Refactoring a Full Stack Web Application to Remove Barriers for Student Developers and to Add Customization for Instructors*

Jack Cook[1], Richard Weiss[1], Jens Mache[2]
[1]The Evergreen State College
{coojac09, weissr}@evergreen.edu
[2]Lewis & Clark College
jmache@lclark.edu

Abstract

This paper describes our experience refactoring EDURange, a full-stack Web application, in order to make it easier for students to do undergraduate research and contribute. As a result, more students were able to contribute to this open source project. In addition, as instructors we wanted to have a simple interface to customize existing exercises and parameterize them so that students could repeat an exercise without it being identical. The main differences were: changing from Ruby on Rails to Python Flask, changing from Virtual Machines to Docker containers, and eliminating dependence on AWS through Terraform. These changes reduced the number of lines of code from 28K to 12K.

1 Introduction

Refactoring [2] a large program is hard and time-consuming. So, why would we do it? In our case, we wanted to make some significant changes to EDURange:

- Make it easy for undergraduates to contribute code,
- Make it easy for instructors to create and add their own exercises,

- Make it more efficient, and
- Make it more portable by removing dependence on AWS.

We made it easier for undergraduates to contribute code by switching to a language they were more familiar with, switching to a simpler Web framework, and reducing the size of the codebase. The original framework was Ruby on Rails and we switched to Python Flask. Ruby is not taught in our curriculum, but Python is. We made it easier for instructors to create exercises by adding a level of automation, switching from Chef to Terraform, and creating a version of the application that runs on a local server as opposed to a Cloud. Using Terraform also increased the portability. Terraform is a more flexible tool and supports several environments. We made it more efficient by switching from using virtual machines (VMs) to Docker containers.

1.1 Background

EDURange [1, 5, 6, 4] is an NSF-funded project that is both a collection of interactive, collaborative cybersecurity exercises and a framework for creating these exercises. It is designed to provide students with an active learning environment focusing on analysis skills rather than the latest tools.[1] EDURange also allows the instructor to observe student interactions, which can help instructors identify student problems. This environment has typically been provided in the form of one or more Amazon Web Services (AWS) virtual machines, which are launched on-demand with the proper configurations for different exercises. For example, there is an exercise Total Recon where students find hosts with specific ports open or services running. In our original version, each host was a separate VM.

The original version of EDURange was developed using Ruby on Rails for the front-end web application in connection with a Postgres database for managing accounts, groups, and exercises. Additionally, the back-end of EDURange is supported by a Redis server for issuing and scheduling commands, such as starting and stopping VMs, or sending emails to users who request a password reset.

One of the design goals for the original version was that it should be easy to use in a demo. We were able to use either pre-made accounts or we could have users sign up at the beginning of a workshop. The one problem that we had was that if users arrived late, then we couldn't add them to the scenario. We have given over 20 demos at conferences and while in the beginning there were some delays of up to 20 minutes, we never had to cancel a demo. It was very successful, but sometimes it became expensive to use in the classroom.

[1] https://edurange.org/about.html

EDURange was initially designed to allow instructors to create and customize their own scenarios. The main feature that enabled this was that scenario creation was scripted. There were template scripts with parameters that would control the complexity of the exercise. That turned out to be difficult to create and maintain using the combination of Chef, YAML and Ruby on Rails. Terraform is allowing us to reintroduce these customization features because of its simplicity and flexibility.

Terraform works with multiple platforms, e.g. AWS, Docker, OpenStack, VSphere, and it is more modular than other virtual infrastructure provisioning frameworks. To get started with Terraform, we create a basic template for a configuration file that includes basic container image and network structure details. From this template, each individual exercise can then add its own required packages, scripts, and the accounts needed for students to play it. This allows complete configuration files to be easily written in a modular fashion with scripts, as opposed to manually writing the configuration for every individual container required for an exercise. There are two layers of automation: the configuration file automates creating the containers, and the template automates creating the configuration file. The templates we need to use are also very concise, Terraform can launch a single Docker container with a configuration file as short as 20 lines.

