### Connected Operations (COps) Platform

# Interim Progress Report 1

**Final Requirements, Design, Implementation/Testing & Installation/Delivery**

**Skyward Federal**

**CSC 492 - 32:**

Jonathan Balliet

Caleb Boswell

Daniel Mills

Jeen Shaji

Spencer Yoder

**North Carolina State University**

**Department of Computer Science**

**March 6, 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. The final product will be named the Connected Operations (COps) Platform. By using a Postgres database in tandem with SELinux labels, we will implement 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.

At the time of this report, our team is finishing iteration 1. Iteration 1 has consisted of configuring a Postgres database to work with SELinux on Skyward’s remote development computers. Additionally, iteration 1 has involved developing an API which is able to retrieve data from the database, and will either succeed or fail depending on the labels associated with the user and data. Part of developing this API has been creating a schema that can showcase the security features of the database, which in our case is a course management system. This system will be able to demonstrate our database security by having different user and data types, each with differing SELinux security contexts.

Our next iteration, iteration 2, involves implementing the remaining APIs after the one that was implemented in iteration 1. These APIs will encompass the functionality of writing new data, editing existing data, and reading reports from the system. With the groundwork for a single API laid out from iteration 1, it should be much easier to continue implementing additional APIs. Additionally, during iteration 2 we will finalize the database configuration, and update it to be able to handle the new API calls that have been developed.

## 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 COps.

**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 regular employee does not need access to income details of other employees. If they get access to this, they are technically invading the privacy of their coworkers.

Our particular project relates to our sponsor’s aim to provide multi-level security solutions to their clients as well as their aim to modernize software systems for their clients. Skyward Federal’s particular motivation for this project was to get a proof-of-concept for a platform specifically meant for government clients with outdated data storage systems.

### Project Goals & Benefits

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

Goals:

* Create a REST API using python which communicates with a Postgres database. Each REST service will be used to authorize a user, store or retrieve data.
* Enforce data security at the operating system level with 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 |
| Editor | Jonathan Balliet |

We intend to follow the Agile methodology as laid out by NC State, 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 which 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 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.

### Challenges

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

Our initial challenge was determining the scope of our project. Skyward Federal had plenty of great and detailed designs and requirements for this project, very few of which apply at the current stage of development. This is due both to the fact that we had to create a concrete plan from a set of abstract plans and the fact that Skyward Federal has been modifying the scope of this project internally. We managed to arrive on a definite course of action by communicating with each other inside and outside of weekly meetings.

Since the scope has been defined, the primary nature of our challenges has stemmed from the need for us to utilize esoteric and unfamiliar technologies. SELinux, in particular, has proven to be an abundant source of frustration due to its sparse and conflicting documentation. As our contact, Danny, pointed out, SELinux and its auxiliary technologies were created and documented; however, a lack of consumer interest has led to a lack of an incentive to update this documentation to reflect changing features. As a result, our progress using SELinux has been incredibly slow-going.

## Requirements

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

### 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 safe. On authentication, users can only access the data they have permission for depending on their role in the company and the device they are using. Every user is assigned a Security Label which is comprised of both a range of security levels and a range of categories. The combination of levels and categories make up the user’s security context, which is what determines how they are able to access data in the system.

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 also 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.

### Glossary

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 receive data from an arbitrary number of external sources.

FR2. The system shall label any data received with an appropriate security context.

FR3. The system shall store this data in a database.

FR4. The system shall be interfaced via an API endpoint.

FR5. The system shall authenticate the user every time they access an individual API.

FR6. Users shall be able to make requests to read data, write data, and view reports.

FR7. The system shall execute or deny requests based on the user’s security context.

Data Storage:

FR8. Data elements shall be stored in the database alongside their SELinux labels

FR9. The user shall only be allowed to access data they have the privilege to access and shall be restricted from accessing any other data.

FR10. Users shall be able to submit data for storage they have access to based on their security context.

Container Runtime:

FR11. The container shall provide computing resources for RESTful services.

FR12. This component shall start up a container with the enforced security context of the user for the requested service.

FR13. The service running inside the container must gracefully handle ‘Access Denied’ errors when attempting to retrieve data from the Data Storage component.

FR14. This container shall be able to be activated and deactivated at arbitrary times to serve data to users with the proper security context.

Logging:

FR15. Components of this system shall log all transactions that occur.

Encryption:

FR16. The system shall encrypt the data based on its security context.

### 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 to enable Multi-Level Security (MLS) with the Data Storage component.

C3. PostgreSQL shall be used as the database for our Data Storage component due to its integration with SELinux.

C4. The PostgreSQL database should use SCRAM-SHA-256 for password authentication.

C5. Each RESTful service shall be given login credentials to connect to the SEPostgreSQL database within the Data Storage component.

### Mock System

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 their course rosters and student grades
* **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]. 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: Course Modification

3.1 Preconditions

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

3.2 Main flow

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

3.3 Subflows

* [S1] Coordinators add courses by specifying a course name, day, time range, and instructor [E1].
* [S2] Coordinators can select a course to remove from a list of existing courses.
* [S3] Coordinators can add a student to a course after it has been created by entering the student’s username [E2] [E3].

