

PrairieLearn Production Server Setup

Senior Design May 2024 - 33

Mitch Hudson
Tyler Weberski
Chris Costa
Andrew Winters
Carter Murawski
Matt Graham

Table of Contents

Setup Introduction	3
Acquire Virtual Machine	3
Firewall Setup	3
Set up SSH	4
SSH Multi-Factor Authentication (Google Authenticator)	6
Reverse Proxy (NGINX)	7
Install NGINX	7
Generate SSL Certificates	7
Configure NGINX	9
PrairieLearn Production Server	10
Install Prerequisites	10
Clone PrairieLearn	11
Set up OAuth2	11
Create Admin Account	18
Set up Start on Boot	19
Add Course to PrairieLearn	20
Conclusion	23
Appendix	24

Setup Introduction

This document intends to outline the steps required to set up and start running a PrairieLearn server in a production environment. For our purposes, we use Ubuntu 22.04, UFW, NGINX, and Docker Compose to run our environment.

Acquire Virtual Machine

The first step to setting up the production environment is to get a virtual machine from the admins at the ECE department of ISU. We used the form at <https://admin.ece.iastate.edu/requests/vm/new> to get a VM running on the university servers. Our specifications are Ubuntu 22.04, 4 CPU cores, 8 GB memory, 100 GB storage, and hostname cpre288-pl-f2023.ece.iastate.edu.

Firewall Setup

Once we have our server, the first thing we set up is our firewall. We do this first to ensure our server is protected while setting up our environment. We use ufw which comes pre-installed in Ubuntu 22.04, so we only have to set up the rules and enable it.

```
$ sudo ufw allow 22/tcp -c "SSH"
$ sudo ufw allow 80/tcp -c "HTTP"
$ sudo ufw allow 443/tcp -c "HTTPS"
$ sudo ufw default deny incoming
$ sudo ufw enable
```

The first three commands set up rules for allowing SSH, HTTP, and HTTPS incoming traffic. We use SSH to connect to the server, and HTTP and HTTPS are connected to our NGINX proxy. Notice we do NOT allow incoming traffic to port 3000 which is where our PrairieLearn server will be hosted. This is because doing so would allow users to connect directly to the PrairieLearn server and bypass our reverse proxy, as well as our HTTPS encryption. We only allow HTTP traffic on port 80 so we can redirect it to HTTPS using NGINX.

The fourth command is only used to ensure the firewall is set up to deny any incoming traffic not specifically mentioned by our rules.

Finally, the fifth command enables the firewall, protecting our server.

Set up SSH

To protect our server, we had each team member create a user on the virtual machine with their own password with sudo privileges. Then, we had each user create public-private key pairs using ssh-keygen to allow easy connection between the team and the server. Once each team member had working keys, we set up SSH to only allow public key authentication, disabling password authentication.

First, we create a new user for the team member's netid. We also make the new user a part of the sudo group, so they can use sudo. Then, we log in as the user to create ssh keys.

```
> ssh vm-user@cpre288-pl-f2023.ece.iastate.edu
Password: (for vm-user)
$ sudo adduser <netid>
Password: (for new account)
$ sudo usermod -aG sudo <netid>
$ sudo login <netid>
Password: (for new account)
```

For best practice, we use ed25519, which has similar security to RSA but with less computation and less space. After creating the keys, we add the public key to the authorized_keys file, and copy the private key to the team member's PC.

On Server:

```
$ ssh-keygen -t ed25519 -f ~/.ssh/<netid>_ed25519
Password: (for ssh key encryption)
$ cat ~/.ssh/<netid>_ed25519.pub >> ~/.ssh/authorized_keys
$ cat ~/.ssh/<netid>_ed25519
---- BEGIN PRIVATE KEY ---- ...
---- END PRIVATE KEY ----
```

We copy the private key to the client PC at
C:/Users/<username>/.ssh/<netid>_ed25519.

Once the user has the key installed on their system, we create a host for the server to simplify the connection process:

```
C:/Users/<username>/.ssh/config
...
Host sdmay24-33
    HostName cpre288-pl-f2023.ece.iastate.edu
    User <netid>
    IdentityFile C:/Users/<username>/.ssh/<netid>_ed25519
    IdentitiesOnly yes
...
```

Once this host is added, team members can log into the server by simply running

```
> ssh sdmay24-33
```

Finally, once every team member has created their account and can successfully connect to the server using public key authentication, we can disable password authentication and the original vm-user account.

