CTFd Documentation

Release 2.2.3

Kevin Chung

Jan 21, 2020

Contents:

CTFd is a Capture The Flag framework focusing on ease of use and customizability.

CTFd comes with most of the features needed by an event organizer to run a competition or workshop. In addition, if CTFd's feature set is insufficient, CTFd allows for the usage of plugins and themes to control almost every aspect of how it looks and behaves.

CTFd is used by many different clubs, universities, and companies to host their own Capture The Flags.

While available as open source, CTFd developers also provide a managed hosting service available at https://ctfd.io/.

CTFd is written in Python and makes use of the Flask web framework.

CHAPTER 1

Deployment

CTFd is a standard WSGI application so most if not all Flask documentation on deploying a Flask based application should apply. This page will focus on the recommended ways to deploy CTFd.

Important: Fully managed and maintained CTFd deployments are available at https://ctfd.io.

1.1 Docker

CTFd provides automatically generated Docker images and one of the simplest means of deploying a CTFd instance is to use Docker Compose.

Caution: While Docker can be a very simple means of deploying CTFd, it can make debugging and receiving support more complicated. Before deploying using Docker, be sure you have a cursory understanding of how Docker works.

1. Install Docker
2. Install Docker Compose
3. Clone the CTFd repository with `git clone https://github.com/CTFd/CTFd.git`
4. Modify the `docker-compose.yml` file from the repository to specify a `SECRET_KEY` environment for the CTFd service.

```
environment:
  - SECRET_KEY=<SPECIFY_RANDOM_VALUE>
  - UPLOAD_FOLDER=/var/uploads
  - LOG_FOLDER=/var/log/CTFd
  - DATABASE_URL=mysql+pymysql://root:ctfd@db/ctfd
  - REDIS_URL=redis://cache:6379
  - WORKERS=4
```

Tip: You can also run `python -c "import os; f=open('.ctfd_secret_key', 'a+'); f.write(os.urandom(64)); f.close()"` within the CTFd repo to generate a .ctfd_secret_key file.

5. Run `docker-compose up`
6. You should now be able to access CTFd at http://localhost:8000

Note: The default Docker Compose configuration files do not provide a reverse proxy server or configure SSL/TLS. This is left as an exercise to the reader by design.

1.2 Standard WSGI Deployment

As a web application, CTFd has a few dependencies which require installation.

1. WSGI server
2. Database server
3. Caching server

1.2.1 WSGI Server

While CTFd is a standard WSGI application and most WSGI servers (e.g. gunicorn, UWSGI, waitress) will likely suffice, CTFd is most commonly deployed with gunicorn. This is because of its simplicity and reasonable performance. It is installed by default and used by other deployment methods discussed on this page. By default it is configured to use gevent as a worker class.

1.2.2 Database Server

CTFd makes use of SQLAlchemy and as such supports a number of SQL databases. As of CTFd 2.0, the recommended database type is MySQL. CTFd is tested and has been installed against SQLite, Postgres, and MariaDB but this could change in the future.

By default CTFd will create a SQLite database if no database server has been configured.

Note: CTFd makes use of the JSON data type. MySQL >= 5.7.8 implements a proper JSON type while MariaDB does not. Small differences like these could eventually result in CTFd only supporting a few database servers.

1.2.3 Caching Server

CTFd makes heavy use of caching servers to store configuration values, user sessions, and page content. It is important to deploy CTFd with a caching server. The preferred caching server option is Redis.

By default if no cache server is configured, CTFd will attempt to use the filesystem as a cache and store values in the `.data` folder. This type of caching is not very performant thus it is highly recommended that you configure a Redis server.

1.3 Vagrant

CTFd provides a basic Vagrantfile for use with Vagrant. To run using Vagrant run the following commands:

```
vagrant up
```

Visit http://localhost:8000 where CTFd will be running.