3.4 Alternate flows

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

3.5 Logging

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Transaction code | Description | Logged in username | Secondary user | Transaction type |
| 300 | Successful add course | coordinator | N/A | Add |
| 301 | Successful delete course | coordinator | N/A | Delete |
| 302 | Successful add student | coordinator | student | Add |
| 303 | Failed add student | coordinator | N/A | Other |

3.6 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 \_ |

#### UC4: User Modification

4.1 Preconditions

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

4.2 Main flow

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

4.3 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.

4.4 Alternate flows

* [E1] Coordinators cannot add users with duplicate usernames.

4.5 Logging

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Transaction code | Description | Logged in username | Secondary user | Transaction type |
| 400 | Successful add user | coordinator | user | Add |
| 401 | Successful delete user | coordinator | user | Delete |
| 402 | Failed add user | coordinator | user | Add |

4.6 Data Format

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

#### UC5: Grade/ Roster 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) (UC3)

5.2 Main flow

Instructors can select a student from any course and modify their grade [S1]. Instructors can select a course and remove a certain student from their roster [S2].

5.3 Subflows

* [S1] Grades are stored for each student-course pair as a floating-point number between 0 and 4 [E1].
* [S2] Instructor can see a roster for each course he is assigned. He can select the option to delete/ remove a student by their username [E2].

5.4 Alternate flows

* [E1] It could be empty if none of the courses have been graded yet.
* [E2] The specified username is not on the roster.

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)

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 be shown 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 |
| 400 | Successful see schedule | coordinator/ student | N/A | View |
| 401 | Successful see roster | coordinator | N/A | View |
| 402 | Successful see grade | student | N/A | View |

#### UC7: View access logs

7.1 Preconditions

A user is a registered user of the Course Manager system. The user has authenticated himself or herself into the system.

7.2 Main Flow

A user can add data or request data. Every transaction will be logged so the coordinator can track any illegal access.

7.5 Logging

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Transaction code | Description | Logged in username | Secondary user | Transaction type |
| 700 | Access logs viewed | coordinator | 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 | Security-Enhanced Linux (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 | ✅ |
| Jenkins | Jenkins is used for CI/CD. It ensures we pass our required test coverage and that there are no significant errors before our software is built and deployed. We plan to use Jenkins for automating the testing, building, and deploying of our project. | 2.190.2 | 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 |

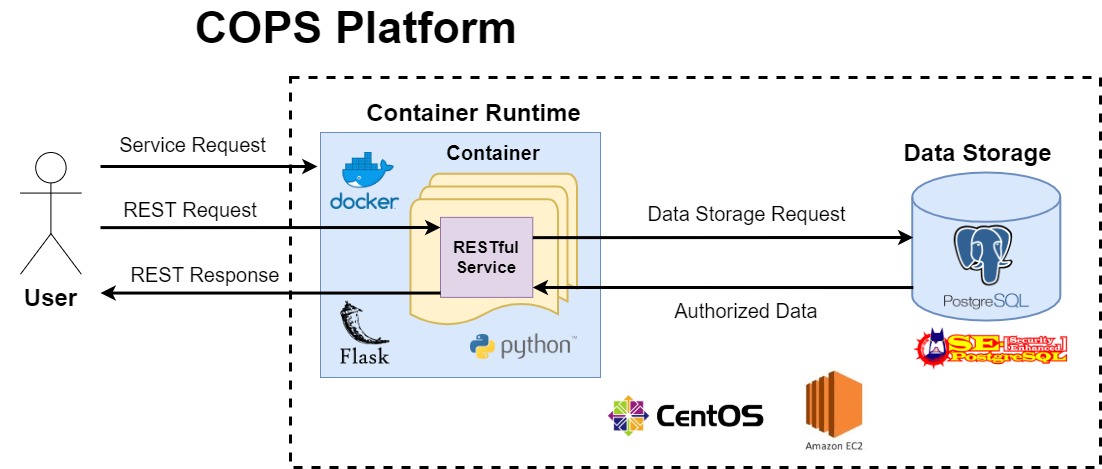
### High Level Design

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 table that makes use of Security Enhanced Linux (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 [FR8].

