# **Connected Operations (COps) Platform**

# Final Project Report

# 

# **Skyward Federal**

# **CSC 492: Team 32**

# Jonathan Balliet

# Caleb Boswell

# Daniel Mills

# Jeen Shaji

Spencer Yoder

**North Carolina State University**

**Department of Computer Science**

**May 1, 2020**

## Executive Summary

|  |  |
| --- | --- |
| Primary Author | Daniel Mills |
| Editor | Spencer Yoder |

Skyward Federal is a Cyber-Logistics startup company that specializes in areas such as cloud storage, infrastructure, app development, and data security. Based in Washington DC, they primarily serve government clients, but their services are relevant to any organization that prioritizes data security including banks, healthcare organizations, and insurance companies. With only around ten employees, Skyward is a very small team that has worked closely with us to help solve the problem of modern multi-level security.

Many companies and organizations need to store data that should only be accessible to certain users, but not others. Some examples include banks, government institutions, and healthcare organizations. The way these organizations typically handle controlling access to data is by keeping data with different levels of access in separate data stores. Data with one security level is kept in a different physical server than data with a different security level. While this is a good solution in terms of security, it is inefficient and leads to a system with multiple potential points of failure.

Our solution to the problem, posed by Skyward Federal, is to keep everything in a centralized data store, and give the data labels that will determine which users are allowed to access them. This solution is named the Connected Operations (COps) Platform. By using a PostgreSQL database in tandem with Security Enhanced Linux (SELinux), we have implemented a simple, integratable system that organizations can use to improve their data storage. An important component of the solution is the use of Docker containers to run the services which interact with the database. This virtualization improves security by preventing the user from directly interacting with the database, and also means that the system does not rely on the availability of one single computer.

Our final project has three components: application, data storage, and container runtime. The application component serves as an example of an application that would benefit from the COps Platform. The application is called Course Manager, as it emulates a system a university might use to manage data about students, instructors, and courses. The data storage component consists of an SEPostgreSQL database, which stores data from the application with SELinux labels. This method of data storage ensures that requests to access the data are allowed or denied at the database level, which greatly improves security. The container runtime component consists of a Docker container that acts as a gateway between the user and the application.

We tested each component in a variety of ways, including Bash scripts with curl commands, unit tests with the unittest Python library, and black box system tests that we perform manually on a front-end interface we developed. The final project has all components implemented, and all tests passing with 87% coverage for the Container Runtime, and 96% coverage for Course Manager.

## Project Description

### Sponsor Background

|  |  |
| --- | --- |
| Primary Author | Jeen Shaji |
| Secondary Author | Spencer Yoder |

Skyward Federal is a software start-up specializing in custom security software solutions for government clients. They have about ten employees. They provide technical solutions that can incorporate security and modern software development practices to solve some of the Department of Defense’s biggest problems. The company was founded in late 2019 and is based in Colorado Springs, Boston, and Washington, D.C.

Our contacts at Skyward have repeatedly communicated to us that what we are able to create here will greatly inform how they approach this project when implementing it for their customers. On their end, this senior design project is a part of the research phase for the Connected Operations (COps) Platform.

**Erin Kotlyn**, Solutions Architect

Contact: [erin@skywardfederal.com](mailto:erin@skywardfederal.com)

**Danny Caudill**, Full Stack Developer

Contact: [danny@skywardfederal.com](mailto:danny@skywardfederal.com)

**Matthew Peters**, CEO Skyward Federal

Contact: [matt@skywardfederal.com](mailto:matt@skywardfederal.com)

### Problem Statement

|  |  |
| --- | --- |
| Primary Author | Jeen Shaji |
| Secondary Author | Spencer Yoder |

Many government and private firms have the need to separate their data by security level. These organizations traditionally store data of different security levels in disparate databases instead of a central location to preserve security. This solution has the problems of being difficult to scale, requiring a complex architecture to navigate, having multiple points of failure, and incurring a lot of time overhead when searching through data from multiple sources. Inefficient security systems can cause information to fall into the wrong hands. For example, a student does not need access to personal information such as grades of their classmates. If one student can obtain access to the personal details of other students, they are technically invading the privacy of their classmates.

### Project Goals & Benefits

|  |  |
| --- | --- |
| Primary Author | Jeen Shaji |
| Secondary Author | Spencer Yoder |

Goals:

* Implement an API endpoint for a web application which can authorize users to access and manipulate the database. Each request will be accepted or denied based on SELinux enforcement.
* Enforce data security at the operating system level with Security Enhanced Linux (SELinux). Check every system request’s security level before giving them access to the data requested.

Benefits:

* A centralized system reduces overhead and system complexity, which stem from having to interface with a set of distributed data stores.
* A virtualized system is robust and recovers in the event of a failure.
* The COps platform will be just as secure as the prior solution.

### Development Methodology

|  |  |
| --- | --- |
| Primary Author | Spencer Yoder |
| Secondary Author | Jonathan Balliet |
| Editor | Jeen Shaji |

We intend to follow the Agile methodology, dividing up our project into iterations and tracking progress with a Kanban board through Github. At the end of each iteration, we intend to have a full end-to-end system that is able to take API calls all the way to the database and back. Since we are working with the Skyward team, who uses an Agile methodology, our iterations will last 2 weeks on average.

We have been communicating with Skyward Federal using slack and we intend to use the Kanban board to keep track of our progress as well as what each member is currently working on. This not only helps us track the pace of development, but also facilitates coordination between group members: if individual team members can see what everyone else is doing, what has been done and what has not been done, they can make decisions about how to react appropriately. A person can assign a task to themselves, create a related issue on github to keep track of progress.

### Challenges

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

* Lack of knowledge
  + None of the members of our group had done this kind of development before the onset of this project. Not only that, but we were working on combining technologies which had likely not been combined in this way before. So we had little intuition for how to proceed, a lot of goals for our project, and little in the way of digestible information for our technologies.
  + We tackled this challenge as one would expect: we spent a lot of time reading, a lot of time learning, and a lot of time making mistakes. This is easily the most pervasive challenge we faced in our efforts because it affected our ability to understand our progress holistically and plan accordingly. We had to get used to revising our plans on the fly and being adaptable. However, this was another learning experience which we were able to surpass.
* Turning abstract requirements into a concrete plan
  + At the beginning of the project, we set our sights on implementing a system close to the one described in Skyward’s project description. However, we hardly ever formally defined things which were not defined in the project description: our services remained ambiguous, our database could contain arbitrary data, our policy rules and labels could be anything, so long as the system as a whole functioned.
  + This, as might be expected, made it difficult for us to focus on our goals. We moved past this challenge by defining the CourseManager system after our planning meeting with Ms. Heil. This allowed us to have a concrete definition for one of our services, our database, and our security policy. We were no longer aiming for a broad set of goals, we were designing this system around a core component.
* Doing teamwork in a research environment
  + Much of our implementation involved a process of jumping head-first into an unfamiliar technology and failing until enough was learned to use the technology. This is a difficult process for a team of people since, if two people are trying two sets of things concurrently, the first functional thing someone finds will be the way the technology is used and the second person’s work will be cast aside and they will be unable to help with the chosen methodology.
  + This exact scenario happened to us a couple times. First with SEPostgreSQL, and then a few more times with integrating things with SELinux. With SEPostgreSQL, installation and usage were a large process and there was useful knowledge in both sets of work. With the integration of SELinux, however, the integration consisted of several small steps, each of which took a lot of research to discover, and a lot of work was done in vain. We tackled this challenge by taking a more modular approach to our division of labor. One person was assigned a small component and had full responsibility for that component. This approach worked well in this environment and folks were able to ask for help when they needed it, but still take charge of progress in their domain.

## 

## Requirements

|  |  |
| --- | --- |
| Primary Author | Jeen Shaji |
| Editor | Spencer Yoder |

### Overview

This project involves the development of an application through which institutions can isolate data and share only relevant information with their work force or customers. Although data is largely available, it is essential to keep it secure. On authentication, users can only access the data they have permission to access, which depends on their role in the company and the device they are using. Each container knows the user’s role before it connects to our core application, Course Manager. We use SELinux Security Policy Module to define access rights of each user with our database.

The final product is the backend infrastructure for making API calls to access data. Authorized users can access relevant information and every request will be logged. Security and privacy of such a system are of paramount importance. Company rules protect information and allow an admin to dictate who can access particular information.

The information and instructions we received from Skyward Federal was for an abstract system. Our sponsors leaned towards a proof of concept instead of an actual product. Hence, we used their functional requirements to create a more specific mock system called Course Manager.

Here is a list of functional requirements, non-functional requirements and constraints for the overall COps system. This is followed by use cases developed for our mock Course Manager system.

### Functional Requirements

Whole System:

FR1. The system shall be interfaced via an API endpoint.

FR2. The system shall authenticate and authorize users at login.

FR3. After logging in, the system shall grant users access to a RESTful service.

FR4. The system shall accept or deny requests made to the service based on the user’s type.

Data Storage:

FR5. The database shall label each database object to facilitate access control.

FR6. Access rights for data shall be granted to users based on the included SELinux security policy.

FR7. The database shall accept multiple incoming connections, each with access rights depending on the connected user.

FR8. The database shall be populated and labeled via an executable script.

Container Runtime:

FR9. The container shall provide computing resources for the CourseManager service.

FR10. This component shall start up a container with the enforced security context of the user for the requested service.

FR11. The service running inside the container must gracefully handle ‘Access Denied’ errors when attempting to retrieve data from the Data Storage component.

FR12. The containers shall be able to be dynamically activated and deactivated to serve data to users with the proper security context.

FR13. This component shall shut down a container when the user disconnects from the service.

### Non-functional Requirements

NFR1. The system shall prioritize security over performance

NFR2. The system shall ensure container startup time is less than 5 seconds.

NFR3. The Docker images shall include all executables, libraries, and configuration data so that the application can run in an offline environment.

### Constraints

C1. The final configuration must run on CentOS 7 with SELinux enabled.

C2. SELinux shall be used as our security module with the Data Storage component.

C3. PostgreSQL shall be used as the database for our Data Storage component because it can integrate with SELinux

C4. Each RESTful service shall be given login credentials to connect to the SEPostgreSQL database within the Data Storage component.

### CourseManager

There are currently three types of users in the Course Manager system. The role grants access and their security context determines their viewing and editing capabilities. Unless mentioned otherwise, a user can be any of the three types.

* **Coordinator**: A coordinator has control over all instructors, students and courses
* **Instructor**: An instructor has control over grades of their students
* **Student:** A student can only view their courses and grades.

Figure 5 on page 26 represents our Security Labels (SL) associated with each user type. Coordinator corresponds to SL s2:c0.c3, Instructor corresponds to SL s1:c0.c1 and Student corresponds to SL s0:c1.c2.

#### UC1: Logging In

1.1 Preconditions

The account with the pertinent user type (student, instructor, coordinator) has been set up in the system.

1.2 Main flow

The user inputs their username [S1] and is greeted with a list of possible actions [S2][S3][S4].

1.3 Subflows

* [S1] The system is able to display the proper menus for the user type associated with the given username [E1]. We will not enforce passwords in this system.
* [S2] A coordinator has the ability to view their account details, modify courses, and modify other users.
* [S3] An instructor has the ability to view their account details and modify grades.
* [S4] A student has the ability to view their account details, view their course schedule, and view their grades.

1.4 Alternate flows

* [E1] If no account has been set up in the system, the user is not allowed to log in.

1.5 Logging

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Transaction code | Description | Logged in username | Secondary user | Transaction type |
| 100 | Failed login | IP address | N/A | Other |
| 101 | Successful login | user | N/A | Other |
| 102 | Logged out | user | N/A | Other |

1.6 Data Format

|  |  |
| --- | --- |
| Field | Format |
| User name | Between 6 and 20 alpha characters and symbols - or \_ |

#### UC2: Viewing Account Details

2.1 Preconditions

The active user has an account set up with the system and has logged in (UC1)

2.2 Main flow

No matter what user type, users can see their username, id number, and full name [S1]. They can see a menu of options which shows all possible API requests but can only access a few. Instructors can see their course schedule and course rosters [S2]. Students can see their course schedule and GPA [S3]. Coordinator can see information about all students and instructors.

2.3 Subflows

* [S1] No user can modify account details.
* [S2] Instructors cannot add their own courses.
* [S3] Course grades are set for students by instructors and can be empty.

2.5 Logging

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Transaction code | Description | Logged in username | Secondary user | Transaction type |
| 200 | View options | user | N/A | View |

#### UC3: User Modification

3.1 Preconditions

A coordinator has an account in the system and has logged in (UC1)

3.2 Main flow

Coordinators can add users [S1], and remove users [S2].

3.3 Subflows

* [S1] Coordinators add users by specifying a user type, username [E1], and full name. Id numbers will be assigned by the system.
* [S2] Coordinators can specify a username to remove from a list of existing users [E2].

3.4 Alternate flows

* [E1] Coordinators cannot add users with duplicate usernames.
* [E2] Coordinators cannot delete non existential users.