Disabling password authentication and enabling MFA

```
$ sudo vim /etc/ssh/sshd_config
...
PubkeyAuthentication yes
PasswordAuthentication no
...
$ sudo systemctl restart sshd
```

Disabling vm-user by replacing /bin/bash with /usr/sbin/nologin

```
$ sudo vim /etc/passwd
...
vm-user:x:1000:1000:vm-user:/home/vm-user:/usr/sbin/nologin
...
```

SSH Multi-Factor Authentication (Google Authenticator)

If password authentication is still desirable despite the security implications, we added in MFA when using password logins. To do this we installed `libpam-google-authenticator` and applied it to our `sshd` configuration.

Installing `libpam-google-authenticator`

```
$ sudo apt install libpam-google-authenticator
```

Adding MFA to SSH.

```
$ sudo vim /etc/pam.d/sshd
...
auth required pam_google_authenticator.so
```

If desired, you can add `nullok` to the end of the `auth required` line to allow password logins without MFA when the user hasn't set it up yet.

Adding MFA to `sshd_config`

```
$ sudo vim /etc/ssh/sshd_config
...
KbdInteractiveAuthentication yes # Change this to yes
...
```

Finally, applying the changes by restarting `sshd`.

```
$ sudo systemctl restart sshd
```

After performing this setup, each user needs to log in and run the following commands to set up MFA.

```
$ google-authenticator
```

Running this command gives you a QR code to scan with the Google Authenticator app, as well as a secret key, if you don't want to scan the QR code. Scanning / entering this code adds the MFA to the app so you can get tokens when logging in.

It also gives you several “scratch codes” that are used to reset MFA if needed, so note these down somewhere. Finally, it will ask you a series of questions, which you need to respond to. I recommend responding in the following way:

- Make tokens “time based”: yes
- Update the `.google_authenticator` file: yes
- Disallow multiple uses: yes
- Increase the original generation time limit: no
- Enable rate limiting: yes

Lastly, it will ask you for the token from the Google Authenticator app before applying MFA to your account, which ensures you have the app set up correctly.

Reverse Proxy (NGINX)

NGINX can act as a reverse proxy, allowing us to pass our PrairieLearn instance through it to encrypt it via HTTPS.

To set this up, we needed to do a few different things.

1. Install NGINX
2. Generate and acquire signed SSL certificates
3. Configure NGINX

Install NGINX

Installing NGINX is a fairly easy process, only a few commands are required.

```
$ sudo apt update && sudo apt upgrade -y  
$ sudo apt install nginx
```

After running these commands, a basic NGINX server is installed and started, allowing us to access the default NGINX server through its hostname in a browser and HTTP.

Generate SSL Certificates

The next step to setting up our proxy is to generate certificates to be used by HTTPS.

This requires several steps and help from the professor, specifically for servers run by the university.

The first step is to use openssl to generate a certificate signing request. We use ECDSA, as it allows for similar security to high bit-count RSA, while using far fewer bits and requiring less arithmetic. First, we generate a new private key, then using that key we create a CSR that can be given to a certificate authority and signed.

```
$ openssl ecparam -aes256 -name prime256v1 -genkey -noout -out mykey.key
```

The `-aes256` part of the command encrypts the key, so we need to give it a password. Then, using the output file from this command we run the following to generate a CSR.

```
$ openssl req -new -sha256 -key mykey.key -out mycsr.csr
```

This CSR is then sent to <https://asw.iastate.edu/cgi-bin/acropolis/request/certificate/ssl> where the admins at ISU will sign it using their certificate authority and give us back a `mycrt.crt` file with the resulting certificate.

Lastly, we put our two files into the `/etc/ssl` folder, give them the proper access control, and rename them to our domain.

```
$ sudo cp mykey.key /etc/ssl/cpre288-pl-f2023.key
$ sudo cp mycrt.crt /etc/ssl/cpre288-pl-f2023.crt
$ sudo chmod 600 /etc/ssl/cpre288-pl-f2023.key
$ sudo chmod 644 /etc/ssl/cpre288-pl-f2023.crt
```

The key file is set to 600 because that means only root can read, write, or execute the file, and no one else is allowed. The certificate file is set to 644 because it needs to be readable by the other users on the system, as it is the public key.

Using the two files (`.key` and `.crt`), we can successfully set up HTTPS with NGINX.