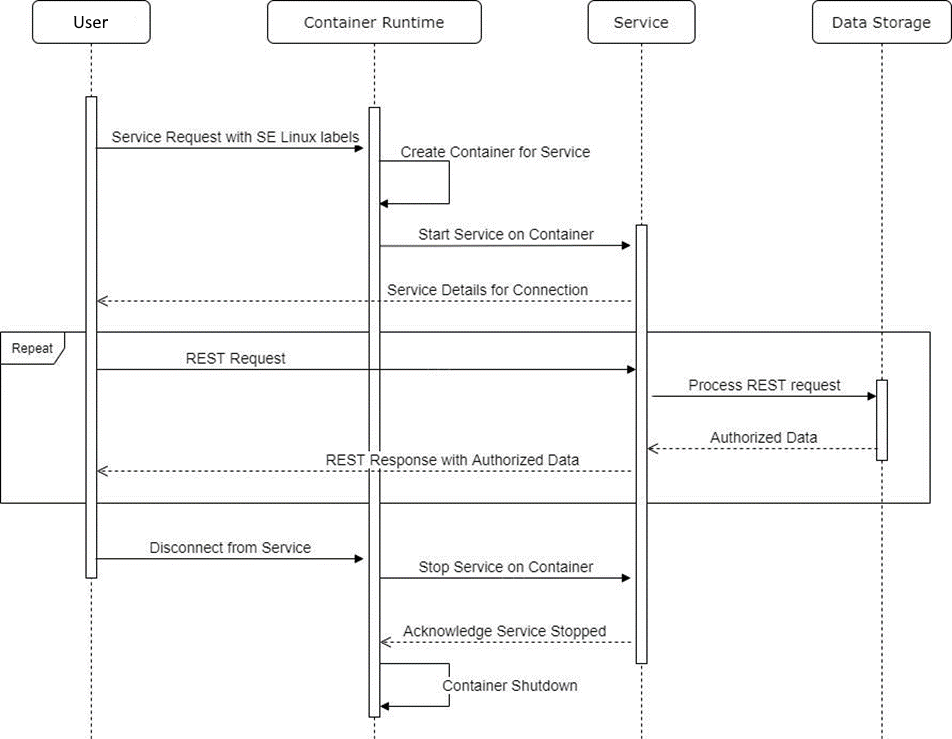
The Container Runtime creates a Docker Container after receiving a service request from a user [FR11]. Docker Containers were chosen as the technology to run our services due to their ability to run in a specific OS image but in such a way that is both dynamic and light-weight. In comparison to virtual 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. In the confines of this project, we are planning to support a single user connection within this Container Runtime component. As a stretch goal, our system will support multiple simultaneous connections of users using Kubernetes. Kubernetes is required for container orchestration in order for the system to dynamically startup, shutdown, and monitor multiple running containers. Each of these containers would run 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.

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 authorized SELinux security labels of the user to the Container Runtime. This user has already been authenticated and authorized in the COps Platform via an Identity and Access Management (IAM) system. This IAM component will eventually be integrated into the COps Platform but it is out of scope for the project deliverables this semester. These security labels determine what data the service is authorized and not authorized to access. After receiving this request, 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 SELinux labels sent with the request. 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*. An overview of the relationship of these modules is illustrated in Figure 3. 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. Services are displayed as an interface in Figure 3 because they must contain certain attributes, such as being RESTful and have a corresponding Dockerfile and Docker image in order to successfully run in the COps platform. The Course Manager Module is our concrete implementation of a RESTful service that we are creating for testing the COps platform infrastructure. All transactions are to be logged in this system, which is handled through the LoggingUtil class. The Container Runtime Module will be further designed and implemented in future iterations of this project. For now the design is abstract to convey how this system operates on a lower level.

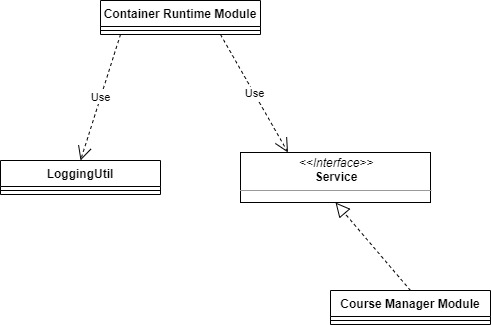


Figure 3: Overview of theLow Level Design Modules for COps Platform

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.