3.5 Logging

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Transaction code | Description | Logged in username | Secondary user | Transaction type |
| 300 | Successful add user | coordinator | user | Add |
| 301 | Successful delete user | coordinator | user | Delete |
| 302 | Failed add user | coordinator | user | Add |

3.6 Data Format

|  |  |
| --- | --- |
| Field | Format |
| userType | Student/ Instructor/ Coordinator |
| username | Between 6 and 20 alpha characters and symbols - or \_ |
| name | Between 6 and 40 alpha characters and symbols - or \_ |

#### UC4: Course Modification

4.1 Preconditions

A coordinator has an account in the system and has logged in (UC1) and an instructor and a student exist in the system (UC3).

4.2 Main flow

Coordinators can add courses [S1], and remove courses [S2]. Coordinators can enroll students in courses [S3].

4.3 Subflows

* [S1] Coordinators add courses by specifying a course name, day, time range, and instructor [E1].
* [S2] Coordinators can enter a valid course name to delete it [E2].
* [S3] Coordinators can add a student to a course after it has been created by entering the student’s username [E3] [E4].

4.4 Alternate flows

* [E1] A course addition is denied for a nonexistent instructor.
* [E2] A course deletion is denied for a nonexistent course
* [E3] A student addition is denied for a nonexistent student.
* [E4] A student cannot be enrolled more than once in the same course.

4.5 Logging

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Transaction code | Description | Logged in username | Secondary user | Transaction type |
| 400 | Successful add course | coordinator | N/A | Add |
| 401 | Successful delete course | coordinator | N/A | Delete |
| 402 | Successful add student | coordinator | student | Add |
| 403 | Failed add student | coordinator | N/A | Other |

4.6 Data Format

|  |  |
| --- | --- |
| Field | Format |
| courseName | Between 6 and 20 alpha characters |
| day | Between 1 and 6 day initials (M,T,W,Th,F,S) |
| time | HH:MM |
| instructor | Between 6 and 20 alpha characters and symbols - or \_ |
| student | Between 6 and 20 alpha characters and symbols - or \_ |

#### UC5: Grade Modification

5.1 Preconditions

An instructor has an account set up with the system, has been added to a course with an enrolled student, and has logged in (UC1) (UC4)

5.2 Main flow

Instructors can select a student from any course and modify their grade [S1].

5.3 Subflows

* [S1] Grades are stored for each student-course pair as a floating-point number between 0 and 4 [E1].

5.4 Alternate flows

* [E1] It could be empty if none of the courses have been graded yet.

5.5 Logging

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Transaction code | Description | Logged in username | Secondary user | Transaction type |
| 500 | Successful add/ change grade | coordinator | student | Add |
| 501 | Successful delete student | coordinator | student | Delete |

5.6 Data Format

|  |  |
| --- | --- |
| Field | Format |
| grade | Float value between 0 and 4 |
| username | Between 6 and 20 alpha characters and symbols - or \_ |

#### UC6: Viewing Course Schedule

6.1 Preconditions

A student or instructor has an account with the system, has logged in, and has been added to a course (UC1) (UC3) (UC4).

6.2 Main flow

The user can see their schedule for the week [S1]. Students see the grades associated with their class [S2] and instructors have the option to navigate to the roster for each class [S3].

6.3 Subflows

* [S1] The user can see their schedule for the upcoming week.
* [S2] If the user is a student, their grade is displayed next to the course [E1].
* [S3] If the user is an instructor, they can just select their course and view the course roster.

6.4 Alternate flows

* [E1] It could be empty if none of the courses have been graded yet.

6.5 Logging

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Transaction code | Description | Logged in username | Secondary user | Transaction type |
| 600 | Successful see schedule | coordinator/ student | N/A | View |
| 601 | Successful see roster | coordinator | N/A | View |
| 602 | Successful see grade | student | N/A | View |

## Resources Needed

|  |  |
| --- | --- |
| Primary Author | Jonathan Balliet |
| Secondary Author | Spencer Yoder |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Resource | Explanation | Version | Licensing  Information | Status |
| CentOs | CentOs, also known as Community Enterprise Operating System, is a free, community driven Linux distribution. This specific Linux operating system was chosen due to the ability to easily integrate with the SELinux module. Both the Container Runtime and Data Storage components will be stored on a single CentOs image. | 7 | Open Source | ✅ |
| PostgreSQL | PostgreSQL is an open-source Object Relational Database Management System. Our sponsors recommended PostgreSQL in particular because of a plugin which allows it to interface with SELinux. | 13 (development) | Open Source | ✅ |
| SELinux | SELinux is a Linux kernel security module that allows admin controls for Multi-level Security (MLS). We are using this module to integrate with both our Data Storage component and the services running in the Container Runtime. This ensures that a user can only access and modify information they are authorized to from the SELinux integrated database. | 2.5 | Open Source | ✅ |
| Docker CE | Docker Community Edition (CE) is a containerization platform. This allows us to Dockerize our Course Manager mock service. This way we can run our application in a CentOs7 image with enforced SELinux labels in order to maintain the authorization security context of the user with the Data Storage component. | 19.03.05 | Open Source | ✅ |
| AWS EC2 | EC2 is a virtual machine management system. We’ve been developing on EC2 virtual machines to ensure a consistent development environment and have the ability to work with specific technologies like SELinux. | N/A | Proprietary | ✅ |
| Python | This is our main programming language we will be using for development for the COps Platform system. This specific version of Python was chosen due to being the default Python 3 version installed on CentOs7. The unittest module that is included automatically in this Python version, will be used for unit testing our classes. | 3.6.8 | Open Source | ✅ |
| Flask | Flask is a micro web framework, using the Python language, that allows us to create a running backend server. Both our Container Runtime and Course Manager applications will be run using the Flask framework. | 1.1.1 | Open  Source | ✅ |
| SQLAlchemy | SQLAlchemy is an Object-Relational-Mapping (ORM) library written using the Python language. ORM allows us to easily communicate to the database using our main programming language, Python, instead of having to do this through SQL statements. | 1.3.13 | Open  Source | ✅ |
| Flask-RESTful | Flask-RESTful is a Python library that allows us to create simple REST APIs which can easily be contained within separate Python classes to enable Object Oriented (OO) design. | 0.3.8 | Open  Source | ✅ |
| Coverage.py | A Python library used for measuring your code coverage. | 5.0.3 | Open Source | ✅ |
| Nose | A Python library that allows easier coverage and unit testing due to automatically running a coverage report based on the argument you pass in when running the nosetest command. | 1.3.7 | Open Source | ✅ |
| Docker.py | Docker.py is a Python library that allows you to communicate with Docker via the Python language. | 4.2.0 | Open Source | ✅ |
| Psycopg2 | Psycopg2 is a Python library which allows communication with a PostgreSQL database. | 2.8.4 | Open Source | ✅ |
| Dotenv | Dotenv is a Python library for loading environment variables to process via a .env file. This is necessary for testing with environment variables due to the Flask test client running in a different environment than your OS machine. | 0.13.0 | Open Source | ✅ |
| GitHub | NCSU provides us with a GitHub repository for our project where all our development code must be remotely committed and stored. Our team plans to make use of Kanban boards on GitHub to keep track of issues and the progress of our iterations. | N/A | Open Source | ✅ |

## Design

|  |  |
| --- | --- |
| Primary Author | Jonathan Balliet |
| Secondary Author | Spencer Yoder |

### High Level Design

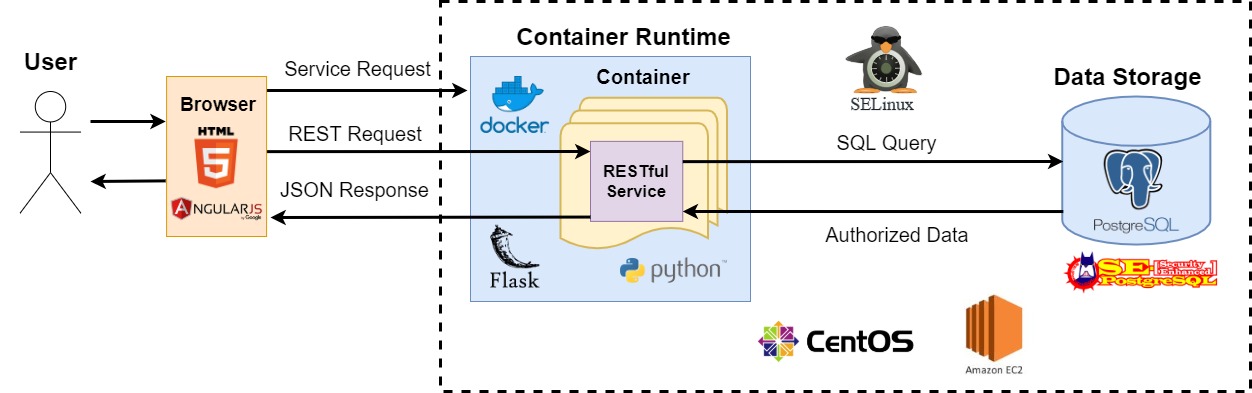
At a high level, the COps Platform is made up of two main components: the *Container Runtime* and the *Data Storage* (Figure 1). The Data Storage component is a PostgreSQL database that makes use of SELinux labels for all stored data. PostgreSQL was specifically chosen as our database due to its ability to integrate with SELinux. The SELinux module is a core technology given to us as a constraint for this system. This is because it allows a single centralized database to contain multi-level security (MLS). MLS ensures that a user can only access the data they are authorized to within this database on a kernel level. As a precondition for this system, an admin is responsible for storing and labeling all of the data in the Data Storage component via a python script [FR8].

The Container Runtime creates a Docker Container after receiving a service request from a user [FR10]. Docker Containers were chosen as the technology to run our services due to their ability to run in a specific OS image but in such a way that is both dynamic and lightweight. In comparison to virtual machines, Docker containers are far quicker to start up, require much less computing power, and can easily be created and shut down. This is a perfect fit for ephemeral services that must run in a SELinux enforced OS image to ensure secure authorization requests are made to the Data Storage component. Furthermore, containers give an extra level of security due to the running service being isolated within this container, both from the Host machine and other containers that are running simultaneously within this Container Runtime. The Container Runtime will make use of Python’s Flask framework for all Data Storage communication, handling all REST communication with the user, and for container creation and shutdown. Python was chosen as our main coding language because of its ease of use, simple integration with Docker, and the ability to quickly create a running prototype. This system is more of a *proof of concept* and one that we have a limited time to develop. Therefore, it made sense to choose a language that allows us to achieve our deliverables in a short amount of time. The Flask framework was chosen as our backend web infrastructure due to similar reasoning. It allows the ability to create a simple backend API infrastructure. Our system currently supports multiple simultaneous connections of users, but admittedly in a manner that is not very scalable. A container orchestration tool, such as Kubernetes, is required to create a scalable container system that supports a large number of users simultaneously. This aspect of the project was considered out of scope. Each of these containers runs their own RESTful service for a specific user as shown in Figure 1.

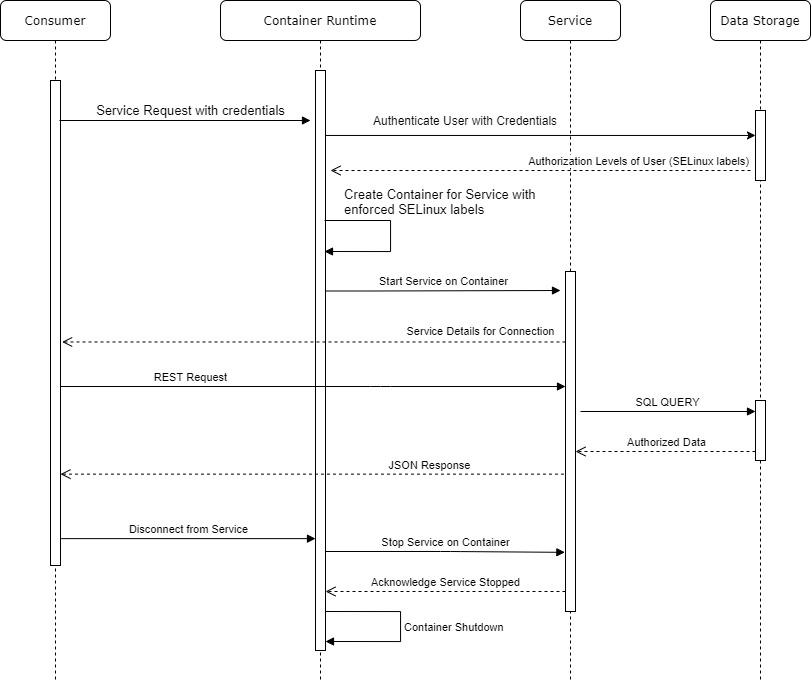
In order to create a mock system to test the COps platform, we designed the concrete Course Manager *RESTful service*. The Course Manager service will run inside of a Docker container and is spun up via a service request from the user as shown in Figure 1. This service will also make use of the Python language and the Flask framework. Keeping the resources used consistent among all components helps to make development easier for our team. Flask will be used to handle all the REST requests and responses for this service that are detailed in the Course Manager Requirements (UC1-UC7). For communicating from this mock service to the Data Storage, the Object-Relational-Mapper (ORM) library, SQLAlchemy, will be used. ORM allows us to communicate to the database using our main programming language, Python, instead of having to do this through SQL statements. This allows easier and quicker development with our database due to many of our team not being proficient in the SQL language.