To access the internal gunicorn terminal session inside Vagrant run:

```
vagrant ssh
tmux attach ctfd
```

Note: CTFd's Vagrantfile is not commonly used and is only community supported

1.4 Debug Server

The absolute simplest way to deploy CTFd merely involves running *python serve.py* to start Flask's built-in debugging server. This isn't recommended for anything but debugging and should not be used for any kind of load. It is discussed here because the debugging server can make identifying bugs and misconfigurations easier. In addition, development mostly occurs using the debug server.

Important: CTFd makes every effort to be an easy to setup application. However, deploying CTFd for large amounts of users can be difficult.

Fully managed and maintained CTFd deployments are available at https://ctfd.io. If you're interested in a specialized CTFd deployment with custom features please contact us.

CHAPTER 2

Configuration

CTFd provides a number of configuration options which are used to configure server behavior. CTFd makes a distinction between configuration values which can be configured only with server-level access and values which can be configured by those with administrative priveleges on CTFd.

2.1 Server Level Configuration

Server level configuration can be modified from the `config.py` file in CTFd.

2.1.1 SECRET_KEY

The secret value used to creation sessions and sign strings. This should be set to a random string. In the interest of ease, CTFd will automatically create a secret key file for you. If you wish to add this secret key to your instance you should hard code this value to a random static value.

You can also remove .ctfd_secret_key from the .gitignore file and commit this file into whatever repository you are using.

http://flask.pocoo.org/docs/latest/quickstart/#sessions

2.1.2 DATABASE_URL

The URI that specifies the username, password, hostname, port, and database of the server used to hold the CTFd database.

e.g. `mysql+pymysql://root:<YOUR_PASSWORD_HERE>@localhost/ctfd`

2.1.3 REDIS_URL

The URI to connect to a Redis server.

e.g. `redis://user:password@localhost:6379`

http://pythonhosted.org/Flask-Caching/#configuring-flask-caching

2.1.4 MAILFROM_ADDR

The email address that emails are sent from if not overridden in the configuration panel.

2.1.5 MAIL_SERVER

The mail server that emails are sent from if not overriden in the configuration panel.

2.1.6 MAIL_PORT

The mail port that emails are sent from if not overriden in the configuration panel.

2.1.7 MAIL_USEAUTH

Whether or not to use username and password to authenticate to the SMTP server

2.1.8 MAIL_USERNAME

The username used to authenticate to the SMTP server if MAIL_USEAUTH is defined

2.1.9 MAIL_PASSWORD

The password used to authenticate to the SMTP server if MAIL_USEAUTH is defined

2.1.10 MAIL_TLS

Whether to connect to the SMTP server over TLS

2.1.11 MAIL_SSL

Whether to connect to the SMTP server over SSL

2.1.12 MAILGUN_API_KEY

Mailgun API key to send email over Mailgun

2.1.13 MAILGUN_BASE_URL

Mailgun base url to send email over Mailgun

2.1.14 LOG_FOLDER

The location where logs are written. These are the logs for CTFd key submissions, registrations, and logins. The default location is the CTFd/logs folder.

2.1.15 UPLOAD_PROVIDER

Specifies the service that CTFd should use to store files.

2.1.16 UPLOAD_FOLDER

The location where files are uploaded. The default destination is the CTFd/uploads folder.

2.1.17 AWS_ACCESS_KEY_ID

AWS access token used to authenticate to the S3 bucket.

2.1.18 AWS_SECRET_ACCESS_KEY

AWS secret token used to authenticate to the S3 bucket.

2.1.19 AWS_S3_BUCKET

The unique identifier for your S3 bucket.

2.1.20 AWS_S3_ENDPOINT_URL

A URL pointing to a custom S3 implementation.

2.1.21 REVERSE_PROXY

Specifies whether CTFd is behind a reverse proxy or not. Set to `True` if using a reverse proxy like nginx.

See Flask documentation for full details.

Tip: You can also specify a comma seperated set of numbers specifying the reverse proxy configuration settings. For example to configure *x_for=1, x_proto=1, x_host=1, x_port=1, x_prefix=1* specify *1,1,1,1,1*. By setting the value to `True`, CTFd will default to the above behavior with all proxy settings set to 1.