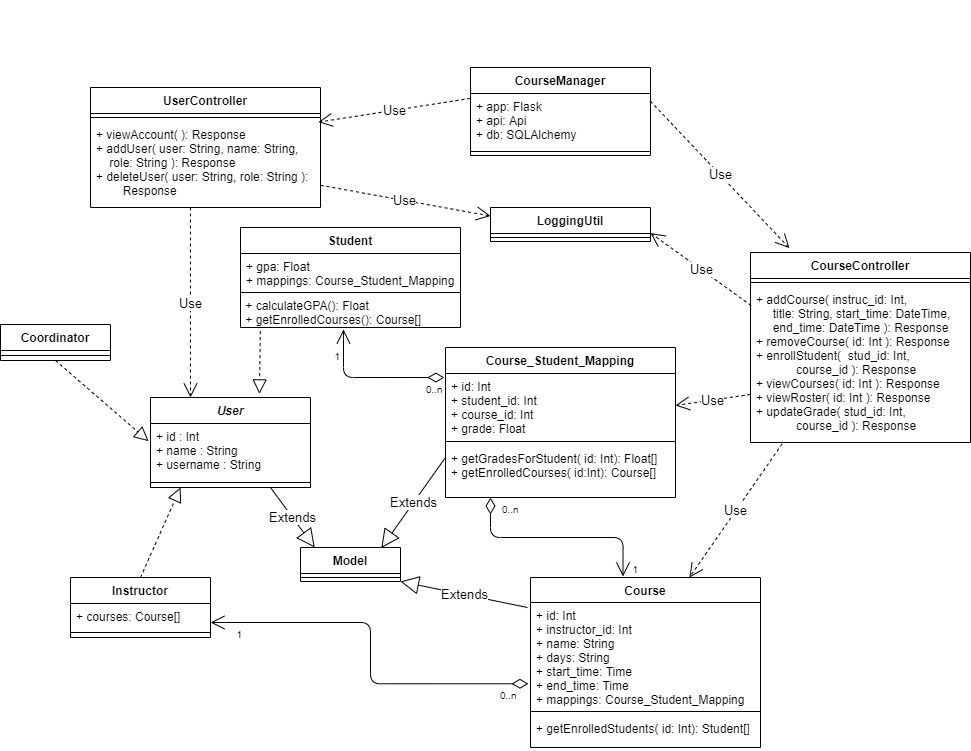
****

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 based on the user’s security levels that it receives in the request. 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}/api/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”
* Creates a container instance for the requested service
  + Each service will have its own Dockerfile and Docker image. These contain the operating system image, along with all the needed dependencies, configurations, and files for running this service.
  + Enforces the passed in SELinux labels for the running service in this container.
  + Starts the RESTful service on the container instance.
* Shut downs running container instances
  + The container instance stops a running service when the user has disconnected from the service.
  + 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 response (along with any authorized data) back to the user.

*Responsibilities:*

* Establishes connection with the user to the Data Storage (PostgreSQL database) after startup
  + Connects to the database using statically configured credentials.
  + Sends a response to the user letting them know whether connection has been successfully established.
* 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”
* 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, weekday, start\_time, end\_time, along with an existing instructor's id are sent in the body of the POST request.
  + Example:
    - name=”CSC316”
    - weekday=”MW”
    - start\_time=”11:15”
    - end\_time=”12:30”
    - instruct\_id=”1”
  + 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\_id=”1”
  + 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/course/enroll
  + The student’s username and the course’s id are sent in the body of the POST request.
  + Example:
    - username=”student tester”
    - course\_id=”2”
  + 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/grade
  + The course’s id and the student’s username are sent in the body of the PUT request.
  + Example:
    - course\_id=”8”
    - username=”student tester”
  + 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/student
  + Returns a response with a list of course objects, along with an associated grade field.
  + Example response:

[

name=”CSC316”

weekday=”MW”

start\_time=”11:15”

end\_time=”12:30”

instruct\_id=”1”

grade=4.0,

name=”CSC333”

weekday=”TH”

start\_time=”2:15”

end\_time=”1:30”

instruct\_id=”5”

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/instructor
  + Returns a response with a list of course objects, along with an associated list of student names.
  + Example response:

[

name=”CSC316”

weekday=”MW”

start\_time=”11:15”

end\_time=”12:30”

instruct\_id=”1”

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

name=”CSC492”

weekday=”TH”

start\_time=”9:30”

end\_time=”10:45”

instruct\_id=”1”

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

]

* Viewing all logged transactions for the Course Manager System (UC7)
  + GET request
  + Handles request to view all logged transactions
  + Endpoint:
    - {base\_path}/api/logs
  + Returns a response with a list of log objects.
  + Example response:
    - code=”401”
    - description=”Successful delete user”
    - logged\_in\_user=”coordinator”
    - secondary\_user=”user”
    - transaction\_type=”Delete”
    - timestamp=” 2020-03-06 20:11:53.0”
* 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 our mock Course Manager system. 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

There is a many-to-many mapping between students and courses through course\_student\_mappings and a one-to-many mapping between courses and instructors.

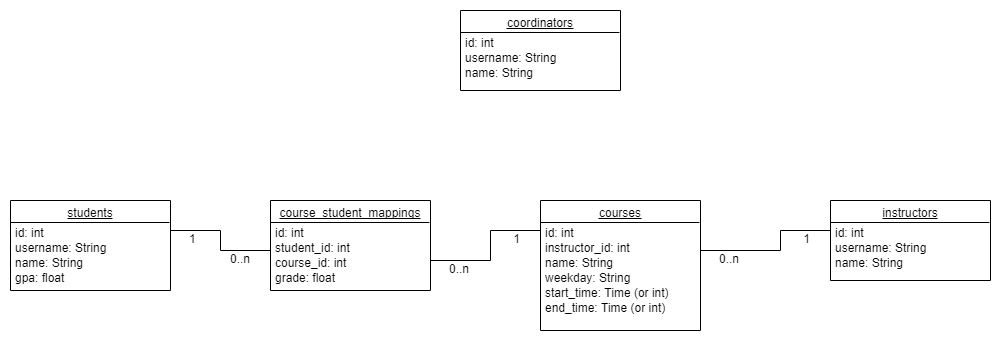
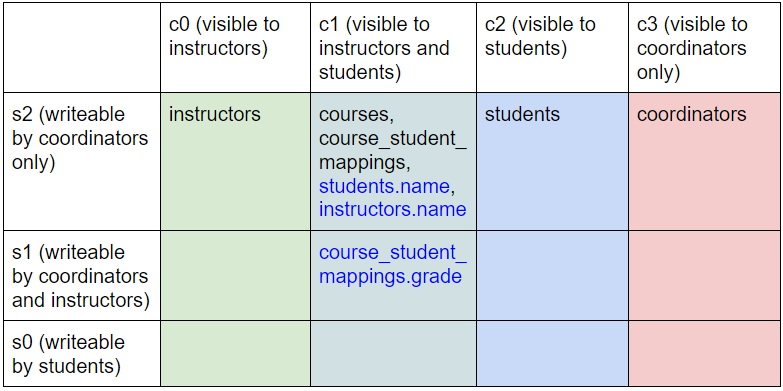