The Container Runtime and Data Storage components will both be stored in a single Elastic Compute Cloud (EC2) instance using Amazon Web Services (AWS). AWS was provided for us by Skyward Federal as our cloud development environment for this project. This EC2 instance will make use of a CentOs 7 operating system image. Using this specific operating system image was given as a constraint for this system due to its capability of easily integrating with the SELinux module.

As shown in Figure 1, the user interacts with the COps platform with a browser. A front-end application was created using HTML and AngularJS which allows the user to send a service request to the Container Runtime. They are then redirected to the front-end of the requested service (in our case, this will just be Course Manager). This user's interaction via a browser will then send REST requests to the running Course Manager application in the Docker Container that then returns back a JSON response.

Figure 1: High Level System Design of the COps Platform 

The sequence diagram in Figure 2 demonstrates the events that occur when a service request is sent from a user to the COps Platform. This service request is sent with the credentials of the user to the Container Runtime. This differs from how the COps platform will work in reality due to us wanting to create a full end-to-end system. In the real COps platform the user has already been authenticated and authorized via an Identity and Access Management (IAM) system and the SELinux labels will be sent with the service request to the Container Runtime. This IAM component will eventually be integrated into the COps Platform but it is out of scope for the project deliverables this semester. In order to moc the IAM, we are having the user authenticated by the Data Storage component instead. Based on the table this user resides in, the Container Runtime can then determine the authorization levels of the user. These security labels determine what data the service is authorized and not authorized to access. After receiving this request and authenticating the user, the Container Runtime starts up a container instance to run this RESTful service. The container enforces the privileges of the running service to exactly match the authorization levels of the user in the form of their corresponding SELinux labels. This running service establishes a connection with the user, and then handles any REST requests that are sent from the user. The service communicates with the Data Storage component to process the request and perform any “create, read, update, and delete” (CRUD) actions it is authorized to do. A status code along with any authorized data is then sent back to the user as a REST response. REST requests and responses between the user and the running service continue in this manner until the user disconnects from the service. The service is then stopped on the running container. After the container receives an acknowledgement that the service has been stopped, the Container Runtime terminates this running container instance.

Figure 2: Sequence Diagram for a Service Request to the COPs Platform

### Low Level Design

#### Overview

Our System consists of two main modules on a low level: the *Container Runtime Module* and the *Course Manager Module*. The Container Runtime Module is the main running backend of the COps platform that handles service requests from the user. It handles creating a service instance via starting up a Docker Container that initializes the requested service. It monitors the entire life-cycle of this container, which includes shutting it down when the user disconnects from the service. The Course Manager Module is our concrete design for a RESTful service that the Container Runtime will launch and run within a Docker Container.

#### Container Runtime

Figure 3 contains a UML class diagram for the Container Runtime that shows our low level design for this module in detail. Container Runtime is a Flask Application, and as per convention, has an App class that is the main runner for this application. This App class contains the *docker\_client* field in order to communicate with the Docker service. Communication to Docker is accomplished using the Python library docker.py, which allows us to create a Docker client that communicates with the Docker service via the Python language.

The Service Controller class, as seen in Figure 3, is the meat of this application. The Container Runtime is a RESTful service that listens on one specific endpoint for a POST request. The credentials and the requested service name are sent as arguments in the body of this request. The first thing the Container Runtime will do is make sure the user is valid by authenticating them in the database. The Container Runtime communicates with the Data Storage, using the SQLAlchemy Python library, to see if this user indeed exists in the system. If a match is found, the authorization levels can then be determined based on which table this user resides in. Since our application is designed around mocking a university system - the tables correspond to each of the roles seen in our Course Manager service: Instructor, Coordinator, and Student. This is clearly a very simplified method to do authentication and authorization. The point was to mock how the IAM system in the real COps platform would operate on a simplified scale in order for us to demonstrate a working end to end system as our final product of this semester.

After authenticating the user, the next step is for the Container Runtime to launch this service inside a Docker Container. The abstract Service Config class contains all of the fields necessary for a service to work within our system. Our plan and thought process with designing the Container Runtime was to have its interactions with services be as generic as possible. This way more services could be added in the future that make sense with the university system concept we have designed this system around. Currently, we are only supporting one service in the form of Course Manager. Course Manager, therefore, has its own concrete service config class where the fields, such it’s login url, container name, and so on, are all statically defined. There are also static global variables contained within the Course Manager Config class. Their inclusion was necessary in order to track all running instances of this service. This guarantees no port conflicts take place since each running Docker Container must be linked to a unique port on the host machine. The global static locking mechanism is also used to ensure that the race condition of two users making a service request will always result in a unique port being assigned to each user. Finally, the active\_users static field is used to keep track of all active users in this system. If a user was allowed to create an endless amount of service requests; this could easily lead to Denial of Service (DoS) attacks in the sheer amount of computing resources required for each running container. For this reason, we restrict a specific user to only being allowed one active connection for each service at the time.

After the Container Runtime starts this service inside the Docker Container, it runs the enforce\_label method to enforce the authorization level of this user based on its unique IP address. The Container Runtime then logs into this service for the user, and returns back a response including the url where this Docker Container is running, along with the user's session cookie for authentication. While having all these Docker Containers running in background was a simple task due to Docker having built in support for this. Knowing when a user disconnected in order to shut down the container [FR13] was more of a challenge, due to this not being an easy solution to monitor this. In order to accomplish this functional requirement, the Health Check Thread class was designed. The Container Runtime starts a “health check” thread to monitor the lifecycle of this container once it has successfully started and the user has been logged in. As shown in Figure 3, this Health Check Thread class extends the Python Thread class and will continue to monitor a specific endpoint in the service every 30 seconds to see if the user’s session is still active. Once the user has become inactive or disconnects, the health check thread will detect this and know to shut down this container.

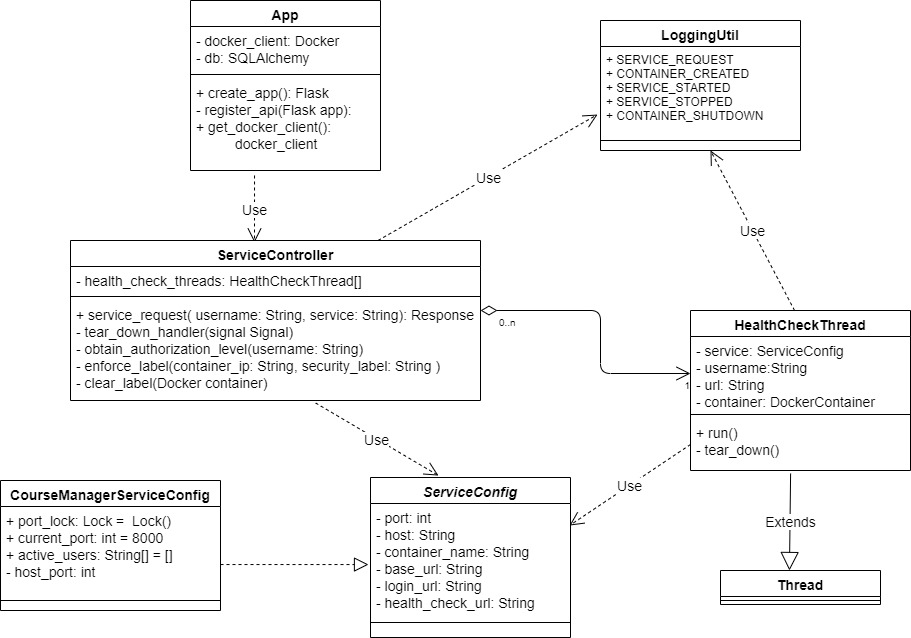


Figure 3: UML Class Diagram for the Container Runtime Module

#### Course Manager

Figure 4 shows an overview of all the classes, and their relationships, that make up the Course Manager Module. This is the module we will be implementing during our first two iterations of this project. The model classes (Student, Coordinator, Instructor, Course, and Course\_Student\_Mapping) each represent a table in our database for this mock Course Manager system. The relationships among these models are shown in more detail in Figure 5 which describes the exact schema for this database. Each of these classes extend the SQLAlchemy Model class. This essentially is an ORM wrapper that allows each Python object to interact and directly represent a SQL table in our database. The User class is an abstract class containing all the shared fields between our three user types. Each user type must concretely be contained within its own table due to how SELinux works with PostgreSQL. For this reason, the Coordinator class is created as its own class, despite having no different fields from the User abstract class it implements.

The two controller classes (User Controller and Course Controller) contain all of the REST API endpoints needed for handling the use cases defined for the Course Manager System. The User Controller contains all of the functionality relating to users in the system. This includes logging in (UC1), logging out (UC1), viewing account details (UC2), and user modification (UC4). The Course Controller similarly contains all of the functionality needed for courses. This includes course modification (UC3), grade/roster modification (UC5), and viewing course schedules (UC6). These controller classes will make use of the Flask-RESTful library in order to create a simple API that can easily be contained within Python classes. The controllers will log all transactions that occur (as defined in our logging section for each UC for Course Manager) based on the logged in user and requested functionality.

The CourseManager class contains the main Flask app that initializes all needed configurations, establishes a connection with the database, and sets the predefined API endpoints from the controller classes. This class then runs this Course Manager RESTful service, handling all incoming REST requests to the user, communicating these requests with the connected PostgreSQL database, and returning REST responses back to the user.

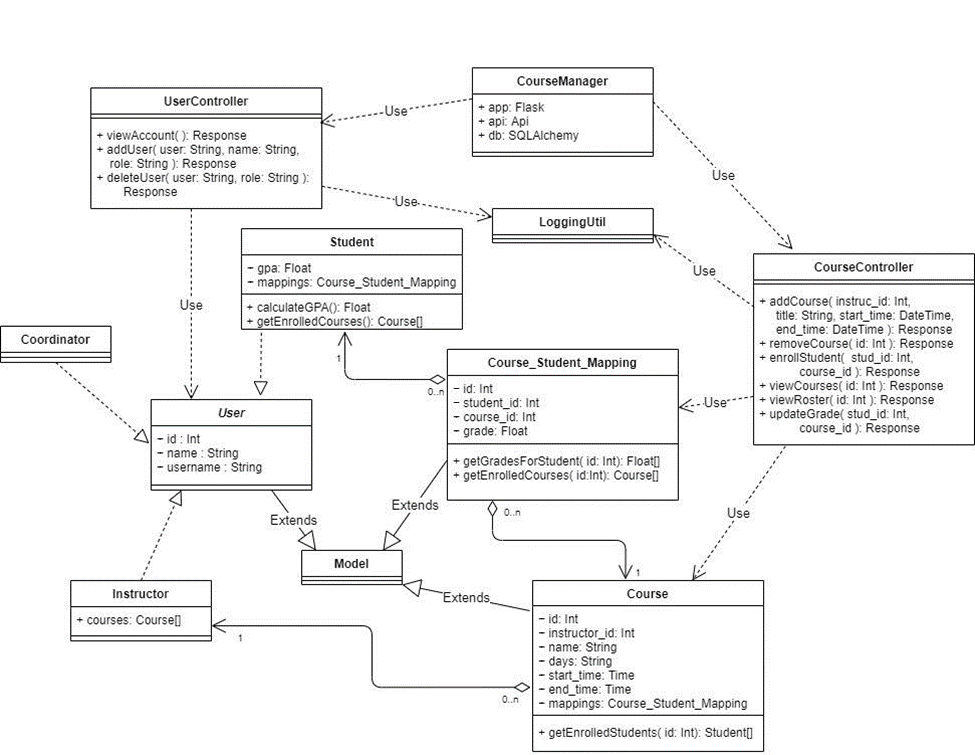
****

Figure 4: UML Class Diagram for the Course Manager Module

#### Modules:

##### Container Runtime Module

*Purpose:*

This module is responsible for handling service requests that are sent by the user. The Container Runtime isolates running services by creating a specific container instance for the service. This is based on the user’s role in the system and enforced via SELinux. This module also handles shutting down its container instances once the user has disconnected from the service.

*Responsibilities:*

* Handles the REST API endpoint for service requests from the user.
  + POST request
  + Endpoint:
    - {base\_path}/service\_request
  + The specific *service* and the user’s *username*, as String values, are sent in the body of the POST request.
  + Example:
    - service=”course\_manager”
    - username=”student”
* Authenticates user
* Determines Authorization level of user
* Creates a container instance for the requested service with authorization levels of user
  + Each service will have its own Dockerfile and corresponding Docker image. These contain the operating system image, along with all the needed dependencies, configurations, and files for running this service.
  + Enforces the SELinux label of the IP address of this container based on this user’s authorization levels.
  + Starts the RESTful service on the container instance.
* Returns a response with a message, and the url where the service is running on the host machine (if successful):
  + Example:
    - message=”Service request was successful”
    - url=”127.0.0.1:8000”
  + Returns the session cookie for this user in the header of the response
    - The Container Runtime automatically logs in this user, to obtain this session, before sending the request back.
* Shut downs running container instances
  + The container instance stops a running service when the user has disconnected from the service.
  + This is accomplished via a health check thread that monitors a special health check endpoint on the running service.
  + The health check thread contacts this endpoint every 30 seconds to see if the user’s session is still active.
  + Destroys the container instance when notified by the container of the service being stopped
* Logs all transactions

##### Course Manager Service Module

*Purpose:*

This module is responsible for implementing the mock Course Manager system as defined in our Course Manager System requirements. It handles all REST requests from the user, communicating these requests to the Data Storage, and returning the JSON response (along with any authorized data) back to the user.