2.1.22 TEMPLATES_AUTO_RELOAD

Specifies whether Flask should check for modifications to templates and reload them automatically.

2.1.23 SQLALCHEMY_TRACK_MODIFICATIONS

Automatically disabled to suppress warnings and save memory. You should only enable this if you need it.

2.1.24 SWAGGER_UI

Enable the Swagger UI endpoint at `/api/v1/`

2.1.25 UPDATE_CHECK

Specifies whether or not CTFd will check whether or not there is a new version of CTFd

2.1.26 APPLICATION_ROOT

Specifies what path CTFd is mounted under. It can be used to run CTFd in a subdirectory. Example: /ctfd

2.1.27 SERVER_SENT_EVENTS

Specifies whether or not to enable to server-sent events based Notifications system.

2.1.28 OAUTH_CLIENT_ID

2.1.29 OAUTH_CLIENT_SECRET

2.2 Application Level Configuration

CHAPTER 3

Scoring

Scoring is central to any CTF. CTFd automatically generates a scoreboard that automatically resolves ties and supports score freezing. CTFd supports two models which can alter the score of a user or team.

3.1 Solves

Solves are what mark a challenge as solved. Solves do not carry a value and defer to the value of their respective Challenge.

3.2 Awards

Awards have a value defined by their creator (usually an admin). They can be used to give a user/team arbitrary (positive or negative) points.

3.3 Tie Breaks

In CTFd, tie breaks are essentially resolved by time. If two teams have the same score, the team with the lower solve ID in the database will be considered on top. For example Team X and Team Y solve the same challenge five minutes apart and both now have 100 points.

Team X will have a Solve ID of 1 for their submission and Team Y will have a Solve ID of 2 for their submission.

Thus Team X will be considered the tie winner.

3.4 Formats

3.4.1 MajorLeagueCyber

MajorLeagueCyber (MLC) is a cyber security event tracker designed and maintained by the developers of CTFd. It provides polling of the CTFd API and can record and aggregate scores between competitions/events. It supports parsing and processing of CTFd's built in scoreboard API format.

To register your event with MLC:

1. Register an account at https://www.majorleaguecyber.org/ if you don't already have one.
2. Create a new event.
3. Edit the event and add the API Scoreboard URL under the Integrations section. For CTFd you should enter `https://[CTFd Instance URL]/api/v1/scoreboard`
4. Access the JSON scoreboard API from MajorLeagueCyber by going to `https://www.majorleaguecyber.org/events/[EVENT_ID]/[EVENT_NAME]/scoreboard.json`

3.4.2 CTFTime

In prior versions CTFd supported a CTFTime compatible scoreboard. This is no longer directly supported because the CTFTime scoreboard format is inherently limiting. However, MLC allows for the polling of any JSON scoreboard API and can translate to the CTFTime scoreboard format.

After registering your event on MLC you can access the legacy scoreboard format by going to `https://www.majorleaguecyber.org/events/[EVENT_ID]/[EVENT_NAME]/scoreboard.json?format=legacy`.

CHAPTER 4

Themes

CTFds allows organizers to customize their CTFd instance with custom themes. The CTFd core routes will load templates from the theme folder specified by the theme configuration value. This value can be configured in the admin panel.

Tip: Official CTFd themes are available at https://ctfd.io/store. Contact us regarding custom themes and special projects.

Tip: Community themes are available at https://github.com/CTFd/themes.

CHAPTER 5

Plugins

5.1 Introduction

CTFd features a plugin interface allowing for the modification of CTFd behavior without modifying the core CTFd code. This has a number of benefits over forking and modifying CTFd:

- Your modifications and plugins can be shared more easily
- CTFd can be updated without losing any custom behavior

The CTFd developers will do their best to not introduce breaking changes but keep in mind that the plugin interface is still under development and could change.

Tip: Official CTFd plugins are available at https://ctfd.io/store. Contact us regarding custom plugins and special projects.

Tip: Community plugins are available at https://github.com/CTFd/plugins.