Configure NGINX

Finally, we can configure NGINX to connect with our production PrairieLearn server and use HTTPS to connect to the end users.

```
$ sudo vim /etc/nginx/sites-available/default

server {
    listen 80 default_server;
    listen [::]:80 default_server;

    server_name _;

    return 301 https://$host$request_uri;
}

server {
    listen          443 ssl;
    listen          [::]:443 ssl;
    server_name     cpre288-pl-f2023.ece.iastate.edu;

    ssl_certificate /etc/ssl/cpre288-pl-f2023.crt;
    ssl_certificate_key /etc/ssl/cpre288-pl-f2023.key;
    access_log      /var/log/nginx/nginx.access.log;
    error_log       /var/log/nginx/nginx.error.log;

    location / {
        proxy_pass      http://localhost:3000;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_redirect  off;
    }
}
}
```

We use two servers, one listening on port 80 as the default and another listening on port 443 for HTTPS. We only use the default server (port 80) for redirecting to https.

In the HTTPS server, we set the `server_name` to our domain, the certificate and certificate key to the certificate files from the last step, and the access and error logs to generic log files in `/var/log`.

Since we are only using NGINX for a reverse proxy, we have no root folder for either server, and the default location `'/'` is only used to pass the proxy to the PrairieLearn server running on port 3000.

Finally, we restart NGINX with the below command

```
$ sudo nginx -s reload
```

PrairieLearn Production Server

To set up the PrairieLearn server in production mode there are several steps.

1. Install Prerequisites
2. Clone PrairieLearn
3. Set up OAuth2
4. Create Admin Account
5. Set up Autorun Service
6. Add Course to PrairieLearn

Install Prerequisites

The first step to setting up PrairieLearn is to install the prerequisites required by PrairieLearn. This includes docker and docker-compose.

```
$ sudo apt update && sudo apt upgrade -y  
$ sudo apt install docker docker.io docker-compose-v2
```

Clone PrairieLearn

Next, we clone PrairieLearn onto the server. To improve security, though, we create a separate user with limited privileges and access.

```
# useradd pl
Password: (pl account password)
# login pl
Password: (pl account password)
$ git clone https://www.github.com/PrairieLearn/PrairieLearn.git
$ cd PrairieLearn
$ touch docker.sh
$ chmod +x docker.sh
$ vim docker.sh
#!/bin/bash
sudo docker compose -f docker-compose-production.yml up
```

The docker.sh file is used by our pl account to start the docker container and run PrairieLearn. Since docker.sh uses sudo, the pl account needs to be able to run the file as root with sudo, so we need to edit the sudoers file. To do this, we add the following.

```
# visudo
...
pl    ALL=(root) /home/pl/PrairieLearn/docker.sh
...
```



All of this means that pl will be able to run ~/PrairieLearn/docker.sh as root using sudo.


Set up OAuth2


The next step is to set up OAuth2 to let users log in to the PrairieLearn server. In order to do this we need to create a Google account to be used as the group's main account. We created cpre288.pl.f2023@gmail.com for this purpose, using a randomly generated password.

With the account created, we can get API credentials through <https://console.cloud.google.com>. First, we create a new project, calling it cpre288.

New Project


 You have 11 projects remaining in your quota. Request an increase or delete projects. [Learn more](#) 

[MANAGE QUOTAS](#) 

Project name * 

Project ID: cpre288-403903. It cannot be changed later. [EDIT](#)

Location * [BROWSE](#)

 No organization

Parent organization or folder

[CREATE](#) [CANCEL](#)

Google API New Project screen


Then, we set up the OAuth consent screen, which is what is shown to users when they access our app.

OAuth consent screen


Choose how you want to configure and register your app, including your target users. You can only associate one app with your project.

User Type

Internal 

Only available to users within your organization. You will not need to submit your app for verification. [Learn more about user type](#) 

External 

Available to any test user with a Google Account. Your app will start in testing mode and will only be available to users you add to the list of test users. Once your app is ready to push to production, you may need to verify your app. [Learn more about user type](#) 

CREATE

[Let us know what you think](#) about our OAuth experience

OAuth consent screen creation

We use external because no one accessing PrairieLearn is in the cpre288.pl.f2023@gmail.com organization. This can be changed later, but would require significant effort and the creation of a Google organization that every student would need to be a part of.