Figure 5: Database Schema for Course Manager Database

Our PostgreSQL database will be configured with SELinux labels protecting columns and tables. The data will be secured in a specific security policy that is described in Figure 6. In this security policy, users get a combination security level (sn) and a category range (cn.cm) called a security context. Users can write to data *at or below* their security level and read data at any level within their category range.

The following security contexts result for each user type:

* Coordinators: *s2:c0.c3*
* Instructors: *s1:c0.c1*
* Students: *s0:c1.c2*

Figure 6: SELinux Integrated Security Policy for PostgreSQL Database

## Implementation

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

#### Current Implementation Status

At the time of this report, our team has completed iteration 0, and are finishing the implementation of iteration 1. The details of each iteration are laid out below. For the data storage component, we have been able to install and configure SEPostgres on Skyward Federal’s remote development computers. After the first successful installation, we created shell scripts that automate this process, so that we can install it on any CentOS 7 computer. These scripts not only made it quicker to install the software on the computers for the first time, but also allow us to arbitrarily launch new AWS instances knowing that we can set it up exactly the same way as the others.

For the container runtime component, we have been developing the Course Manager system in Python, using the PyCharm IDE. This has involved setting up the model objects for each user and data type, creating a mock database using SQLite, and laying the groundwork for an API using Flask. The project directory structure and notable source code components can be found later in this section.

#### 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, FR6-FR9, FR11, FR13

##### 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, FR10, FR15

##### Iteration 3

* Dockerize the running service in a container
* Stretch goals:
  + Implement a simple front-end for demonstration purposes
* Requirements:
  + FR12, FR14, FR16

##### Iteration 4

* Reserved as a catch-up iteration if needed
* Encrypt and decrypt data at REST

#### 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 which 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 SEPostgres configuration in place, a student is only able to view their own grades.

##### Integrity

The integrity of our system is achieved by having security at the database label. When data is stored in our database, it is necessarily stored along-side an SELinux security label. When a user attempts to access this data, the label associated with the data must fall within the user’s security context. Otherwise, the request is denied and the data is never accessed. By having the security so closely tied with the database, integrity can be ensured. We will be able to demonstrate this in our Course Manager system. We are intentionally allowing users to access API calls which will attempt to access unauthorized data. Even though a student can make the API call, the system will still deny the request due to the database-level security. This is only for demonstration of data security, and an actual system would probably not do this.

##### 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. A system which uses the COps platform will be able to use the API calls to implement a front-end application that is user-friendly. As seen in the iterations, one of our stretch-goals is to make a graphical user interface to demonstrate the availability in our system.

##### Identification & Authentication

Since our Course Manager is primarily meant to demonstrate the database security of the system, our implementation will not include an authentication component. However, this was part of the initial design given to us by our sponsor. The design had an Identification and Access Management (IdAM) component. With our implementation acting as a groundwork for a fully fleshed-out multi-level security system, a future group would likely include an external 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

Privacy in the system will come from data encryption at REST. This is listed as a stretch goal in iteration 3, however we fully expect to be able to implement this. Encrypting the data helps to ensure that if someone was able to breach the database without going through the API, they are still not able to learn any private information from it. Our Course Manager system would be able to demonstrate this through the coordinator user type. Coordinators are the only type of user that can directly access the database. If a coordinator were to try to view a student’s grades directly through the database, they would just be able to view the encrypted data. This helps ensure students’ privacy.

#### Project Structure

Figure 7 below shows the directory structure of our project in PyCharm. At the moment, our team’s main focus is on the code in the services/course\_manager folder. The “controllers” directory is for the implementation of the API endpoints. The source files use the flask library to implement REST API functionality for users. The “db” directory is for any code related to database storage. At the moment, we are using the SQLite library to use a mock database. However, we will eventually have the code talk to the actual Postgres database. The “models” directory contains all of the model objects for each type of user and data. There are separate files containing classes for coordinators, courses, instructors, and students. Additionally, there is a model which maps out the relationship between students and courses. The “tests” directory contains all unit tests for the source code, each one using the unittest Python library.