*Responsibilities:*

* Establishes connection to the Data Storage (PostgreSQL database) after startup
  + Connects to the database using statically configured credentials.
* Authentication of user via login REST API endpoint (UC1)
  + POST request
  + Handles logining in of all users
  + Endpoint:
    - {base\_path}/api/login
  + The user’s username is sent in the body of the POST request
  + Example request:
    - username=”student\_test”
  + (***NOTE:*** For the purpose of this mock system, the system only looks to see if a user exists with this username. In a real system, a password, that is stored with a secure hashing and a salt, would be used in conjunction.)
  + Returns a response containing a session cookie with a randomly generated value to maintain authentication with this user for the session.
* Viewing user account details via REST API endpoint (UC2-S1)
  + GET request
  + Handles request to view logged in user’s account details
  + Endpoint:
    - {base\_path}/api/user
  + Returns a response with a status code. If successful (200) the user’s account details, including their username, name, and id are contained in the response.
  + Example 200 status response:
    - id=”1”
    - name=”Student Tester”
    - username=”student\_test”
    - gpa=4.0
* Adding new user via REST API endpoint (UC4-S1)
  + POST request
  + Handles requests to add a new user to the system.
  + Endpoint:
    - {base\_path}/api/user
  + The user’s username, name, and role are sent in the body of the POST request
  + Example request:
    - username=”student\_test2”
    - name=”Student Tester2”
    - role=”student”
  + Returns a response with the status code and a message for whether adding a new user was successful or not.
* Deleting existing user via REST API endpoint (UC4-S2)
  + DELETE request
  + Handles requests to delete an existing user from the system.
  + Endpoint:
    - {base\_path}/api/user
  + The user’s username and role are sent in the body of the DELETE request
  + Example request:
    - username=”student\_test”
    - role=”student”
  + Returns a response with the status code and a message for whether deleting an existing user was successful or not.
* Adding new course via REST API endpoint (UC3-S1)
  + POST request
  + Handles request for adding a new course to the system
  + Endpoint:
    - {base\_path}/api/course
  + The course’s name, days, start\_time, end\_time, along with an existing instructor's id are sent in the body of the POST request.
  + Example:
    - name=”CSC316”
    - days=”MW”
    - start\_time=”11:15”
    - end\_time=”12:30”
    - instructor\_username=”jtking”
  + Returns a response with the status code and a message for whether creating a new course was successful or not.
* Removing existing course via REST API endpoint (UC3-S2)
  + DELETE request
  + Handles request to delete an existing course from the system
  + Endpoint:
    - {base\_path}/api/course
  + The course’s id is sent in the body of the DELETE request.
  + Example:
    - course\_name=”CSC316”
  + Returns a response with the status code and a message for whether deleting an existing course was successful or not.
* Adding student to an existing course via REST API endpoint (UC3-S3)
  + POST request
  + Handles request to enroll a student to an existing course
  + Endpoint:
    - {base\_path}/api/mapping
  + The student’s username and the course’s id are sent in the body of the POST request.
  + Example:
    - username=”student tester”
    - course\_name=”CSC316”
  + Returns a response with the status code and a message for whether enrolling the student in the course was successful or not.
* Modifying a student’s grade for enrolled course via REST API endpoint (UC5-S1)
  + PUT request
  + Handles request to update a student’s grade for a course
  + Endpoint:
    - {base\_path}/api/mapping/grade
  + The course’s id and the student’s username are sent in the body of the PUT request.
  + Example:
    - course\_name=”CSC316”
    - username=”student tester”
    - grade=”4.0”
  + Returns a response with the status code and a message for whether updating the student’s grade was successful or not.
* Viewing course schedule via REST API endpoint for Student (UC6-S2)
  + GET request
  + Handles request to view a student’s course schedule with grades
  + Endpoint:
    - {base\_path}/api/schedule
  + Returns a response with a list of course objects, along with an associated grade field.
  + Example response:

[

name=”CSC316”

days=”MW”

start\_time=”11:15”

end\_time=”12:30”

instruct\_name=”Jason King”

grade=4.0,

name=”CSC333”

days=”TH”

start\_time=”2:15”

end\_time=”1:30”

instruct\_name=”Sarah Heckman”

grade=2.0

]

* Viewing course schedule via REST API endpoint for Instructor (UC6-S3)
  + GET request
  + Handles request to view an instructor’s course schedule with list of enrolled students
  + Endpoint:
    - {base\_path}/api/schedule
  + Returns a response with a list of course objects, along with an associated list of student names.
  + Example response:

[

name=”CSC316”

days=”MW”

start\_time=”11:15”

end\_time=”12:30”

students=[ “Jeen”, “Caleb”, “Spencer”, “Daniel”, “Jonathan” ],

name=”CSC492”

days=”TH”

start\_time=”9:30”

end\_time=”10:45”

students=[ “Jeen”, “Caleb”, “Spencer”, “Daniel”, “Jonathan” ]

]

* Disconnection of user via user logout endpoint
  + GET request
  + Handles logging out of all users
  + Endpoint:
    - {base\_path}/api/logout
  + Disconnects from the database
  + Notifies Container Runtime of service shutdown
  + Shutdown running application
* Logs all transactions

##### Logging Util Module

*Purpose:*

This utility module is responsible for logging all transactions that occur within each component.

*Responsibilities*

* Each module contains its own Logging Util component with the static codes and descriptions for the specific logging transactions it handles. These logging transactions are referenced in our Overall Requirements for the Container Runtime Module and in the Course Manager Requirements for the Course Manager Module.

### Database Design

Figure 5 details the database schema for CourseManager. Since we are primarily testing security, we designed a minimally complex database to meet the following criteria:

* Contains tables with different access rights (tables accessible by a single user and combinations of users)
* Contains columns with different access rights than their parent tables
* Contains one-to-many and many-to-many relationships

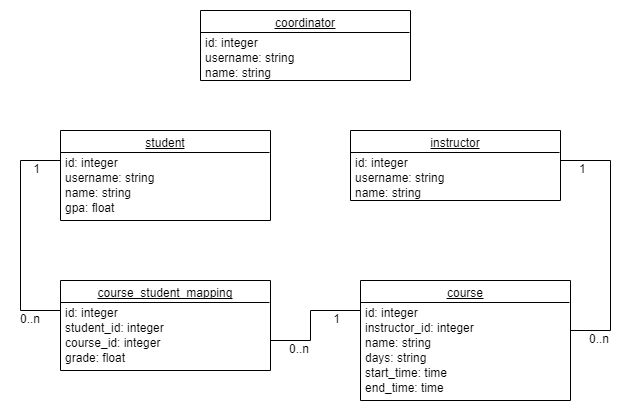


Figure 5: Database Schema for Course Manager Database

This diagram matches the final state of the database at project completion. Note that a many-to-many relationship is present between student and course through course\_student\_mapping; however, this table is also used to store a student’s grade in a given course. It’s also worth noting that the gpa column in students is unused in the application. This is an artifact of an earlier design which is still present in the database.

### Security Policy Design

Our goal for the security policy was to enforce the access rights represented in figure 6:

## 

Figure 6: access rights for CourseManager  
Dotted lines represent read access, solid lines represent read/write access

Based on these access rights, we were able to divide the system up into 3 SELinux user types: **coordinator\_t, instructor\_t, and student\_t**. And 5 SELinux data types, each corresponding to a different set of access rights: **coordinator\_data\_t, instructor\_data\_t, student\_data\_t, course\_data\_t, and grade\_data\_t**. Figures 7 and 8 show the division of access rights for each of these types:

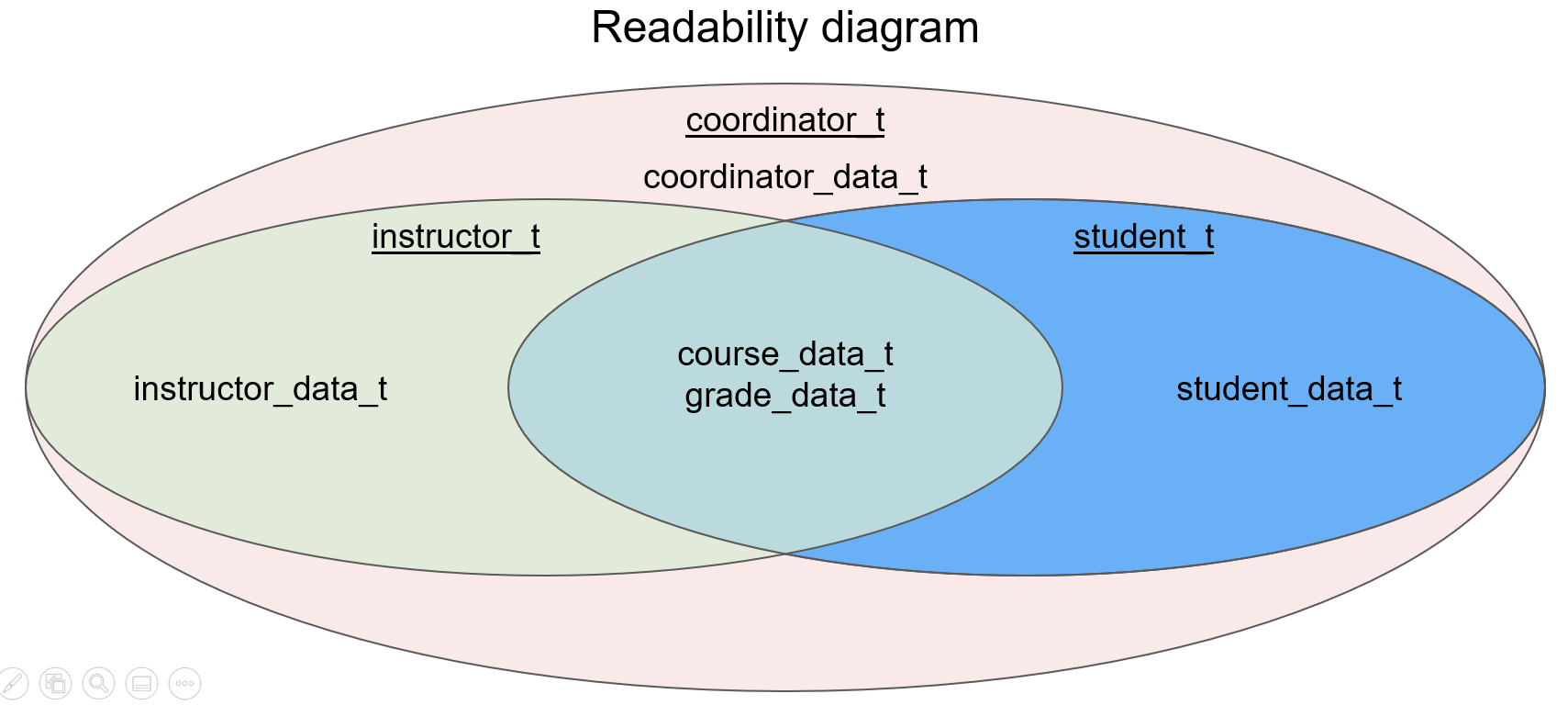


Figure 7: read access rights in the security policy

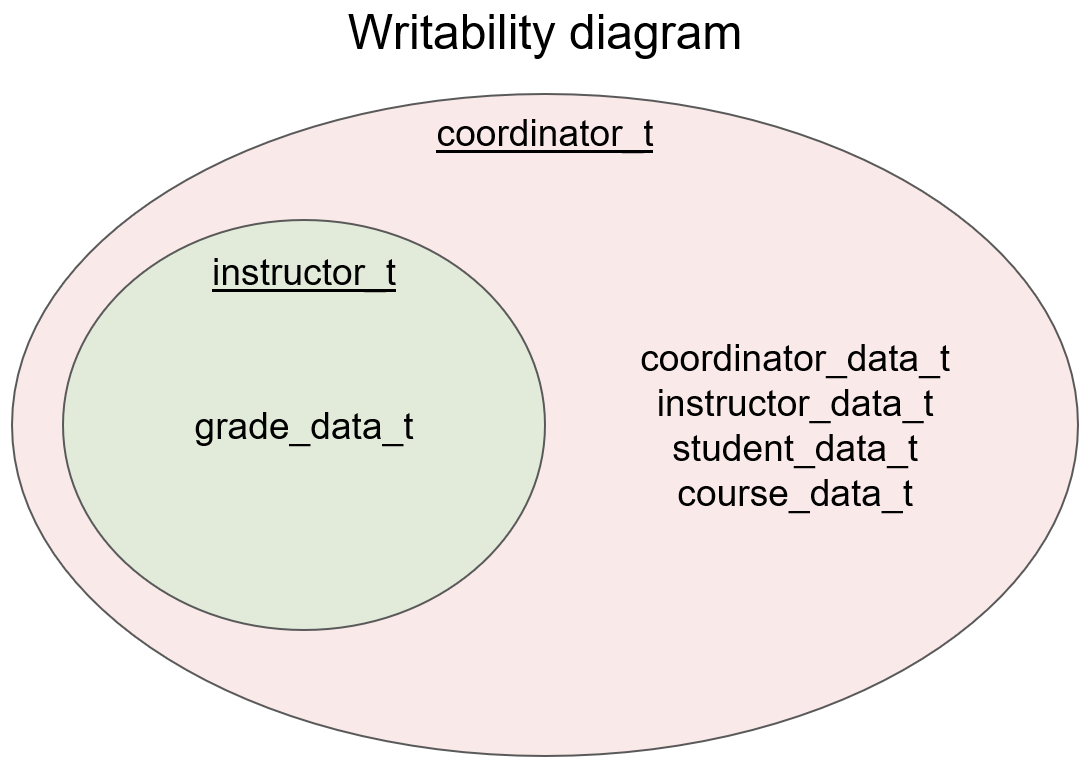


Figure 8: write access rights in the security policy

These data types were used to label the database as shown in figure 9:

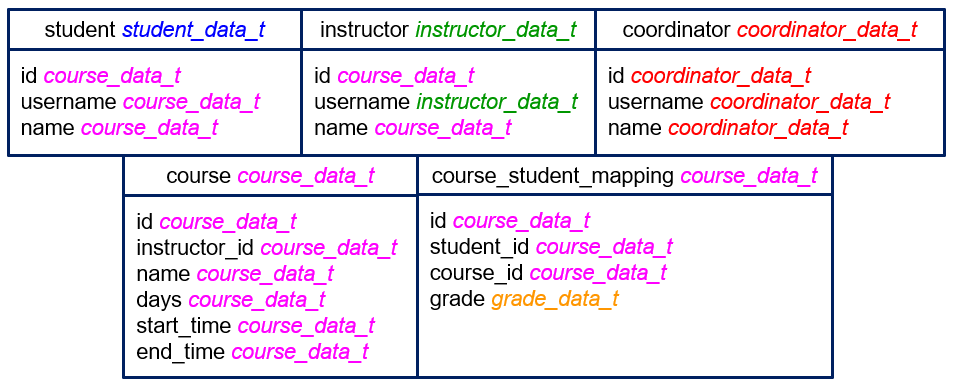


Figure 9: database labels

Note that the student table has no columns visible only to the student. This is because we found, after implementing our use cases, that instructors need to have access to all these columns in order to give students grades. Any database object not included in this table is labeled as **unlabeled\_t**, SEPostgres’s default label.

### Front-end Design

The goal of the front-end for this project was to have a convenient, easy-to-understand way to demonstrate our security functionalities and run system tests. A significant part of this was intentionally making all options available to every user type. The front-end should allow each user to attempt to make every API call in the system, which would show that our SELinux security policy does indeed deny requests when it is supposed to.

The front-end design was done in Balsamiq, which is a tool to create mockups of user interfaces. We created a design based around a simple web application, in which a user logs in and is presented with a menu of all the different actions in the system. Each item in the action menu corresponds to one API call in the system. The user fills out any necessary data to make the API call, which is then either allowed or denied in the back-end by SELinux. It was also convenient to completely separate this from the back-end, because it allowed us to turn off SELinux with a simple command and demonstrate how that affects the application’s behavior in the front-end. With SELinux turned off, no requests are denied, so every user can perform every action in the system.

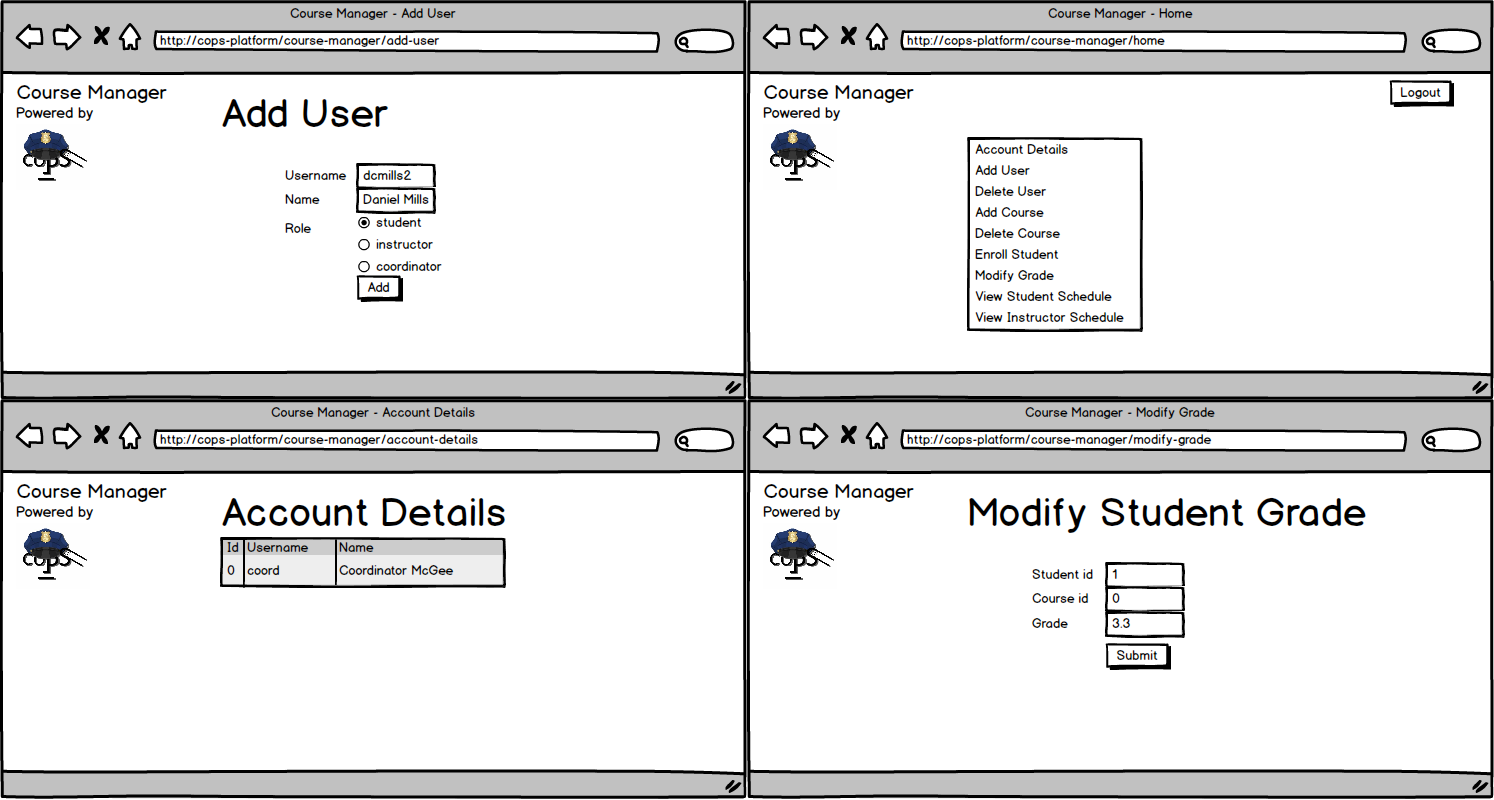


Figure 10: Front-end mockups

## Implementation

|  |  |
| --- | --- |
| Primary Author | Daniel Mills |
| Secondary Author | Jonathan |

#### Final Implementation Status

COps currently consists of the following components:

* A web-based front-end
  + Written in HTML/AngularJS
  + Run in Firefox and connected to the EC2 machines via SSH tunnel
* A container runtime application
  + Written in Python using the Flask framework
  + Authenticates users attempting to connect by REST request
  + Dynamically starts up Docker containers running CourseManager
    - Which connect to Postgres via a labeled TCP socket
  + Labels container connections based on the connected user type via a bash script
  + Shuts down containers after users have disconnected
* Course Manager
  + Another Flask application
  + Comprised of
    - Models, which correspond to database objects
      * coordinator.py
      * instructor.py
      * student.py
      * course.py
      * course\_student\_mapping.py
    - Controllers, which service REST requests
      * course\_controller.py
      * health\_check\_controller.py
      * user\_controller.py
* An SELinux security policy module
  + Written in SELinux’s [Kernel Policy Language](https://selinuxproject.org/page/PolicyLanguage#Kernel_Policy_Language)
  + Compiled and built onto the system’s SELinux policy rules
* A Postgres database
  + Built from the Postgres master branch with the SEPostgres module
  + Data is generated and labeled via the Python script: generate\_db.py

### Iterations

#### Iteration 0

* Research the technologies pertinent to our system
* Create proper requirements and design documents
* Acquire access to the resources needed for development

#### Iteration 1

* Design, configure, and populate a properly labeled sample database
* Configure a Docker container with the proper technologies installed
* Create the infrastructure to facilitate a single, successful REST API call
* Use Cases and Requirements:
  + UC1, UC2, UC6
  + FR1-FR4, FR7-FR9

#### Iteration 2

* Design and implement all remaining API calls, providing the functionality of creating data, editing data, and viewing reports
* Ensure API calls are properly executed or denied based on SELinux labels
* Update database configuration to handle new API calls
* Implement logging
* Use Cases and Requirements:
  + UC3, UC4, UC5, UC7
  + FR6

#### Iteration 3

* Dockerize the running service in a container
* Stretch goals:
  + Implement a simple front-end for demonstration purposes
* Requirements:
  + FR10-FR13

#### Iteration 4

* Ensure all unit and system tests pass
* Consolidate final deliverables and documentation
  + Final handoff presentation, final project report, supplemental guides

### Security Considerations

#### Confidentiality

Confidentiality is integral to the security system we are implementing. Since we are using SELinux labels to restrict access to certain types of data, any system that uses the COps Platform can improve its confidentiality. The Course Manager system demonstrates this through its different user types. It would violate confidentiality if a student was able to see another student’s grade in the database. With our SEPostgreSQL configuration in place, a student is only able to view their own grades.

#### Integrity

One of the original goals for the project was to encrypt data during transit. While we did not end up implementing encryption, a future team might consider implementing this feature to improve integrity. Encryption ensures that data is not compromised when sent from the front-end to the application, or from the application to the data storage. Additionally, integrity is improved by having security at the database label. The usage of SELinux labels stored with data in the database ensures that data cannot be modified or destroyed by SQL injection. Only authorized requests will be carried out.

#### Availability

Availability is achieved by the REST API endpoints in our implementation. These APIs provide a simple interface for a user to access the data that they are authorized to access. Our application also has a front-end interface to make the process simple and user-friendly. The container runtime component also helps ensure availability. Since the container runtime is always listening for new connections, a new user will always be able to connect to the application and start up a new container. Even if other users are using the system at the same time, new service requests will be accepted.

#### Identification & Authentication

Each user in our system has a unique user ID that they use to authenticate. Once authenticated, they are given a randomly-generated session cookie that is sent with each API call to ensure that the user is properly authenticated to make each request. This ensures that only authenticated users can use the system, and that one user cannot mimic another user with different security privileges. Another potential addition to the project to improve in this area would be an Identification and Access Management (IdAM) component. This component was included in the original project design, however it did not make it to our final implementation. A future team might consider implementing an IdAM to handle authentication.

#### Accountability

Accountability is ensured via the logs throughout our system. At the database level, all requests, both successful and denied, are written to a log. These logs include the username associated with the request, as well as which data they were trying to access. These logs allow all actions in the system to be traced to the user performing the action. Even if a breach in the data security were to happen, the logs would point to the person who caused it.

#### Privacy

Our system helps ensure privacy by having a customizable SELinux security policy that can be tweaked to any application’s needs. Our Course Manager application, for example, is configured so that a student is not able to access or modify another student’s data. There is one flaw which could be improved upon, which is that an instructor can modify the grade of any student in any course, even if they are not the instructor for that course. This could be a breach of privacy since an instructor is allowed to modify another instructor’s data. However, since we are using custom SELinux labels, this could be changed with SELinux commands. Another system that prioritizes that kind of privacy could easily implement it using the COps platform.

### Project Structure

#### Top-Level

The Top-Level directory (2020SpringTeam32) of our project is pictured in Figure 11 and 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 applications, 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.

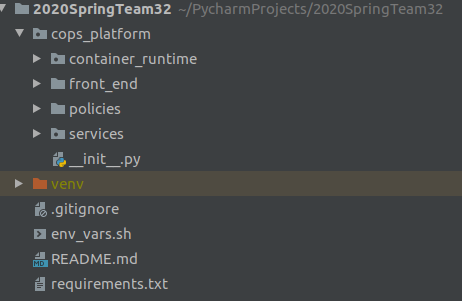


Figure 11 - Top Level Project Structure

#### 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). This directory, container\_runtime, is pictured in Figure 12 and contains the following contents:

* \_\_init\_\_.py
  + This file 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.