App information

This shows in the consent screen, and helps end users know who you are and contact you

App name *
prairielearn-cpre288

The name of the app asking for consent

User support email *
cpre288.pl.f2023@gmail.com

For users to contact you with questions about their consent

App domain

To protect you and your users, Google only allows apps using OAuth to use Authorized Domains. The following information will be shown to your users on the consent screen.

Application home page
https://cpre288-pl-f2023.ece.iastate.edu

Provide users a link to your home page

Application privacy policy link

Provide users a link to your public privacy policy

Application terms of service link

Provide users a link to your public terms of service

Authorized domains

When a domain is used on the consent screen or in an OAuth client's configuration, it must be pre-registered here. If your app needs to go through verification, please go to the [Google Search Console](#) to check if your domains are authorized. [Learn more](#) about the authorized domain limit.

Authorized domain 1 *
iastate.edu

[+ ADD DOMAIN](#)

Developer contact information

Email addresses *
sdmay24-33@iastate.edu

These email addresses are for Google to notify you about any changes to your project.

App creation screen

Next, we created a new app for the OAuth consent screen. We called our app prairielearn-cpre288 to reflect the course and the fact that we are using PrairieLearn. We also added a link to the homepage, authorized iastate.edu as a domain, and added the sdmay24-33@iastate.edu email group to the developer contact information.

Our app didn't need any access to users' Google accounts, so we ignored the scopes page and moved on.

We also didn't need any test users, so we ignored that page too.

Once we created the OAuth consent screen, we moved on to creating an OAuth2 client ID so we could connect PrairieLearn to the service. We did this by clicking CREATE CREDENTIALS and OAuth client ID in the drop down.

The screenshot shows the Google Cloud API & Services console. On the left, the 'APIs & Services' sidebar is visible with the 'Credentials' option selected and highlighted in blue, indicated by a red arrow. The main content area is titled 'Credentials' and features a '+ CREATE CREDENTIALS' button, also highlighted with a red arrow. Below this button, there is a link to 'Learn more'. The main area is divided into three sections: 'API Keys', 'OAuth 2.0 Client IDs', and 'Service Accounts'. Each section has a table header with a checkbox, a column name, and a 'Creation date' column with a downward arrow. The 'API Keys' section shows 'No API keys to display'. The 'OAuth 2.0 Client IDs' section shows 'No OAuth clients to display'. The 'Service Accounts' section shows 'No service accounts to display'.

Instructions to creating OAuth2 credentials

A client ID is used to identify a single app to Google's OAuth servers. If your app runs on multiple platforms, each will need its own client ID. See [Setting up OAuth 2.0](#) for more information. [Learn more](#) about OAuth client types.

Application type *
Web application ▼

Name *
production-server

The name of your OAuth 2.0 client. This name is only used to identify the client in the console and will not be shown to end users.

i The domains of the URIs you add below will be automatically added to your [OAuth consent screen](#) as [authorized domains](#).