2 EDURange Structure

There are three main parts to the structure for EDURange: 1) the web framework, 2) the control framework for provisioning the virtual environment and collecting data while the exercises are running, and 3) the database for storing information about scenarios and users.

2.1 Web Framework

In this refactor, we chose to use Python as our language for web development instead of Ruby on Rails. The primary motivation for this choice was that although Ruby on Rails may be a more popular enterprise platform, Python is offered as part of our curriculum, so Python is much more likely to be accessible by future developers. In fact, we have had summer research students who worked on designing scenarios but never managed to learn Ruby on Rails. Additionally, we wanted a more lightweight framework than Rails, allowing developers to more easily stay organized and make small, modular changes to the application.

To kickstart the development of the new Flask application, the cookiecutter-flask[2] tool was used to setup the directory structure of the project, as well as provide basic initial configuration for several Flask extensions. WebPack is integrated to compress Javascript and CSS files, which significantly improves the performance of the website.

At a high level, the refactor web application is a REST API that processes user requests in three stages. First the application determines what page is being requested and by what method, also known as the route. Then, browser session variables are checked against the database to validate whether or not a user is allowed to access that particular route. Lastly, if the check passes, the data from any form submitted by the request is saved, and the page is served. This design choice was motivated by the desire to create more explicit separation between different utilities of the web application, and requiring the ability to make modular and incremental changes. For example, if a developer wanted to add a new page to the website, it would be a three step process:

1. Define the route to your page, edurange.org/new.html for example.
2. Create that HTML file, which only requires two lines to inherit the site-wide navigation and style options.
3. Create any forms or tables as needed for the functions of the new page.

This workflow and separation of different utilities greatly simplifies adding new features and pages to the website. Comparatively, adding a new page on the Ruby on Rails EDURange platform would require the definition of the new route from the Rails command line, followed by the creation and modification of several Ruby and HAML files, as well as individual modals for all forms, tables, and popup dialog boxes.

In summary, Flask is in a preferable framework because it is more lightweight and more easily extensible. One of the most important extensions to our application that Flask allows us to preserve is the Redis service, which is integral to exercise management.

2.2 Control Framework: Scenario Management

One of the advantages of containers is that it would bring down cost if running on AWS, and it would make it feasible to run scenarios on a local server. Amazon also imposes limits on the number of virtual clouds and machines can be in use in a given region, which can limit the number of exercises that can be running simultaneously. Some scenarios require 10 virtual environments. Lastly, we also wanted instructors to be able to run EDURange on a local network not connected to the Internet. [3]

Using containers means:

[2] https://github.com/cookiecutter-flask/cookiecutter-flask

1. Only one machine would be required; the same machine that hosts the EDURange application would host the containers that encapsulate different scenario environments.
2. While this host machine would need significantly improved performance over the current host machine, the EDURange application could be more easily distributed and hosted anywhere, without dependence on AWS.

Because Terraform is capable of configuring virtual subnets to connect containers, there is no need to create virtual clouds and new IP addresses through AWS, instead scenarios are hosted on an open port of the host server. This does mean that everything is running on one machine, which would require significant boosts to computational power and network bandwidth. However, containers are lightweight enough that the capacity for running scenarios could be nearly limitless, while the costs would remain static for only the price of one machine. Alternatively, instructors could put EDURange on one of their institution's servers, and access it locally without concerning themselves with AWS fees.

2.3 Database Schema Improvements

Another major improvement was refactoring the Postgresql Database, to remove obsolete tables and simplify the relationships between them. Here is a comparison of the tables contained in the legacy system, versus the refactor:

Legacy Database

Answers	Bash-histories	Clouds	Groups
Instance-groups	Instance-roles	Instances	Players
Questions	Recipes	Role-recipes	Roles
Scenarios	Student-group-users	Student-groups	Subnets
Users			

Refactor Database

Answers	Bash-histories	Group-users	Groups
Questions	Scenario-users	Scenarios	Users

Some tables that stored internal metadata in the legacy system have been omitted for brevity, but overall we've managed to reduce the number of tables required for the minimal operation of the platform from 24 to 8. Primarily, all of the tables related to Roles and Recipes were only required for Chef, and the monitoring of individual instances and subnets has been transferred to Terraform rather than relying on the database. We've also simplified the way groups and scenarios are managed, since having tables for "Users, Players, Student-Groups, Student-Group-Users and Instance-Groups" is needlessly confusing for new developers. It is extremely difficult to determine the importance or roles of those tables in the legacy database, even when looking deeper than just the table names.