5.2 Architecture

CTFd plugins are implemented as Python modules with some CTFd specific files.

```
CTFd
└── plugins
    └── CTFd-plugin
        ├── README.md          # README file
        ├── __init__.py        # Main code file loaded by CTFd
        ├── requirements.txt   # Any requirements that need to be installed
        └── config.json        # Plugin configuration file
```

Effectively CTFd will look at every folder in the `CTFd/plugins` folder for the `load()` function.

If the `load()` function is found, CTFd will call that function with itself (as a Flask app) as a parameter (i.e. `load(app)`). This is done after CTFd has added all of its internal routes but before CTFd has fully instantiated itself. This allows plugins to modify many aspects of CTFd without having to modify CTFd itself.

5.2.1 config.json

`config.json` exists to give plugin developers a way to define attributes about their plugin. It's primary usage within CTFd is to give users a way to access a Configuration or Settings page for the plugin.

This is an example `config.json` file:

```
{
    "name": "CTFd Plugin",
    "route": "/admin/custom_plugin_route"
}
```

This is ultimately rendered to the user with the following template snippet:

```
{% if plugins %}
<li>
    <a href="#" class="dropdown-toggle" data-toggle="dropdown" role="button" aria-
↪haspopup="true" aria-expanded="false">Plugins <span class="caret"></span></a>
    <ul class="dropdown-menu">
            {% for plugin in plugins %}
                    <li><a href="{{ request.script_root }}{{ plugin.route }}">{{␣
↪plugin.name }}</a></li>
            {% endfor %}
    </ul>
</li>
{% endif %}
```

5.2.2 config.html

In the past CTFd used a static file known as `config.html` which existed to give plugin developers a page that is loaded by the CTFd admin panel. This has been superceded in favor of *config.json* but is still supported for backwards compatability.

The `config.html` file for a plugin is available by CTFd admins at `/admin/plugins/<plugin-folder-name>`. Thus if `config.html` is stored in `CTFd-S3-plugin`, it would be available at `/admin/plugins/CTFd-S3-plugin`.

`config.html` is loaded as a Jinja template so it has access to all of the same functions and abilities that CTFd exposes to Jinja. Jinja templates are technically also capable of running arbitrary Python code but this is ancillary.

5.3 Adding New Routes

Adding new routes in CTFd is effectively just an exercise in writing new Flask routes. Since the plugin itself is passed the entire app, the plugin can leverage the `app.route` decorator to add new routes.

A simple example is as follows:

```
from flask import render_template


def load(app):
    @app.route('/faq', methods=['GET'])
    def view_faq():
        return render_template('page.html', content="<h1>FAQ Page</h1>")
```

5.4 Modifying Existing Routes

It is slightly more complicated to override existing routes in CTFd/Flask because it is not strictly supported by Flask. The approach currently used is to modify the `app.view_functions` dictionary which contains the mapping of routes to the functions used to handle them.

```
from flask import render_template

from CTFd.models import db
from CTFd.utils import admins_only, is_admin

from CTFd import utils

def load(app):
    def view_challenges():
        return render_template('page.html', content="<h1>Challenges are currently␣
↪closed</h1>")

    # The format used by the view_functions dictionary is blueprint.view_function_name
    app.view_functions['challenges.challenges_view'] = view_challenges
```

If for some reason you wish to add a new method to an existing route you can modify the `url_map` as follows:

```
from werkzeug.routing import Rule

app.url_map.add(Rule('/challenges', endpoint='challenges.challenges_view', methods=[
↪'GET', 'POST']))
```

5.5 Adding Database Tables

Sometimes CTFd doesn't have enough database tables or columns to let you do what you need. In this case you can use a plugin to create a new table and then use the information in the previous two sections to create routes or modify existing routes to access your new table.

```
from CTFd.models import db


class Avatars(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    team = db.Column(db.Integer, db.ForeignKey('teams.id'))
    location = db.Column(db.Text)

    def __init__(self, team, location):
        self.target = team
```

(continues on next page)

(continued from previous page)

```
        self.location = location


def load(app):
    app.db.create_all()
    @app.route('/profile/avatar', methods=['GET', 'POST'])
    def profile_avatars():
        raise NotImplementedError
```

5.6 Replacing Templates

In some situations it might make sense for your plugin to replace the logic for a single page template instead of creating an entire theme.