Authorized JavaScript origins **?**

For use with requests from a browser

URIs 1 *
https://cpre288-pl-f2023.ece.iastate.edu

+ ADD URI

Authorized redirect URIs **?**

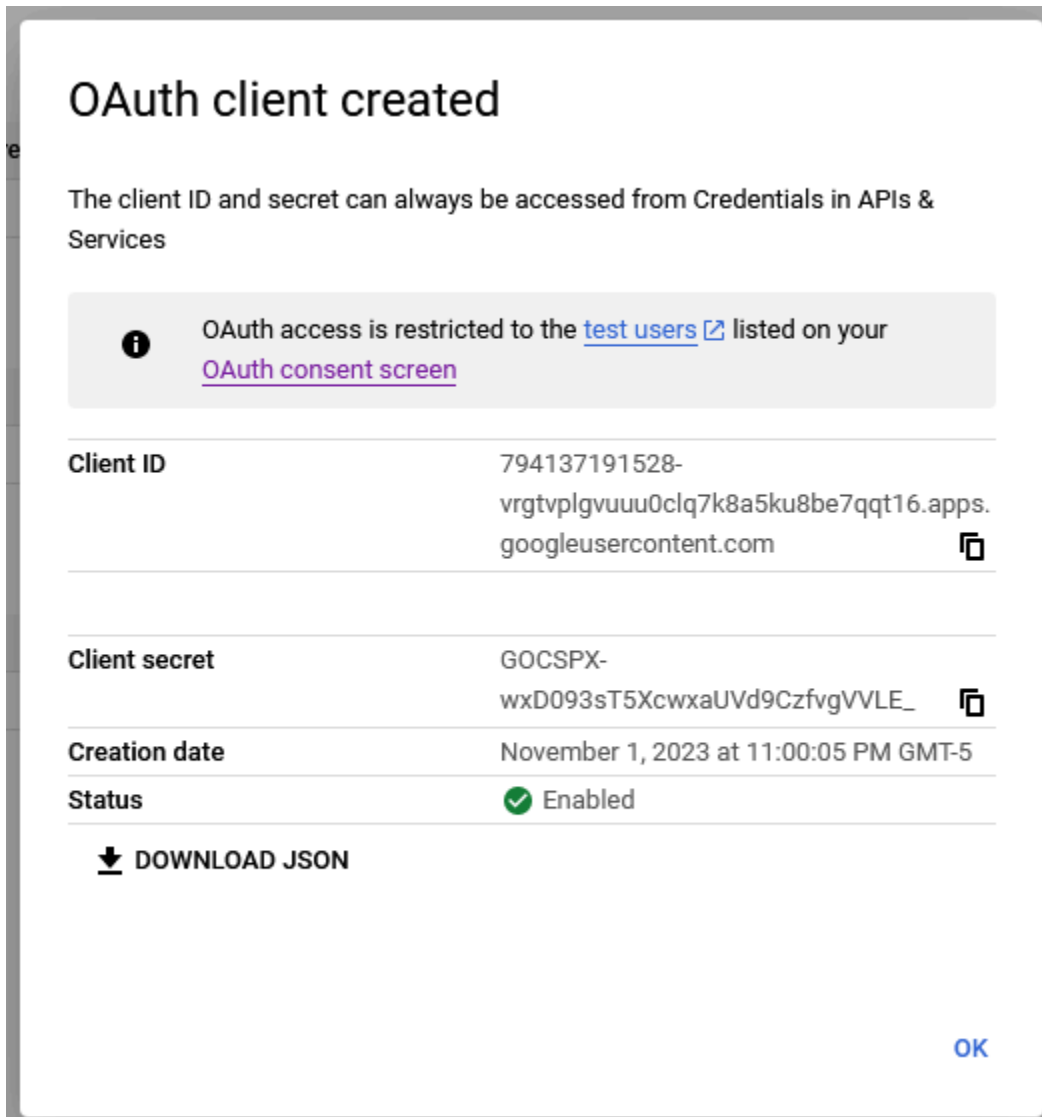
For use with requests from a web server

URIs 1 *
https://cpre288-pl-f2023.ece.iastate.edu/pl/oauth2callback

+ ADD URI

OAuth client ID creation screen

In the following menu we filled in the application type as a Web app, named it production-server, and filled in our URI for the JS origins and redirect URI. *Note: The url for the redirect is <https://cpre288-pl-f2023.ece.iastate.edu/pl/oauth2callback> as described by the PrairieLearn docs*



OAuth client created

The client ID and secret can always be accessed from Credentials in APIs & Services

i OAuth access is restricted to the [test users](#) listed on your [OAuth consent screen](#)

Client ID	794137191528-vrgtvplgvuuu0clq7k8a5ku8be7qqt16.apps.googleusercontent.com
Client secret	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
Creation date	November 1, 2023 at 11:00:05 PM GMT-5
Status	✔ Enabled

↓ DOWNLOAD JSON

OK

Successful OAuth client creation

Finally, we are given the client ID and secret for our OAuth2 app, which we can give to PrairieLearn. *Note: the above client ID and secret are not the actual ID and secret.*

To put these credentials into PrairieLearn, we create a new file in the PrairieLearn clone root folder called config.json

```
$ vim ~/PrairieLearn/config.json

{
  "serverCanonicalHost": "https://cpre288-pl-f2023.ece.iastate.edu",
  "googleClientId": "<Client ID>",
  "googleClientSecret": "<Client Secret>",
  "googleRedirectUrl":
  "https://cpre288-pl-f2023.ece.iastate.edu/pl/oauth2callback",
  "hasOAuth": true
}
```

The first config “serverCanonicalHost” tells PrairieLearn what URL it is hosted at, and the last four are for setting up the OAuth.

Finally, we need to tell PrairieLearn to use our config file, and since we use docker compose, we need to edit the docker-compose-production.yml file to include it.

```
$ vim ~/PrairieLearn/docker-compose-production.yml

...
services:
  pl:
    ...
    volumes:
      ...
      - ./config.json:/PrairieLearn/config.json
    ...
...