2.3.1 Scheduling: Redis with Celery

As previously mentioned, Redis has been the primary service used for enabling EDURange to run console commands. This includes the two core Terraform commands "terraform apply" and "terraform destroy", which respectively allocate and free resources. For the purposes of security and organization, these commands are handled by a scheduling worker that runs within the Redis broker. In the Rails version of EDURange this scheduler is Sidekiq, and for the Python refactor the scheduler is a service called Celery. These services perform the same utility, and that is to make sure that requested tasks (such as starting a scenario, or collecting logs) are monitored, executed, and don't cause conflicts. For example, it should prohibit the modification of a running scenario or the collection of logs from a scenario that's been deleted.

What makes Celery perfect for this piece of the project is the ease with which developers can specify new tasks. Essentially, each task is just a single python function that is defined in central "tasks.py" file. These tasks can range from being extremely complex scripts that write out Terraform configuration files for scenarios (a work in progress), or very simple like downloading logs from a running container.

One simple task that we are able to re-use for many different utilities in EDURange is our *send_async_email* task, which is only 7 lines long. Celery simply packages *email_data* from a web form into a Flask Mail *Message* object, and sends it off. Because of how generally this function is designed, we can use it for things like password resetting, scenario duration warnings, or any other utility that requires an email to be sent. These Celery tasks can be written to support any arbitrary utilities, and the number of concurrent running tasks can be scaled easily by running parallel workers, though that shouldn't ever be necessary.

2.4 Deployment

At this point it may seem that all of these services would take a great deal of effort to install and configure to work properly, but a major priority of this project has been to make the EDURange platform more easily accessible, and even possible for instructors to easily run it on their own machines. In service of this goal, we've implemented two options for running the EDURange platform.

The first is a wrapper using Node Package Manager which, after a one-time installation of Node, can gather and update all the requirements of the application, and run all the required services with a working out-of-the-box configuration. This is how the EDURange server will run once all scenarios are available in the refactor.

The second option is to run the platform through a pre-defined Docker-compose file. The advantage of this option is that it requires no additional installation of tools on the host server, once Docker is installed. Instead of running the required services directly on the system, it will download a pre-configured container for each micro-service. We have yet to do performance testing on this approach, but it may be more costly due to running extra possibly nested containers. That wouldn't matter for instructors that may only be running one or two scenarios at a time, but it would be much more costly in terms of performance if the central server were running this way.

3 Results

3.1 Website

The front-end web application has been fully implemented, excluding some specific functionality related to scenarios such as the code responsible for monitoring and logging student activity. Because some of these features are missing, it's not completely fair to compare the amount of code from the two web applications. However, the massive difference between them is still indicative of the scale of the reorganization, yet this was accomplished in 6 months as a two-quarter project course for three students.

The legacy application actually has more Python than the refactor, despite being written primarily in Ruby. This is due to frequent redundancy in code, whereby some very large Python files for logging student activity need to be stored redundantly across all scenarios. The reduction in code was one part of the reorganization. Simply cleaning up and removing redundant code would not have simplified the structure and could have taken a comparable amount of time.

Legacy Web App

Language	Number of Lines
Bash	1204
C	359
CoffeeScript	49
CSS	418
HCL	3973
JavaScript	1917
Markdown	2133
Pan	1238
Perl	3680
Python	3770
Ruby	7635
Ruby HTML	840
YAML	1254
Total	28,290

Refactor Web App

Language	Number of Lines
Assembly	394
Bash	745
C	1977
CSS	152
Dockerfile	46
HTML	2062
Javascript	391
JSON	1668
Python	3323
Shell	47
YAML	932
Total	11,737

3.2 Scenarios

At this point, six of the eight original EDURange scenarios have been converted to work in the refactor, and the system is fully functional. Scenarios are being dynamically created from Terraform templates, and the management system ensures that network addresses do not overlap, and that all containers are accessible.