The `utils.override_template()` function allows a plugin to replace the content of a single template within CTFd such that CTFd will use the new content instead of the content in the original file.

```
from CTFd.utils import override_template
import os

def load(app):
    dir_path = os.path.dirname(os.path.realpath(__file__))
    template_path = os.path.join(dir_path, 'new-scoreboard.html')
    override_template('scoreboard.html', open(template_path).read())
```

With this code CTFd will use `new-scoreboard.html` instead of the `scoreboard.html` file it normally would have used.

5.7 Registering Assets

Very often you will want to provide users with static assets (e.g. JS, CSS). Instead of registering handlers for them on your own, you can use the CTFd built in plugin utilities, `register_plugin_assets_directory` and `register_plugin_asset`.

For example to register an entire assets directory as available to the user:

```
from CTFd.plugins import register_plugin_assets_directory

def load(app):
    # Available at http://ctfd/plugins/test_plugin/assets/
    register_plugin_assets_directory(app, base_path='/plugins/test_plugin/assets/')
```

Or to only provide a single file:

```
from CTFd.plugins import register_plugin_asset

def load(app):
    # Available at http://ctfd/plugins/test_plugin/assets/file.js
    register_plugin_asset(app, asset_path='/plugins/test_plugin/assets/file.js')
```

5.8 Challenge Types

In CTFd, there is a concept of a type of challenge. Most CTFs only ever provide challenges as a snippet of text alongside some files. CTFd expands upon this and allows developers to create new challenge types which diversify what users will see.

Ultimately, users will still read some text, and submit some value but CTFd allows you to style and customize this so users can submit data in new ways.

For example, instead of an input to submit a single flag value, you might require teams to submit multiple flags or you might create some kind of customized UI where teams need to arrange blocks or text in some order.

The approach used by CTFd here is to give each "type" of challenge an ID and a name.

Tip: You can see how CTFd implements its default standard challenge here. You can also see how CTFd implements dynamic scoring using this feature.

Each challenge is implemented as a child class of the `BaseChallenge` and implements static methods named `create`, `read`, `update`, `delete`, `attempt`, `solve`, and `fail`.

When a user attempts to solve a challenge, CTFd will look up the challenge type and then call the `solve` method as shown in the following snippet of code:

```
chal_class = get_chal_class(chal.type)
status, message = chal_class.attempt(chal, request)

if status:  # The challenge plugin says the input is right
    if ctftime() or is_admin():
        chal_class.solve(team=team, chal=chal, request=request)
    return jsonify({'status': 1, 'message': message})

else:  # The challenge plugin says the input is wrong
    if ctftime() or is_admin():
        chal_class.fail(team=team, chal=chal, request=request)
```

This structure allows each Challenge Type to dictate how they are attempted, solved, and marked incorrect.

The Challenge Type also dictates the database table that it uses to store data. By default this uses the `type` column as a `polymorphic_identity` to implement table inheritance. Effectively each child table will use the Challenges table as a parent. The child table can add whatever columns it wishes but still leverage the existing columns from the parent.

We can see in the following code that the polymorphic_identity is specified to be `dynamic` as well as the `type` parameter. We can also see the call to `create_all()` which will create the table in our database.

```
class DynamicChallenge(Challenges):
    __mapper_args__ = {'polymorphic_identity': 'dynamic'}
    id = db.Column(None, db.ForeignKey('challenges.id'), primary_key=True)
    initial = db.Column(db.Integer)
    minimum = db.Column(db.Integer)
    decay = db.Column(db.Integer)

    def __init__(self, name, description, value, category, type='dynamic', minimum=1,␣
↪decay=50):
        self.name = name
        self.description = description
        self.value = value
```

(continues on next page)

(continued from previous page)

```
        self.initial = value
        self.category = category
        self.type = type
        self.minimum = minimum
        self.decay = decay


def load(app):
    app.db.create_all()
    CHALLENGE_CLASSES['dynamic'] = DynamicValueChallenge
    register_plugin_assets_directory(app, base_path='/plugins/DynamicValueChallenge/
↪assets/')
```

This code creates the necessary tables for the Challenge Type plugin which should be used in addition to the staticmethods used to define the challenge's behavior.