```

We can now run the production server by running docker.sh as the pl user.

```
$ sudo ~/PrairieLearn/docker.sh
```

Create Admin Account

The next step is to create an admin account in PrairieLearn that can create and modify courses. We do this by first logging into the production server using our new Google

account from the OAuth setup. Once we log in our user is added to PrairieLearn's database, and we can use a second SSH instance to access the docker container to change its permissions.

```
$ sudo docker ps
CONTAINER ID   ...           NAMES
d47b4b7ff78f   ...           <name>
$ sudo docker exec -it <name> /bin/bash
# psql postgres
# SELECT * FROM users;
user_id | ...
-----+-----
      <id> | ...
# INSERT INTO administrators (user_id) VALUES (<id>);
```

The first command finds the docker container name, and the second connects to it with an interactive bash shell. Next, we log into the postgres database and find the user we created. Finally, we add the user into the administrators table, thus making our account an administrator that can edit, create, and delete courses.

Set up Start on Boot

To make the server startup on boot (in case the server needs to be rebooted, for example) we only need to change a few things.

First, we need to set docker to run on boot.

```
$ sudo systemctl start docker
$ sudo systemctl enable docker
```

Then, we need to edit the docker-compose-production.yml file to auto restart the service by adding the line restart: always after pl.

```
$ vim ~/PrairieLearn/docker-compose-production.yml

services:
  pl:
    restart: always
    ...
```

Now, whenever the docker container stops (except manually), docker is restarted, or the virtual machine is restarted, the PrairieLearn docker will be automatically restarted.

Add Course to PrairieLearn

The last step in creating our environment is adding the course from our git repository to the PrairieLearn production server.


First, in GitLab, we need to create an access token that has the Reporter role with repository_read access.

Add a project access token


Token name

For example, the application using the token or the purpose of the token. Do not give sensitive information.

Expiration date

Select a role

Select scopes

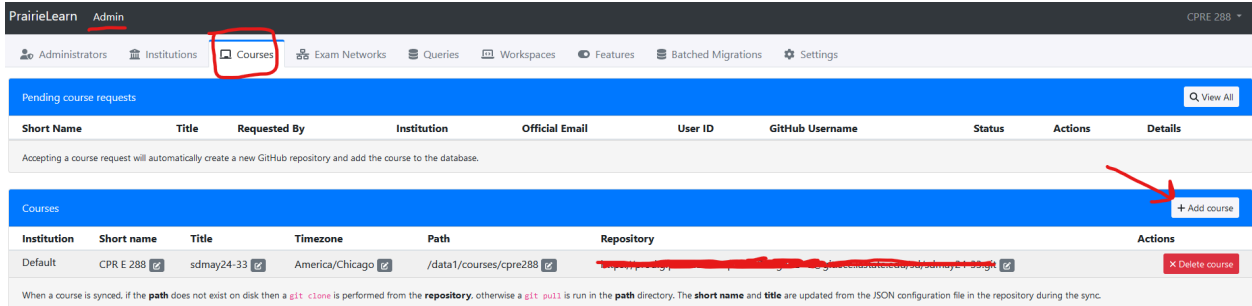
Scopes set the permission levels granted to the token. [Learn more.](#)

- api**
Grants complete read and write access to the scoped project API, including the Package Registry.
- read_api**
Grants read access to the scoped project API, including the Package Registry.
- create_runner**
Grants create access to the runners.
- k8s_proxy**
Grants permission to perform Kubernetes API calls using the agent for Kubernetes.
- read_repository**
Grants read access (pull) to the repository.
- write_repository**
Grants read and write access (pull and push) to the repository.

Add token screen

Once we get the token, we can use the url <https://prod:<token>@git.ece.iastate.edu/sd/sdmay24-33.git> to pull updates from the production server. We can also add write_repository later to allow admins to edit the courses.

The last step to adding the course is to go to the Admin console from the website and add a course with the above link and a path.



Add course button

The image shows a 'Add new course' form with the following fields and values:

- Institution: Default
- Short name: XC 101
- Title: Template course title
- Timezone: America/Chicago
- Path: /data1/courses/cpre288
- Repository: ate.edu/sd/sdmay24-33.git
- Branch: master

Buttons: Cancel, Add course

Background elements: CPRE 288, View All, + Add course, Delete course

Add course prompt

Once it is created, all we need to do is sync it to download all the updates from the git repo.