The first release demonstrated some performance and efficiency improvements. Most notably, the minimum amount of time required to boot up a scenario has gone from two minutes, to less than ten seconds. This is because Terraform can simply start up containers from pre-downloaded images, rather than having to communicate with AWS and waiting for virtual machines to start. The refactor also improves the performance of the scenarios themselves, because containers can evenly divide up the hardware resources of the host machine rather than being restricted by preset AWS instance sizes.

4 Conclusions and Future Work

At this point, the refactor has been a success in terms of making the platform easier to approach for new developers, as well as more easily expandable. This can be demonstrated through the vastly simplified database schema, the reduction of the amount of code used to initialize the web application, and the better organized workflow for adding new pages and features to the platform. The best evidence for the success of the refactor, however, comes from the active participation and rapid progress made by the new development team.

Previously, after attempting to train 10 new developers to work on the Rails platform over the Summer of 2019, only 1 developer was able to become confident with Ruby on Rails development. By comparison, the 3 students trained during this refactor were all able to feel equipped to begin contributing within a few weeks of learning how to use Flask.

The next task in this refactor will be expanding the scenario management functionality. The platform should facilitate instructors creating their own fully customized scenarios with relative ease, by automating the generation of Terraform configurations from templates through the instructor interface. Since Terraform configurations are modular, we can easily implement a form on the web application where instructors can input information about the changes they'd like to make to a scenario. Student researchers can contribute early on in their learning process by writing bash scripts that accomplish small tasks, such as installing packages or changing ssh port numbers. As our library of small bash scripts grows, instructors will more easily be able to customize and create new scenarios.

The scenario scoring system must be re-implemented. In order to assess student understanding we don't just rely on their completing the tasks, instead we have questions that they answer in the student interface. In general, the correct answers are supplied by the database. This was never fully implemented in the original system, where ad hoc methods were used. However, using the database allows the creation of queries to filter and project tables to give instructors customized views of the scoring results.

Another feature that will be integral to research work related to the platform is student activity logging. This activity includes everything a student types and sees, and everywhere a student goes within a scenario. On the Rails EDURange platform this is implemented through custom TTYLog scripts and a data pipeline using an AWS S3 bucket from the scenarios to the central server [4] Log management will hopefully be simplified since the scenario containers will be more easily accessible than VM's, but the logging software is fragile since it must handle anything the student can type.

Research Artifacts

If you would like to find out more about EDURange, and maybe even use it yourself, see our GitHub repository for more details and setup instructions: `https://github.com/edurange/edurange-flask`

Acknowledgements

This work was partially supported by National Science Foundation grants 1723705 and 1723714.

References

[1] Stefan Boesen, Richard Weiss, James Sullivan, Michael E Locasto, Jens Mache, and Erik Nilsen. EDURange: meeting the pedagogical challenges of student participation in cybertraining environments. In *7th Workshop on Cyber Security Experimentation and Test (CSET)*, 2014.

[2] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA, 1999.

[3] Cynthia E. Irvine, Michael F. Thompson, Michael McCarrin, and Jean Khosalim. Live lesson: Labtainers: A docker-based framework for cybersecurity labs. In *2017 USENIX Workshop on Advances in Security Education (ASE 17)*, Vancouver, BC, August 2017. USENIX Association.

[4] Jelena Mirkovic, Aashray Aggarwal, David Weinman, Paul Lepe, Jens Mache, and Richard Weiss. Using terminal histories to monitor student progress on hands-on exercises. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, pages 866–872, 2020.

[5] Richard Weiss, Michael E. Locasto, and Jens Mache. A reflective approach to assessing student performance in cybersecurity exercises. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, SIGCSE '16, page 597–602, New York, NY, USA, 2016. Association for Computing Machinery.

[6] Richard Weiss, Franklyn Turbak, Jens Mache, and Michael E. Locasto. Cybersecurity education and assessment in EDURange. *IEEE Security & Privacy*, 15(03):90–95, May 2017.