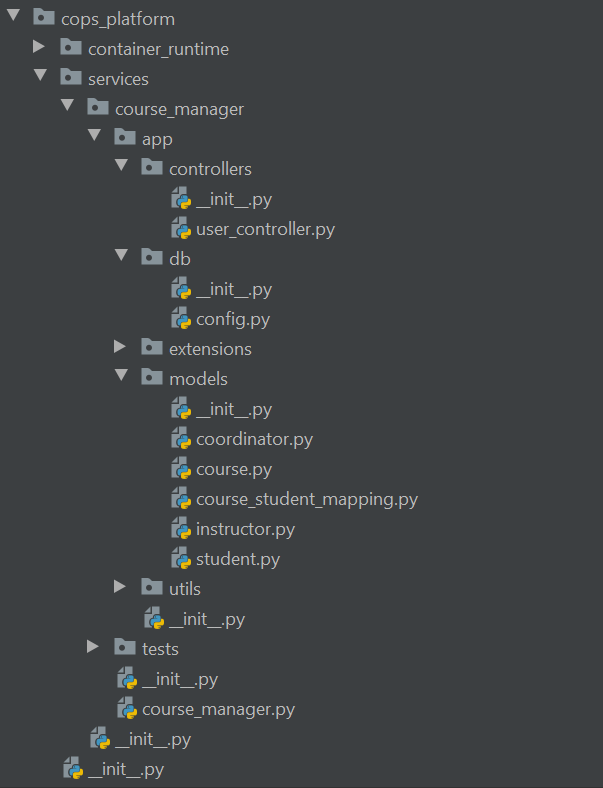
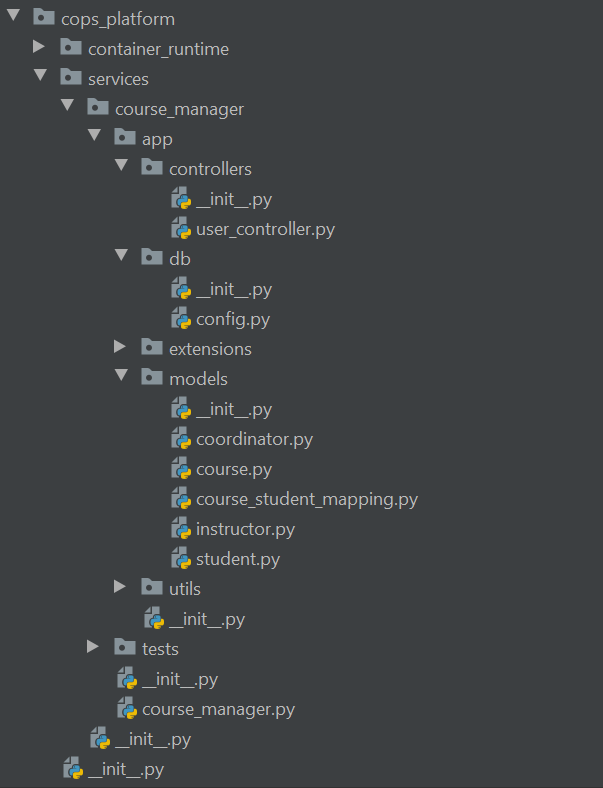


Figure 7: Project directory structure

## Test Plan & Results

|  |  |
| --- | --- |
| Primary Author | Jonathan Balliet |

#### Overview:

##### Blackbox testing tool:

We are planning to do system tests using curl commands due to currently having no front-end interface in place. As a stretch goal we will be implementing a front-interface for the Course Manager System. If achieved, Selenium will be used for system testing instead.

##### Whitebox testing tool:

We will use the unittest module in Python to do all of our unit and integration testing.

##### Expected Coverage:

We are anticipating 70% coverage as recommended by the sponsors.

##### 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, 70% 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.

#### System Test Plan

##### *Testing Overview*:

The system tests require a CentOs7 operating system image enforced with the Security Enhanced Linux (SELinux) module. A SELinux integrated PostgreSQL database must be installed on the same CentOs7 image with the SELinux security policy defined in Figure 6. The COPS platform project must also be installed on this running operating system image. Our GitHub repository is available for cloning at the following url: <https://github.ncsu.edu/engr-csc-sdc/2020SpringTeam32.git>

Currently our System Test Plan only includes the Use Cases related to user functionality in the requirements for our mock Course Manager System. These tests cover backend functionality for UC1, 2, and 4. System tests for UC 3, 5, 6, and 7 will be completed in iteration 2. Future iterations will include system tests for our still to be implemented Container Runtime component. Scripts for installing SEPostgreSQL on CentOs7 and for creating our database security policy will eventually be added to our project to automate these preconditions.

The following Linux users exist in the running CentOs7 image:

* User 1:
  + Username: *student*
  + Security context for running course\_manager.py: *s0.c1,c2*
* User 2:
  + Username: *instructor*
  + Security context for running course\_manager.py: *s1.c0,c1*
* User 3:
  + Username: *coordinator*
  + Security context for running course\_manager.py: *s2.c0,c2*

*Resetting the database and adding needed test data:*

Run the *db\_generate.py* file located at the path *cops\_platform/services/course\_manager/tests/db* in the COPS Platform project using the following command:

*python db\_generate.py*

This python script resets the state of the database and adds all of the needed users and data in the database that is required for testing.

*Running the Course Manager Service:*

Run the *course\_manager.py* file located at the path *cops\_platform/services/course\_manager* in the COPS Platform project using the following command:

*python course\_manager.py <role>*

The <role> command line argument is a string representing the user’s role (or security levels). This is used to know which database table to look at for verifying user information for login and account details. (***NOTE:*** This will eventually be automated by the Container Runtime when the security context of the user from the service request is passed in as this argument.)

