 and port 9000, where the root is your current directory (the one with all the HTML pages)

1. Open Firefox, navigate to Options, scroll to Network Settings, click Settings.
2. Select the radio button that says “Manual proxy configuration”
3. For SOCKS Host, enter 127.0.0.1, and enter 9090 for its port.
4. Make sure SOCKS v5 is selected
5. Click OK
6. Type [http://0.0.0.0:9000/](http://0.0.0.0:8000/) into the address bar, and you should see the list of available pages

### Notes about back-end support in Flask

If you try to connect to the Flask app as is, and the app uses authentication with cookies, you will get CORS errors. To fix this, you will need to support credentials in both the back-end and front-end

* In the back-end, install the flask\_cors Python library:
  + pip install flask\_cors
* Next, go to where you initialize the Flask app, import CORS from flask\_cors, and add the following line of code underneath where you initialize the app object:
  + CORS(app, supports\_credentials=True)
* In the front-end, you need to set the defaults.withCredentials field of the $http object. Before you make the http request, add the following line of code:
  + $http.defaults.withCredentials = true;

### Resources:

* A video showing Firefox connecting to a simple HTML application:

[https://drive.google.com/open?id=1z0Lo0Is0bamrLSiPMh59k7uo9BIsstS](https://drive.google.com/open?id=1z0Lo0Is0bamrLSiPMh59k7uo9BIsstS3)

1. Forgot to use -i here [↑](#footnote-ref-0)