Questions Sync / Choose course instance... ▾

ances Files Issues ? Questions Settings Staff Sync # Tags 99 Topics

bc779d0521af7e87a93b0fec7ed5f79aa115590c

/data1/courses/cpre288

https://prod:glpat-dkzYWcqMvbPCH5gaAD-C@git.ece.iastate.edu/sd/sdmay24-33.git

Show server git status

Pull from remote git repository

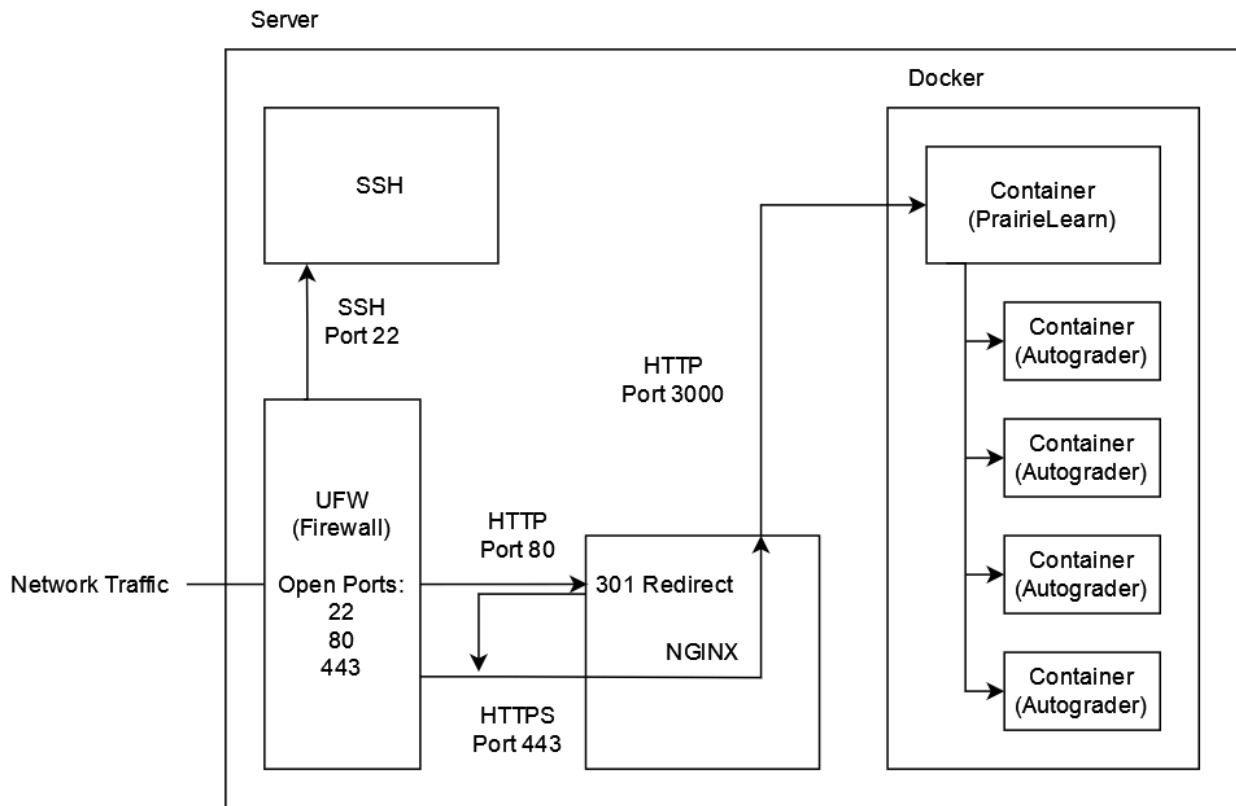
Tag	Digest	Image size	Last sync	Actions	Used by
					1 question <small>Show</small>

Sync button

Conclusion

Through this process we created and set up a virtual machine to host PrairieLearn in a Docker container behind an NGINX reverse proxy encrypting traffic through HTTPS. Appendix A shows the full internal diagram of the virtual machine. UFW monitors traffic and ensures that the server only responds to SSH, HTTP, and HTTPS traffic. NGINX redirects HTTP traffic to HTTPS and encrypts responses from PrairieLearn using HTTPS. PrairieLearn and Docker handle the course environment visible to students and several Docker containers are created as autograders to be used by PrairieLearn for compiling and testing student-submitted code.

Appendix



Appendix A: Internal server diagram