Every challenge type must be added to the global dictionary that specifies all challenge types:

```
CHALLENGE_CLASSES = {
    "standard": CTFdStandardChallenge
}


def get_chal_class(class_id):
    cls = CHALLENGE_CLASSES.get(class_id)
    if cls is None:
        raise KeyError
    return cls
```

The Standard Challenge type provided within CTFd can be used as a base from which to build additional Challenge Type plugins.

Once new challenges are registered, CTFd will provide a dropdown allowing you to choose from all the challenge types you can create.

Each Challenge Type contains templates and scripts dictionaries which contain the routes for HTML and JS files needed for the operation of the modals used to create and update the challenges.

These routes are not automatically defined by CTFd.

Each challenge type plugin specifies the location of their own templates and scripts. An example is the built in standard challenge type plugin. It specifies the URLs that the assets are located at for the user's browser to load:

```
templates = {  # Templates used for each aspect of challenge editing & viewing
    'create': '/plugins/challenges/assets/create.html',
    'update': '/plugins/challenges/assets/update.html',
    'view': '/plugins/challenges/assets/view.html',
}
scripts = {  # Scripts that are loaded when a template is loaded
    'create': '/plugins/challenges/assets/create.js',
    'update': '/plugins/challenges/assets/update.js',
    'view': '/plugins/challenges/assets/view.js',
}
```

These files are registered with Flask with the following code:

```
from CTFd.plugins import register_plugin_assets_directory
```

(continues on next page)

(continued from previous page)

```
def load(app):
    register_plugin_assets_directory(app, base_path='/plugins/challenges/assets/')
```

The aforementioned code handles the Python logic around new challenges but in order to fully integrate with CTFd you will need to create new Nunjucks templates to give admins/teams the ability to modify/update/solve your challenge. The templates used by the Standard Challenge Type should serve as examples.

5.9 Flag Types

Flag types conversely are used to give developers a way to allow teams to submit flags which do not conform to a hardcoded string or a regex-able value.

The approach is very similar to Challenges with a base Flag/Key class and a global dictionary specifying all the Flag/Key types:

```
class BaseFlag(object):
    name = None
    templates = {}

    @staticmethod
    def compare(self, saved, provided):
        return True


class CTFdStaticFlag(BaseFlag):
    name = "static"
    templates = {  # Nunjucks templates used for key editing & viewing
        "create": "/plugins/flags/assets/static/create.html",
        "update": "/plugins/flags/assets/static/edit.html",
    }

    @staticmethod
    def compare(chal_key_obj, provided):
        saved = chal_key_obj.content
        data = chal_key_obj.data

        if len(saved) != len(provided):
            return False
        result = 0

        if data == "case_insensitive":
            for x, y in zip(saved.lower(), provided.lower()):
                result |= ord(x) ^ ord(y)
        else:
            for x, y in zip(saved, provided):
                result |= ord(x) ^ ord(y)
        return result == 0


class CTFdRegexFlag(BaseFlag):
    name = "regex"
    templates = {  # Nunjucks templates used for key editing & viewing
        "create": "/plugins/flags/assets/regex/create.html",
        "update": "/plugins/flags/assets/regex/edit.html",
```

(continues on next page)

(continued from previous page)

```
    }

    @staticmethod
    def compare(chal_key_obj, provided):
        saved = chal_key_obj.content
        data = chal_key_obj.data

        if data == "case_insensitive":
            res = re.match(saved, provided, re.IGNORECASE)
        else:
            res = re.match(saved, provided)

        return res and res.group() == provided


FLAG_CLASSES = {"static": CTFdStaticFlag, "regex": CTFdRegexFlag}


def get_flag_class(class_id):
    cls = FLAG_CLASSES.get(class_id)
    if cls is None:
        raise KeyError
    return cls
```

When a challenge solution is submitted, the challenge plugin itself is responsible for:

1. Loading the appropriate Key class using the `get_flag_class()` function.
2. Properly calling the static `compare()` method defined by each Flag class.
3. Returning the correctness boolean and the message displayed to the user.