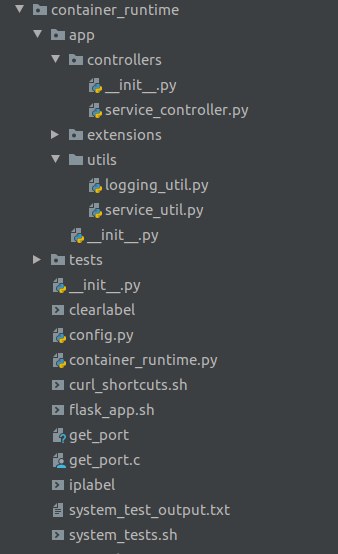


Figure 12 - Container Runtime Project Structure

#### 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.

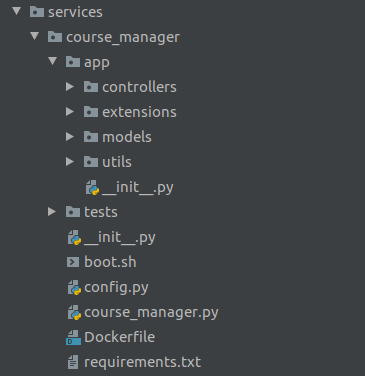
The Course Manager tests/controllers directory is located at the following location:

(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.

Figure 13 - Course Manager Project Structure

#### 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 and shown in Figure 14:

* 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.

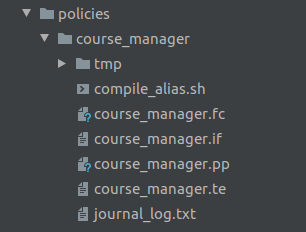


Figure 14 - Policies Directory Project Structure

#### Front-end

The front\_end directory (2020SprintTeam32/cops\_platform/front\_end) is shown in Figure 15 and 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 contains 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.

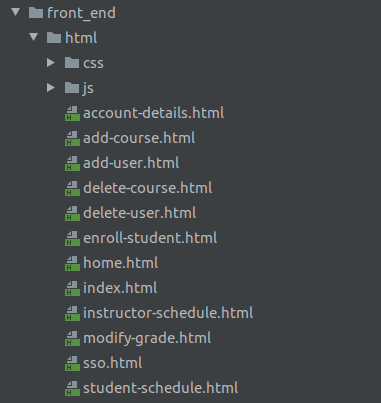


Figure 15 - Frontend Project Structure

## Test Plan & Results

|  |  |
| --- | --- |
| Primary Author | Caleb Boswell |
| Secondary Author | Jonathan Balliet |
| Editor | Daniel Mills |

### Overview

#### Blackbox testing tools:

System tests were done via two interfaces: the HTML/AngularJS front end, and the terminal using curl commands. These tests were formalized in a black box test plan, which was run manually, and a bash script, which was run automatically and was primarily used to ensure access rights were correctly enforced for the three CourseManager user types.

#### Whitebox testing tool:

The unittest module in Python will be used for doing all of our unit and integration testing.

#### Expected Coverage:

We are hoping to achieve at least a minimum of 80% statement coverage for each of the applications featured in our project.

#### System:

System testing for this project will be done by automating the REST API calls given the Course Manager Service is running with the security context of a specific user. This allows us to test whether that user, with a specific security context, can access or modify the database in a way that matches their authorization levels and the security policy of the database.

#### Security:

For security it will be necessary to test every combination of access control. For example we will use three users, one with low level security access, one with moderate level security access, and a third with administrator access. We will have each of these users try to access data from each level of security via REST API endpoints. Security is paramount for our system and will be the main focus of our system testing.

#### Unit:

The unit testing will not need to be as extensive as our system testing, however, we plan to maintain, at the minimum, 80% statement coverage. The unit tests will ensure that the main logic of all components for this system is working correctly and as expected. This way we can feel confident in testing the complex security interactions between our mock course manager system and the database.

### Backend System Test Plan

The file cops\_platform/container\_runtime/system\_tests.sh is a bash script which runs through each API call as each of the three user types.

#### Preconditions

* The user has loaded the environment variables from 2020SpringTeam32/env\_vars.sh with the command:  
  $ source env\_vars.sh
* The current working directory is cops\_platform/container\_runtime
* All the preconditions for launching container runtime (see ‘Running Container Runtime’ in the Developer’s Guide) are met.
* The system is running SELinux in ‘enforced’ mode.

#### Execution Instructions

$ sh system\_tests.sh

#### Expected Results

Much of the output from running these tests is irrelevant to the correctness of the tests, so the results should be compared against the following table. If a more accurate list of expected results is desired, results can be compared against

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

These results are not fully accurate either, since not all output could be piped to this text file, but they include more of the terminal output.

|  |  |  |  |
| --- | --- | --- | --- |
| **Command** | **Coordinator** | **Instructor** | **Student** |
| View account info | {"id": 1, "username": "coordinator", "name": "Coordinator"} | {"id": 1, "username": "instructor", "name": "Instructor"} | {"id": 1, "username": "student", "name": "Student", "gpa": 4.0} |
| View schedule | [] | [Long list of courses] | [Shorter list of courses] |
| Add student | "User successfully added!" | {"message": "Access Denied."} | {"message": "Access Denied."} |
| Add instructor | "User successfully added!" | {"message": "Access Denied."} | {"message": "Access Denied."} |
| Add coordinator | "User successfully added!" | {"message": "Access Denied."} | {"message": "Access Denied."} |
| Delete student | "User successfully deleted!" | {"message": "Access Denied."} | {"message": "Access Denied."} |
| Delete instructor | "User successfully deleted!" | {"message": "Access Denied."} | {"message": "Access Denied."} |
| Delete coordinator | "User successfully deleted!" | {"message": "Access Denied."} | {"message": "Access Denied."} |
| Add course | "Course successfully added!" | {"message": "Access Denied."} | {"message": "Access Denied."} |
| Enroll student | "Student successfully enrolled!" | {"message": "Access Denied."} | {"message": "Access Denied."} |
| Set grade | "Student's grade successfully modified!" | "Student's grade successfully modified!" | {"message": "Access Denied."} |
| Delete empty course | "Course successfully deleted!" | {"message": "Access Denied."} | {"message": "Access Denied."} |
| Delete course with enrolled students | "Course successfully deleted!" | {"message": "Access Denied."} | {"message": "Access Denied."} |

#### Actual Results

The contents of system\_test\_output.txt are the portion of the test we were able to pipe to a file and match our expected results.

### Frontend System Test Plan

#### Testing Overview

The System Test Plan consists of a suite of black box tests that can be performed manually using the front-end interface for Course Manager. The tests cover each use case for Course Manager, including error cases and alternative flows. Each test cases has a test ID, a description of the steps required to perform the test, the expected results, and the actual results.

#### Preconditions

* The COps Platform and Course Manager have been installed and configured as described in the Installation Guide
* The generate\_db.py script has been run, and no other changes to the database have been made

#### Running the Course Manager Application

To perform the tests, run the application and connect to if in Firefox as described in the User Guide.

#### Test Data Summary

The following users exist in the Course Manager system:

*Tester #1*

**Username:** instructor

**Name:** Instructor

**SELinux type:** instructor\_t

*Tester #2*

**Username:** student

**Name:** Student

**SELinux type:** student\_t

*Tester #3*

**Username:** coordinator

**Name:** Coordinator

**SELinux type:** coordinator\_t

#### Acceptance Test Plan

|  |  |  |  |
| --- | --- | --- | --- |
| **Test ID** | **Description** | **Expected Results** | **Actual Results** |
| **InstructorLogin**  **(Testing UC1)** | **Preconditions:**  User *instructor* is registered in the system.  **Steps:**   1. Enter “instructor” in the username text box. 2. Enter any text into the password text box (it is ignored currently) 3. Click the Submit button. | The Course Manager home page is successfully transitioned to after clicking Submit. | The Course Manager home page is successfully transitioned to after clicking Submit. |
| **StudentLogin**  **(Testing UC1)** | **Preconditions:**  User *student* is registered in the system  **Steps:**   1. Enter “student” in the username text box. 2. Enter any text into the password text box (it is ignored currently) 3. Click the Submit button. | The Course Manager home page is successfully transitioned to after clicking Submit. | The Course Manager home page is successfully transitioned to after clicking Submit. |
| **CoordinatorLogin**  **(Testing UC1)** | **Preconditions:**  User *coordinator* is registered in the system  **Steps:**   1. Enter “coordinator” in the username text box. 2. Enter any text into the password text box (it is ignored currently) 3. Click the Submit button. | The Course Manager home page is successfully transitioned to after clicking Submit. | The Course Manager home page is successfully transitioned to after clicking Submit. |
| **NoAccountLogin**  **(Testing UC1)** | **Preconditions:**  User *unregistered* is not registered in the system.  **Steps:**   1. Enter “unregistered” in the username text box 2. Enter any text into the password text box (it is ignored currently) 3. Click the Submit button. | The user is not allowed to login and the Course Manager home page is not transitioned to. | Response:  ERROR 400  BAD REQUEST  Given credentials are not valid |
| **ViewAccountDetailsInstructor**  **(Testing UC2)** | **Preconditions:**  The ***InstructorLogin*** test has passed.  Currently on the Home Page of Course Manager.  **Steps:**   1. Select the “Account Details” options in the list of options on the Home Page. | The following information is displayed:  Id:  (given id)  Username:  instructor  Name:  Instructor | The following information is displayed:  Id:  1  Username:  instructor  Name:  Instructor |
| **ViewAccountDetailsStudent**  **(Testing UC2)** | **Preconditions:**  The ***StudentLogin*** test has passed.  Currently on the Home Page of Course Manager.  **Steps:**   1. Select “Account Details” option in the list of options on the home page. | The following information is displayed:  ID:  (given id)  Username:  student  Name:  Student  GPA:  4.0 | The following information is displayed:  ID:  1  Username:  student  Name:  Student  GPA:  4 |
| **ViewAccountDetailsCoordinator**  **(Testing UC2)** | **Preconditions:**  The ***CoordinatorLogin*** test has passed.  Currently on the Home Page of Course Manager.  **Steps:**   1. Select “Account Details” option in the list of options on the home page. | The following information is displayed:  ID:  (given id)  Username:  coordinator  name:  Coordinator | The following information is displayed:  ID:  1  Username:  coordinator  name:  Coordinator |
| **AddCourse**  **(Testing UC 3 - S1)** | **Preconditions:**  The ***CoordinatorLogin*** test has passed.  **Steps:**   1. Select “Add Course” option in the list of options on the home page. 2. Enter “CSC492” in name textbox 3. Enter “TH” in days textbox 4. Enter “12:50” in start time textbox 5. Enter “2:40” in end time textbox 6. Enter “instructor” in instructor textbox 7. Click the “Add” button | Course successfully added and no error returned. | Course successfully added and no error returned. |
| **AddCourseInstructor**  **(Testing UC3)** | The ***InstructorLogin*** test has passed.  **Steps:**   1. Select “Add Course” option in the list of options on the home page. 2. Enter “CSC492” in name textbox 3. Enter “TH” in days textbox 4. Enter “12:50” in start time textbox 5. Enter “2:40” in end time textbox 6. Enter “instructor” in instructor textbox 7. Click the “Add” button | Course is not added and error is returned saying user role does not have access to this function |  |
| **AddCourseStudent**  **(Testing UC3)** | The ***StudentLogin*** test has passed.  **Steps:**   1. Select “Add Course” option in the list of options on the home page. 2. Enter “CSC492” in name textbox 3. Enter “TH” in days textbox 4. Enter “12:50” in start time textbox 5. Enter “2:40” in end time textbox 6. Enter “instructor” in instructor textbox 7. Click the “Add” button | Course is not added and error is returned saying user role does not have access to this function |  |
| **DeleteCourse**  **(Testing UC 3 - S2)** | **Preconditions:**  The ***CoordinatorLogin*** test has passed.  ***AddCourse*** test has passed.   1. Select “Delete Course” option in the list of options on the home page 2. Enter “CSC492” in course name textbox 3. Click the “Delete” button | Course successfully deleted and no error returned. | Course successfully deleted and no error returned. |
| **DeleteCourseInstructor**  **(Testing UC 3)** | **Preconditions:**  The ***InstructorLogin*** test has passed.  ***AddCourse*** test has passed.   1. Select “Delete Course” option in the list of options on the home page 2. Enter “CSC492” in course name textbox 3. Click the “Delete” button | Course is not deleted and error is returned saying user role does not have access to this function. |  |
| **DeleteCourseStudent**  **(Testing UC 3)** | **Preconditions:**  The ***InstructorLogin*** test has passed.  ***AddCourse*** test has passed.   1. Select “Delete Course” option in the list of options on the home page 2. Enter “CSC492” in course name textbox 3. Click the “Delete” | Course is not deleted and error is returned saying user role does not have access to this function. |  |
| **EnrollStudent**  **(Testing UC 3 - S3)** | **Preconditions:**  The ***CoordinatorLogin*** test has passed.  ***AddCourse*** test has passed.   1. Select the “Enroll Student” option in the list of options on the home page. 2. Enter “student” in the username textbox 3. Enter “CSC492” in the course name text box 4. Click the “Enroll” button | Student successfully enrolled and no error returned. | Student successfully enrolled and no error returned. |
| **EnrollStudentInstructor**  **(Testing UC 3 - S3)** | **Preconditions:**  The ***InstructorLogin*** test has passed.  ***AddCourse*** test has passed.   1. Select the “Enroll Student” option in the list of options on the home page. 2. Enter “student” in the username textbox 3. Enter “CSC492” in the course name text box 4. Click the “Enroll” button | Student is not enrolled and error is returned saying user role does not have access to this function |  |
| **EnrollStudentStudent**  **(Testing UC 3 - S3)** | **Preconditions:**  The ***StudentLogin*** test has passed.  ***AddCourse*** test has passed.   1. Select the “Enroll Student” option in the list of options on the home page. 2. Enter “student” in the username textbox 3. Enter “CSC492” in the course name text box 4. Click the “Enroll” button | Student is not enrolled and error is returned saying user role does not have access to this function |  |
| **AddCourseInvalid**  **(Testing UC 3 - E1)** | **Preconditions:**  The ***CoordinatorLogin*** test has passed.  ***AddCourse*** test has passed.  **Steps:**   1. Select “Add Course” option in the list of options on the home page. 2. Enter “CSC417” in name textbox 3. Enter “TH” in days textbox 4. Enter “12:50” in start time textbox 5. Enter “2:40” in end time textbox 6. Enter “none” in instructor textbox 7. Click the “Add” button | Course not created and error is returned | Response:  ERROR 400  BAD REQUEST  Error handling the addition of this course. |
| **EnrollStudentInvalid**  **(Testing UC 3 - E2)** | **Preconditions:**  The ***CoordinatorLogin*** test has passed.  ***EnrollStudentTest*** has passed.   1. Select the “Enroll Student” option in the list of options on the home page. 2. Enter “none” in the username textbox 3. Enter “CSC492” in the course name text box 4. Click the “Enroll” button | Enrollment unsuccessful and error is returned | Response:  ERROR 400  BAD REQUEST  Error with enrolling student |
| **EnrollStudentNoDuplicate**  **(Testing UC 3 - E3)** | **Preconditions:**  The ***CoordinatorLogin*** test has passed.  ***EnrollStudentTest*** has passed.  **Steps:**   1. Select the “Enroll Student” option in the list of options on the home page. 2. Enter “student” in the username textbox 3. Enter “CSC492” in the course name text box 4. Click the “Enroll” button | Enrollment is unsuccessful and error returns saying student is already enrolled | Response:  ERROR 400  BAD REQUEST  The student is already enrolled in this course |
| **AddNewUserInstructor**  **(Testing UC4-S1)** | **Preconditions:**  The ***CoordinatorLogin*** test has passed.  The ***InstructorLogin*** test has passed.  **Steps:**   1. Select the “Add User” option in the list of options on the home page 2. Select “Instructor” user type 3. Input “instructor2” into username textbox 4. Input “Instructor2” into name textbox 5. Click the “Add” button | User added successfully and no error is returned. | User added successfully and no error is returned. |
| **AddNewUserInstructorDuplicate**  **(Testing UC4-E1)** | **Preconditions:**  The ***CoordinatorLogin*** test has passed.  The ***InstructorLogin*** test has passed.  The ***AddNewUserInstructor*** test has passed.  **Steps:**   1. Select the “Add User” option in the list of options on the home page 2. Select “Instructor” user type 3. Input “instructor” into username textbox 4. Input “Instructor” into name textbox 5. Click the “Add” button | User not added and error returns saying users with same username cannot be added | Response:  ERROR 400  BAD REQUEST  Failed to add a new user. A user with this username already exists in the system. |
| **AddNewUserStudent**  **(Testing UC4-S1)** | **Preconditions:**  The ***CoordinatorLogin*** test has passed.  The ***StudentLogin*** test has passed.  **Steps:**   1. Select the “Add User” option in the list of options on the home page 2. Select “Student” user type 3. Input “student2” into username textbox 4. Input “Student2” into name textbox 5. Click the “Add” button | User successfully added and no error returned | User successfully added and no error returned |
| **AddNewUserStudentDuplicate**  **(Testing UC4-E1)** | **Preconditions:**  The ***CoordinatorLogin*** test has passed.  The ***StudentLogin*** test has passed.  The ***AddNewUserStudent*** test has passed  **Steps:**   1. Select the “Add User” option in the list of options on the home page 2. Select “Student” user type 3. Input “student” into username textbox 4. Input “Student” into name textbox 5. Click the “Add” button | User is not added and error returns saying users cannot be added with same usernames | Response:  ERROR 400  Response:  ERROR 400  BAD REQUEST  Failed to add a new user. A user with this username already exists in the system. |
| **AddNewUserCoordinator**  **(Testing UC4-S1)** | **Preconditions:**  The ***CoordinatorLogin*** test has passed.  **Steps:**   1. Select the “Add User” option in the list of options on the home page 2. Select “Coordinator” user type 3. Input “coordinator2” into username textbox 4. Input “Coordinator2” into name textbox 5. Click the “Add” button | User successfully added and no error is returned. | User successfully added and no error is returned. |
| **AddNewUserCoordinatorDuplicate**  **(Testing UC4-E1)** | **Preconditions:**  The ***CoordinatorLogin*** test has passed.  The ***AddNewUserCoordinator*** test has passed  **Steps:**   1. Select the “Add User” option in the list of options on the home page 2. Select “Coordinator” user type 3. Input “coordinator” into username textbox 4. Input “Coordinator” into name textbox 5. Click the “Add” button | User is not added and error is returned saying users cannot be added with duplicate usernames | Response:  ERROR 400  BAD REQUEST  Failed to add a new user. A user with this username already exists in the system. |
| **AddNewUserInstructorByInstructor**  **(Testing UC4-S1)** | **Preconditions:**  The ***InstructorLogin*** test has passed.  **Steps:**   1. Select the “Add User” option in the list of options on the home page 2. Select “Instructor” user type 3. Input “instructor2” into username textbox 4. Input “Instructor2” into name textbox 5. Click the “Add” button | User is not added and error is returned saying user role does not have access to this function |  |
| **AddNewUserStudentByStuddent**  **(Testing UC4-S1)** | **Preconditions:**  The ***StudentLogin*** test has passed.  **Steps:**   1. Select the “Add User” option in the list of options on the home page 2. Select “Student” user type 3. Input “student2” into username textbox 4. Input “Student2” into name textbox 5. Click the “Add” button | User is not added and error is returned saying user role does not have access to this function |  |
| **DeleteUserInstructor**  **(Testing UC4-S2)** | **Preconditions:**  The ***CoordinatorLogin*** test has passed.  The ***InstructorLogin*** and ***AddNewUserInstructor*** tests have passed.  **Steps:**   1. Select the “Delete User” option in the list of options on the home page 2. Input “instructor2” in the username textbox 3. Select “Instructor” 4. Click the “Delete” button | User is successfully deleted and no error is returned | User is successfully deleted and no error is returned |
| **DeleteUserInstructorStudent**  **(Testing UC4-E1)** | **Preconditions:**  The ***CoordinatorLogin*** test has passed.  The ***InstructorLogin*** and ***AddNewUserInstructor*** tests have passed.  The ***DeleteUserInstructor*** test has passed.  **Steps:**   1. Select the “Delete User” option in the list of options on the home page 2. Input “instructor” in the username textbox 3. Select “Student” 4. Click the “Delete” button | User is not deleted and error is returned | Response:  ERROR 400  BAD REQUEST  Error with deleting user |
| **DeleteUserInstructorCoordinator**  **(Testing UC4-E1)** | **Preconditions:**  The ***CoordinatorLogin*** test has passed.  The ***InstructorLogin*** and ***AddNewUserInstructor*** tests have passed.  The ***DeleteUserInstructor*** test has passed  **Steps:**   1. Select the “Delete User” option in the list of options on the home page 2. Input “instructor” in the username textbox 3. Select “Coordinator” 4. Click the “Delete” button | User is not deleted and error is returned | Response:  ERROR 400  BAD REQUEST  Error with deleting user |
| **DeleteUser Student**  **(Testing UC4-S2)** | **Preconditions:**  The ***CoordinatorLogin*** test has passed.  The ***StudentLogin*** and ***AddNewUserStudent*** tests have passed.  **Steps:**   1. Select the “Delete User” option in the list of options on the home page 2. Input “student2” in the username textbox 3. Select “Student” 4. Click the “Delete” button | User is deleted successfully and no error is returned. | User is deleted successfully and no error is returned. |
| **DeleteUser StudentInstructor**  **(Testing UC4-E1)** | **Preconditions:**  The ***CoordinatorLogin*** test has passed.  The ***StudentLogin*** and ***AddNewUserStudent*** tests have passed.  The ***DeleteUserStudent*** test has passed.  **Steps:**   1. Select the “Delete User” option in the list of options on the home page 2. Input “student” in the username textbox 3. Select “Instructor” 4. Click the “Delete” button | User is not deleted and error is returned | Response:  ERROR 400  BAD REQUEST  Error with deleting user |
| **DeleteUser StudentCoordinator**  **(Testing UC4-E1)** | **Preconditions:**  The ***CoordinatorLogin*** test has passed.  The ***StudentLogin*** and ***AddNewUserStudent*** tests have passed.  The ***DeleteUserStudent*** test has passed.  **Steps:**   1. Select the “Delete User” option in the list of options on the home page 2. Input “student” in the username textbox 3. Select “Coordinator” 4. Click the “Delete” button | User is not deleted and error is returned | Response:  ERROR 400  BAD REQUEST  Error with deleting user |
| **DeleteUserCoordinator**  **(Testing UC4-S2)** | **Preconditions:**  The ***CoordinatorLogin*** and ***AddNewUserCoordinator*** tests have passed.  **Steps:**   1. Select the “Delete User” option in the list of options on the home page 2. Input “coordinator2” in the username textbox 3. Select “Coordinator” 4. Click the “Delete” button | The user is successfully deleted and no error is returned. | The user is successfully deleted and no error is returned. |
| **DeleteUserCoordinatorInstructor**  **(Testing UC4-E1)** | **Preconditions:**  The ***CoordinatorLogin*** and ***AddNewUserCoordinator*** tests have passed.  The ***DeleteUserCoordinator*** test has passed.  **Steps:**   1. Select the “Delete User” option in the list of options on the home page 2. Input “coordinator” in the username textbox 3. Select “Instructor” 4. Click the “Delete” button | User is not deleted and error is returned | Response:  ERROR 400  BAD REQUEST  Error with deleting user |
| **DeleteUserCoordinatorStudent**  **(Testing UC4-E1)** | **Preconditions:**  The ***CoordinatorLogin*** and ***AddNewUserCoordinator*** tests have passed.  The ***DeleteUserCoordinator*** test has passed.  **Steps:**   1. Select the “Delete User” option in the list of options on the home page 2. Input “coordinator” in the username textbox 3. Select “Student” 4. Click the “Delete” button | User is not deleted and error is returned | Response:  ERROR 400  BAD REQUEST  Error with deleting user |
| **DeleteUserInstructorByInstructor**  **(Testing UC4-S2)** | **Preconditions:**  The ***InstructorLogin*** and ***AddNewUserInstructor*** tests have passed.  **Steps:**   1. Select the “Delete User” option in the list of options on the home page 2. Input “instructor2” in the username textbox 3. Select “Instructor” 4. Click the “Delete” button | User is not deleted and error is returned saying user role does not have access to this functionality | Response:  ERROR 401  UNAUTHORIZED  Access Denied. |
| **DeleteUserStudentByStudent**  **(Testing UC4-S2)** | **Preconditions:**  The ***StudentLogin*** and ***AddNewUserStudent*** tests have passed.  **Steps:**   1. Select the “Delete User” option in the list of options on the home page 2. Input “student2” in the username textbox 3. Select “Student” 4. Click the “Delete” button | User is not deleted and error is returned saying user role does not have access to this functionality | Response:  ERROR 401  UNAUTHORIZED  Access Denied. |
| **ModifyGradeInstructor**  **(Testing UC5-S1)** | **Preconditions:**  The ***InstructorLogin*** test has passed.  The ***EnrollStudent*** test has passed.  **Steps:**   1. Select the “Modify Grade” option in the list of options on the home page 2. Input “student” in the username textbox 3. Input “CSC492” in course name textbox 4. Input “3.0” in grade textbox 5. Click the “Submit” button | Student’s grade is successfully changed and no error is returned | Student’s grade is successfully changed and no error is returned |
| **ModifyGradeCoordinator**  **(Testing UC5-S1)** | **Preconditions:**  The ***CoordinatorLogin*** test has passed.  The ***EnrollStudent*** test has passed.  **Steps:**   1. Select the “Modify Grade” option in the list of options on the home page 2. Input “student” in the username textbox 3. Input “CSC492” in course name textbox 4. Input “2.0” in grade textbox 5. Click the “Submit” button | Student’s grade is successfully changed and no error is returned | Student’s grade is successfully changed and no error is returned |
| **ModifyGradeStudent**  **(Testing UC5-S1)** | **Preconditions:**  The ***StudentLogin*** test has passed.  The ***EnrollStudent*** test has passed.  **Steps:**   1. Select the “Modify Grade” option in the list of options on the home page 2. Input “student” in the username textbox 3. Input “CSC492” in course name textbox 4. Input “3.0” in grade textbox 5. Click the “Submit” button | The grade is not modified and an error is returned saying user role does not have access to this function | Response:  ERROR 401  UNAUTHORIZED  Access Denied. |
| **ModifyGradeNotEnrolled**  **(Testing UC5-E1)** | **Preconditions:**  The ***InstructorLogin*** test has passed.  The ***EnrollStudent*** test has passed.  The ***AddUserStudent*** test has passed.  **Steps:**   1. Select the “Modify Grade” option in the list of options on the home page 2. Input “student2” in the username textbox 3. Input “CSC492” in course name textbox 4. Input “3.0” in grade textbox 5. Click the “Submit” button | Student grade is not modified and error is returned | Response:  ERROR 400  BAD REQUEST  Error with modifying student's grade |
| **ViewStudentSchedule**  **(Testing UC6-S1)** | **Preconditions:**  The ***StudentLogin*** test has passed.  The ***EnrollStudent*** test has passed.  The ***ModifyGrade*** test has passed.  **Steps:**   1. Select the “Student Schedule” option in the list of options on the home page | User is taken to the view schedule page, which displays the following info:  CSC316, MW, 12:30, 2:15, Instructor, 4  CSC492, TH, 12:50, 2:40, Instructor, 3 | User is taken to the view schedule page, which displays the following info:  CSC316, MW, 12:30, 2:15, Instructor, 4  CSC492, TH, 12:50, 2:40, Instructor, 3 |
| **ViewInstructorSchedule**  **(Testing UC6-S1)** | **Preconditions:**  The ***InstructorLogin*** test has passed.  The ***EnrollStudent*** test has passed.  **Steps:**   1. Select the “Instructor Schedule” option in the list of options on the home page | User is taken to the view schedule page, which displays the following info:  CSC316, MW, 12:30, 2:15  CSC492, TH, 12:50, 2:40 | User is taken to the view schedule page, which displays the following info:  CSC316, MW, 12:30, 2:15  CSC492, TH, 12:50, 2:40 |
| **ViewScheduleRoster**  **(Testing UC6-S3)** | **Preconditions:**  The ***InstructorLogin*** test has passed.  The ***EnrollStudent*** test has passed.  The ***ViewInstructorSchedule*** test has passed.  **Steps:**   1. Select the “Instructor Schedule” option in the list of options on the home page 2. Click “Show students” button next to “CSC492” course | User is shown list of students:  Students - CSC492  Student | User is shown list of students:  Students - CSC492  Student |