Pass in the corresponding value in the command line argument based on the current user being tested:

***Tester #1:*** *instructor*

***Tester#2:***  *student*

***Tester#3:*** *coordinator*

Once the *course\_manager.py* Flask application has been started it will be running on localhost at port 5000 (These are the default settings for Flask).

***Test Data Summary*:**

The following users exist in the Course Manager system:

***Tester #1***

**Username:** instructor

**Name:** Instructor

**Attributes:**

**sensitivity\_range:** “s1”

**categories:** “c0,c1”

***Tester #2***

**Username:** student

**Name:** Student

**Attributes:**

**sensitivity\_range:** “s0”

**categories:** “c1,c2”

***Tester #3***

**Username:** coordinator

**Name:** Coordinator

**Attributes:**

**sensitivity\_range:** “s1,s2”

**categories:** “c3”

##### Acceptance Test Plan

|  |  |  |  |
| --- | --- | --- | --- |
| **Test ID** | **Description** | **Expected Results** | **Actual Results** |
| **InstructorLogin**  **(Testing UC1)** | **Preconditions:**  Logged in as the *instructor* Linux user.  Course Manager Service is running with the *instructor* argument passed in.  **Steps:**   1. Send POST request to [*http://127.0.0.1:5000/api/login*](http://127.0.0.1:5000/api/login) containing the following data in the body of the request:   username=  ”instructor”   1. Await response | Response with an OK status and session cookie in the header is returned. |  |
| **StudentLogin**  **(Testing UC1)** | **Preconditions:**  Logged in as the *student* Linux user.  Course Manager Service is running with the *student* argument passed in.  **Steps:**   1. Send POST request to [*http://127.0.0.1:5000/api/login*](http://127.0.0.1:5000/api/login) containing the following data in the body of the request:   username=  ”student”   1. Await response | Response with an OK status and session cookie in the header is returned. |  |
| **CoordinatorLogin**  **(Testing UC1)** | **Preconditions:**  Logged in as the *coordinator* Linux user.  Course Manager Service is running with the *coordinator* argument passed in.  **Steps:**   1. Send POST request to [*http://127.0.0.1:5000/api/login*](http://127.0.0.1:5000/api/login) containing the following data in the body of the request:   username=”coordinator”   1. Await response | Response with an OK status and session cookie in the header is returned. |  |
| **ViewAccountDetailsInstructor**  **(Testing UC2)** | **Preconditions:**  Logged in as the *instructor* Linux user.  Course Manager Service is running with the *instructor* argument passed in.  The ***InstructorLogin*** test has passed.  **Steps:**   1. Send GET request to *http://127.0.0.1:5000/api/user*. 2. Await response | Response with an OK status and following data is returned:  username: instructor  name: Instructor |  |
| **ViewAccountDetailsStudent**  **(Testing UC2)** | **Preconditions:**  Logged in as the *student* Linux user.  Course Manager Service is running with the *student* argument passed in.  The ***StudentLogin*** test has passed.  **Steps:**   1. Send GET request to *http://127.0.0.1:5000/api/user*. 2. Await response | Response with an OK status and following data is returned:  username: student  name: Student |  |
| **ViewAccountDetailsCoordinator**  **(Testing UC2)** | **Preconditions:**  Logged in as the *coordinator* Linux user.  Course Manager Service is running with the *coordinator* argument passed in.  The ***CoordinatorLogin*** test has passed.  **Steps:**   1. Send GET request to *http://127.0.0.1:5000/api/user*. 2. Await response | Response with an OK status and following data is returned:  username: coordinator  name: Coordinator |  |
| **AddNewUserInstructor**  **(Testing UC4-S1)** | **Preconditions:**  Logged in as the *instructor* Linux user.  Course Manager Service is running with the *instructor* argument passed in.  The ***InstructorLogin*** test has passed.  **Steps:**   1. Send POST request to [*http://127.0.0.1:5000/api/user*](http://127.0.0.1:5000/api/user) with the following data in the body:   username=  ”tester”  name=”Test”   1. Await response | Response with an ACCESS DENIED status is returned |  |
| **AddNewUserStudent**  **(Testing UC4-S1)** | **Preconditions:**  Logged in as the *student* Linux user.  Course Manager Service is running with the *student* argument passed in.  The ***StudentLogin*** test has passed.  **Steps:**   1. Send POST request to [*http://127.0.0.1:5000/api/user*](http://127.0.0.1:5000/api/user) with the following data in the body:   username=  ”tester”  name=  ”Test”   1. Await response | Response with an ACCESS DENIED status is returned |  |
| **AddNewUserCoordinator**  **(Testing UC4-S1)** | **Preconditions:**  Logged in as the *coordinator* Linux user.  Course Manager Service is running with the *coordinator* argument passed in.  The ***CoordinatorLogin*** test has passed.  **Steps:**   1. Send POST request to [*http://127.0.0.1:5000/api/user*](http://127.0.0.1:5000/api/user) with the following data in the body:   username=  ”tester”  name=  ”Test”   1. Await response | Response with an OK status is returned |  |
| **DeleteUserInstructor**  **(Testing UC4-S2)** | **Preconditions:**  Logged in as the *instructor* Linux user.  Course Manager Service is running with the *instructor* argument passed in.  The ***InstructorLogin*** and ***AddNewUserCoordinator*** tests have passed.  **Steps:**   1. Send DELETE request to [*http://127.0.0.1:5000/api/user*](http://127.0.0.1:5000/api/user) with the following data in the body:   username=  ”tester”   1. Await response | Response with an ACCESS DENIED status is returned |  |
| **DeleteUser Student**  **(Testing UC4-S2)** | **Preconditions:**  Logged in as the *student* Linux user.  Course Manager Service is running with the *student* argument passed in.  The ***StudentLogin*** and ***AddNewUserCoordinator*** tests have passed.  **Steps:**   1. Send DELETE request to [*http://127.0.0.1:5000/api/user*](http://127.0.0.1:5000/api/user) with the following data in the body:   username=  ”tester”   1. Await response | Response with an ACCESS DENIED status is returned |  |
| **DeleteUserCoordinator**  **(Testing UC4-S2)** | **Preconditions:**  Logged in as the *coordinator* Linux user.  Course Manager Service is running with the *coordinator* argument passed in.  The ***CoordinatorLogin*** and ***AddNewUserCoordinator*** tests have passed.  **Steps:**   1. Send DELETE request to [*http://127.0.0.1:5000/api/user*](http://127.0.0.1:5000/api/user) with the following data in the body:   username=  ”tester”   1. Await response | Response with a OK status is returned |  |