This is properly implemented by the following code copied from the default standard challenge:

```
@staticmethod
def attempt(challenge, request):
    data = request.form or request.get_json()
    submission = data['submission'].strip()
    flags = Flags.query.filter_by(challenge_id=challenge.id).all()
    for flag in flags:
        if get_flag_class(flag.type).compare(flag, submission):
            return True, 'Correct'
    return False, 'Incorrect'
```

CHAPTER 6

Contributing

6.1 Developing on CTFd

Developing new code for CTFd is easy and fun! CTFd is organized into components which each serve a distinct purpose.

6.1.1 Core Routes/Controllers

The core routes are identified in blueprints in the main CTFd folder for user facing routes and in the CTFd/admin folder for the admin panel. These core routes are used to display theme content and render the main server response that the user will receive.

6.1.2 API Routes/Controllers

The API routes are implemented in the `CTFd/api` folder as seperate blueprints for each type of resource used in CTFd. Most behavior that manipulates data should be implemented at the API level and seperated by method and resource level. The most common API methods are `GET`, `POST`, `PATCH`, `DELETE`.

6.1.3 Models

CTFd makes heavy usage of the SQLAlchemy ORM to access database contents. The core CTFd models are defined here. Plugins are however capable of adding their own models.

CTFd makes use of `alembic` and `Flask-Migrate` to perform database migrations between versions.

6.1.4 Schemas

Schemas provide an abstraction layer on top of the database models for permissioning and filtering of data. Schemas and the API work together to make distinctions about what data to show users and what data a user can edit.

6.1.5 Jinja2

Jinja2 is used by the CTFd server to render HTML content with data. In some cases (i.e. JavaScript challenge rendering), Mozilla Nunjucks templates are used as a mostly compatible alternative. Any templates written in Nunjucks must still be compatible with Jinja2.

6.1.6 JavaScript & CSS

JavaScript & CSS are used to style the front end of CTFd.

6.2 Linting

6.2.1 Python

Python code in CTFd is linted with *flake8* and *black*.

6.2.2 Javascript & CSS

JavaScript and CSS are linted with *prettier*.

Tip: The recommendation is to integrate all linters into your editor as your changes will fail to pass if the lint checks fail. See the `Makefile` for `make lint`

6.3 Testing

6.3.1 Python

Python tests are run using `pytest` on Travis. To run the test suite you can run *make test*. By default tests run against sqlite but this can be configured by setting the *TESTING_DATABASE_URL* environment variable.

CTFd will support both Python 2 and 3 until Python 2's EOL in 2020.

Tests are run in parallel with `pytest-xdist` and each test is run in its own database.

6.4 Documentation

CTFd's documentation is written using Sphinx and hosted by Read the Docs.

To build the documentation, you should go into the `docs` folder and run *make html*. The content output into the *docs/_build* folder will be the resulting hosted output.

6.5 Tips & Tricks

Typically while developing CTFd, developers use the provided `serve.py` script or its `make serve` wrapper and access CTFd at `http://localhost:4000`.

Very often you will need to generate testing data so that you can exercise CTFd's behavior. The included `populate.py` script will insert randomized testing data into the CTFd database.

The `export.py` script can be used to create a CTFd export on the command line.

The `import.py` script can be used to load in a CTFd export on the command line.

If you need to wipe CTFd completely, you should:

- delete the database (CTFd/ctfd.db by default)
- empty the cache. By default it will be stored in the `.data` folder if Redis is unavailable
- (optional) remove the contents of the *CTFd/uploads* folder

CHAPTER 7

Indices and tables

- genindex
- modindex
- search