The black box tests cover successful flows and alternative flows to ensure the entire end-to-end system is functioning properly. The alternative flows include bad input, as well as authentication errors that would come from SELinux. Every test passes, which shows that the front-end, the container runtime, and the database are all working with each other as intended.

### Unit Testing

#### Unit Testing Information

Unit Testing Tool: Python’s unittest module

Coverage Tool: Python’s coverage.py module

Exempt Units: None

#### Test Coverage Report

Due to Course Manager and the Container Runtime being separate Flask applications, an individual coverage report was generated for each of these components. Figure 12 shows the coverage report generated after running our entire automated test suite with Course Manager and the coverage.py module. This resulted in an overall 96% statement coverage for Course Manager. These unit tests were responsible for testing all of our model classes, database interactions, and REST API endpoints in Course Manager. There was a large focus on making sure the logic of everything was working as expected and returning the correct output and message, both with valid requests and invalid ones.

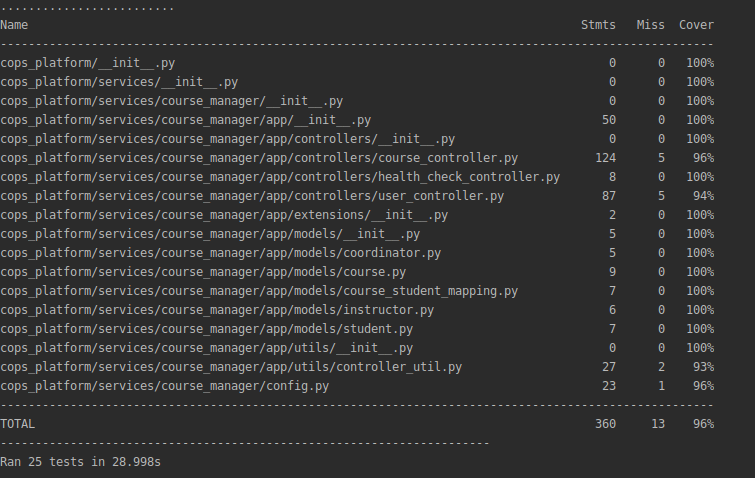


Figure 16: Coverage Report for Course Manager

While testing Course Manager was relatively simple after figuring out how to set up and configure everything for our unit tests; the Container Runtime was a different story. There were a few challenges encountered with testing the Container Runtime. This included the threads (from the HealthCheckThread class) being ignored in the coverage report and exceptions being thrown due to us needing to send a signal to gracefully shut down the Container Runtime application. There are a number of clean up actions the Container Runtime must take care of when it shuts down such as stopping any running containers, terminating all stopped containers, and shutting down any threads that are running. It’s likely there is a simple solution to prevent this exception from occurring. However, due to the time constraints and other pressing things we needed to accomplish in this project, we were not able to successfully fix the inaccurate coverage report or exceptions being thrown by the unit tests.

That said, the coverage report generated for the Container Runtime, which can be viewed in Figure 13, still shows that we did an adequate job of testing all of our code. The biggest improvements could be made to the service\_controller.py and config.py Python files. The service\_controller.py contains the HealthCheckThread class whose code was not being covered at all by the coverage.py module. The unit tests for Container Runtime were responsible for testing that exceptions for bad requests were handled and that the correct bad status code was returned. It aslo tests whether a valid service request resulted in a successful Course Manager application running inside a Docker Container. Finally, these unit tests made sure the health check threads were working and the container was shut down after the user logged out of the service.

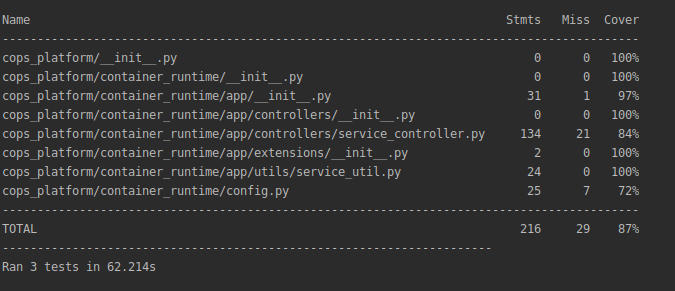


Figure 17: Coverage Report for Container Runtime

## Suggestions for Future Teams

|  |  |
| --- | --- |
| Primary Author | Spencer Yoder |
| Editor | Jeen Shaji |

### Advice for Future Senior Design Teams

#### Fully Read and Understand Our Documentation

The material in this report and the guides is a comprehensive summary of the knowledge we gained over the course of this project, and also a consolidation and contextualization of online documentation for the technologies we used. This documentation is just as much, if not more of a product than our implementation of COps.

The documentation we were able to find online for SELinux, SEPostgreSQL, and how to integrate these with the other parts of our system was largely incomplete, incorrect, and nonexistent. Much of what you will find in this documentation, we could only learn through trial and error. When encountering a problem or unknown, try to locate it in these documents before searching online.

#### Develop Incrementally, Prioritizing Bare Essentials

Initially, we had very lofty goals for our system, but then we came to learn how our small list of functionalities amounts to a series of great accomplishments, because each one was accompanied by a large number of failures and learning experiences. So, we recommend for each iteration picking a small number of features which are core to the system and developing them fully before moving on. Much more time will be spent learning how to work with these technologies than creating deliverables with them.

#### Document Thoroughly and Continuously

Again, since our best resources were in our own written deliverables, we were constantly referring to our documentation in order to develop our system. It’s easy to forget what series of steps are required to get a certain thing to function, and it’s difficult to be constantly communicating between teammates to debug problems with someone else’s components. The more we worked, the more we documented every step of our process because it solved these problems. We found the most success in writing rough, unrefined documentation during our process and later going back and refining this documentation.

### Areas for Improvement

#### Scrapped Features from Project Description

##### Encryption

The original description instructed that our data should be encrypted at rest (in the database) and in transit based on the security label. This should be a relatively easy (and useful) feature to add to the existing system due to the fact that there it should require minimal changes to the infrastructure outside of, perhaps, changing the structure of the database to hold encrypted data

##### IAM

Our project is missing a formal IAM (Identity and Access Management) system, which was originally supposed to authenticate users at multiple levels throughout the system. This could be a potentially difficult feature to add, since it would need to be integrated to every part of the system which needs access to user credentials, including SELinux.

##### Container Orchestration

Our system manages the lifecycle of containers fairly effectively. However, we’re missing a formal container orchestration system like Kubernetes. This could be particularly useful in seamlessly transitioning between containers for users should the need arise.

Out of the features mentioned, this is the one which would be most difficult to add to the system. Primarily because it would involve completely replacing a major component of our system. Furthermore, our container runtime application is entirely responsible for labeling container connections. Figuring out how to do this with a more complicated container orchestration system where you don’t have access to all the code could be incredibly difficult.

##### Logging

Python, Postgres, and SELinux all write logs to the system; however, these logs are not compiled and our system as a whole has no formal logging system. As far as integration, a logging component would be very easy to add to the system. The difficulty would arise from determining logistics: What gets logged? Where do the logs get sent? How do the logs need to be secured?

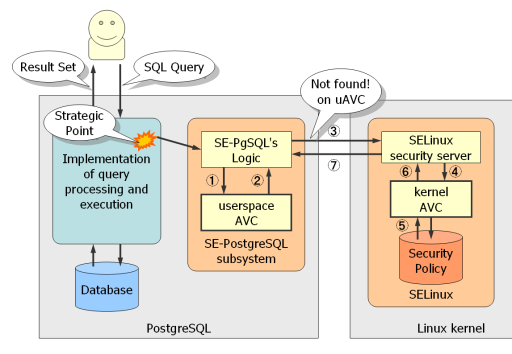
##### Data Consolidation

The end-goal for COps is to replace a system which achieves multi-level data storage by using separate data stores. For this to be possible, COps must consolidate data from these separate stores. Our system doesn’t have this functionality, it generates data. This should be an easy component to add. Our generate\_data.py script shows how to use Postgres queries to label data. What could be difficult is improving on our labeling system to parameterized queries, which we had limited success doing using Python.

### Flaws With the Current Implementation

##### SEPostgres may perform denials in userspace

The uncertainty with this statement reflects that our group is unsure of what makes SEPostgreSQL secure (or not secure). However, we bring it up because SEPostgreSQL seems to communicate with SELinux via system calls. Furthermore, it keeps track of its own access vector cache in userspace:

  
Figure 18: SEPostgreSQL denial control flow

The security of this system needs to be determined, and if it’s insecure, supplemented with a different technology.

##### Docker containers are unlabeled

The state of our current system has certain Docker container accesses performed in the domain: unlabeled\_t. Network operations had to be allowed for unlabeled\_t in order for the containers to communicate with the database. There is a way to label Docker containers on startup; however, we could not get this to work, instead opting for the IP address labeling scheme.

While the IP labeling serves the functional purpose of labeling communication between containers and the Postgres server, being able to label containers would be useful, not only for security (not having to allow unlabeled\_t to communicate over the network), but also for finer granularity in the security policy.

##### Our security policy does not make use of MLS

While we simulate an MLS-like system with our different user levels, the reality is that we only rely on type enforcement for our security policy. The reason for this is we initially did not know how SELinux, especially its labels, functioned in a security policy. We designed a security policy which was originally intended to be a modification of MLS rules, but ended up being easily enforceable through type enforcement.

MLS can be added on to the policy easily, but research should be done as to how it can be used in conjunction with SELinux types which will still be the driving force for the access right separation in the system.

##### Launching the container runtime application after booting up the computer is a multi-step process which can be automated

In order to launch the container runtime, environment variables must be set for flask, Postgres must be restarted, and the localhost IP must be labeled via netlabelctl. These are easily-automated functions, which were not automated because we prioritized them below project completion.

For Postgres, it’s unknown currently why we had to restart it to run the container runtime application. The Postgres configuration file was changed to allow connections from Docker containers, but this change does not seem to apply until a restart of the application.