#### Unit Testing Information

Unit Testing Tool: unittest

Coverage Tool: coverage.py

Exempt Units: None

#### Test Coverage Report

The coverage report for the project has yet to be compiled due to issues we have encountered in working with the coverage.py tool.

## Task Plan

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

##### Project Scope:

The focus of our project is that we are working on the REST API and the containers for storing and retrieving the data. We will be focusing on making sure the processes are secure when users are authorized and when retrieving data from the containers so that they are only able to retrieve data within their security level.

##### Task Plan Table:

|  |  |  |  |
| --- | --- | --- | --- |
| **Team 32 Task Plan** | | | |
| **Item** | **Owner** | **Due Date** | **Status** |
| Research | | 01/31/2020 |  |
| SELinux | Spencer |  | Complete |
| Docker | Jonathan |  | Complete |
| PostgreSQL | Daniel |  | Complete |
| Zuul | Jeen |  | Complete |
| Firecracker | Caleb |  | Complete |
| Requirements Draft | | 02/14/2020 |  |
| Functional Requirements | Jonathan, Daniel, Spencer |  | Complete |
| Use cases | Jeen, Daniel |  | Complete |
| Design Draft | | 02/21/2020 |  |
| Database relational diagram | Spencer |  | Complete |
| UML Class diagram | Jonathan |  | Complete |
| Iteration 1: Implement an API request | | 03/06/2020 |  |
| Create service to facilitate a single GET REST call | Jonathan |  | Complete |
| Create model classes for database objects | Jeen, Caleb |  | Complete |
| Create unit tests for all model classes | Jeen, Caleb |  | Complete |
| Install and initialize SEPostgres | Spencer, Daniel |  | Complete |
| Create/approximate a security policy for CourseManager according to data separation design | Spencer, Daniel |  | In progress |
| Integrate CourseManager with SEPostgres | Spencer, Daniel, Jonathan |  | In progress |
| Iteration 2: Complete System testing | | 03/20/2020 | Not started |
| Perform System Testing for a single API call | Spencer, Daniel, Jonathan |  |  |
| Implement other API calls | Spencer, Daniel, Jonathan |  |  |
| Test all the implemented API requests | Jeen, Jonathan |  |  |
| Check coverage and reach minimum 80% | Jeen, Caleb |  |  |
| Iteration 3: Dockerize the system | | 04/03/2020 | Not Started |
| Utilize docker containers to run the system | Jonathan |  |  |
| Simple front end for P & P presentation | Jeen, Spencer |  |  |
| Test front end interaction | Daniel, Caleb |  |  |
| Iteration 4: Encryption of data | | 04/17/2020 |  |
| Research and learn about encryption | Everyone |  |  |
| Implement Encryption to the sponsor’s satisfaction | Everyone |  |  |

##### Team Member Contact Information

|  |  |  |
| --- | --- | --- |
| Spencer Yoder | [smyoder@ncsu.edu](mailto:smyoder@ncsu.edu) | 919-818-7079 |
| Jeen Shaji | [jshaji@ncsu.edu](mailto:jshaji@ncsu.edu) | 747-229-8907 |
| Daniel Mills | [dcmills2@ncsu.edu](mailto:dcmills2@ncsu.edu) | 919-909-4330 |
| Jonathan Balliet | [jballie@ncsu.edu](mailto:jballie@ncsu.edu) | 919-608-3331 |
| Caleb Boswell | [caboswel@ncsu.edu](mailto:caboswel@ncsu.edu) | 336-816-8172 |