**Chapter 17: Security**

The World Wide Web is a dangerous place. There are many bad actors and even more automated bots that will try to hack into your website and cause ill. Therefore understanding and implementing security features is a must in any website.

Fortunately, Django has a very strong record when it comes to security thanks to its years of experience handling web security issues as well as a robust and regular security update cycle.

However, as with any tool, it’s important to implement security features correctly and in this chapter we’ll cover how to do so in our Bookstore project.

**Social Engineering**

The biggest security risk to any website is ultimately not technical: it is people. The term social engineering refers to the technique of finding individuals with access to a system who will willingly or unwillingly share their login credentials with a bad actor.

These days phishing is probably the most likely culprit if you are in a technical organization. All it takes is one bad click on an email link for a malicious actor to potentially gain access to the system, or at least all the access the compromised employee has.

To mitigate this risk, implement a robust permissions scheme and only provide the exact security access an employee needs, not more. Does every engineer need access to the production database? Probably not. Do non-engineers need write access? Again, probably not. These are discussions best had up front and a good default is to only add permissions as needed, not to default to superuser status for everyone!

**Django updates**

Keeping your project up-to-date with the latest version of Django is another important way to stay secure. And I don’t just mean being current with the latest feature release (3.2, 4.0, 4.1, etc) which comes out roughly every 8 months. There are also monthly security patch updates that take the form of 4.0.1, 4.0.2, 4.0.3, etc.

What about **long-term support (LTS) releases**? Certain feature releases are designated as LTS and receive security and data loss fixes for a guaranteed period of time, usually around 3 years. For example, Django 3.2 LTS came out in April 2021 and will have extended support until April 2024. Django 4.2, the next LTS release, comes out in April 2023 and will have extended support until April 2026.

Can you stay on only LTS versions and not upgrade in the interim? Yes. Should you? No. It is better and far more secure to stay up-to-date with the latest version of Django if possible.

Resist the temptation and reality of many real-world projects which is not to devote a portion of developer time to staying current with Django versions. A website is like a car: it needs regular maintenance to run at its best. You are only compounding the problem if you put off updates.

How to update? Django features deprecation warnings that can and should be run for each new release by typing python -Wa manage.py test. It is far better to update from 3.0 to 3.1 to 3.2 and run the deprecation warnings each time rather than skipping

multiple versions.

**Deployment Checklist**

To assist with with deployment and checking security settings, the Django docs contain a dedicated deployment checklist that further describes security settings.

Even better there is a command we can run to automate Django’s recommendations, python manage.py check --deploy, that will check if a project is deployment ready. It uses the Django system check framework which can be used to customize similar commands in mature projects.

Since we are working in Docker we must prepend docker-compose exec web to the command though.

docker-compose exec web python manage.py check --deploy

System check identified some issues:

WARNINGS:

...

System check identified 6 issues (0 silenced).

How nice! A descriptive and lengthy list of issues which we can go through one-by-one to prepare our Bookstore project for production.

**docker-compose-prod.yml**

Ultimately, our local development settings will differ from our production settings. We already started to configure this back in Chapter 9: Environment Variables. Recall that we added environment variables for SECRET\_KEY, DEBUG, and DATABASES. But we did not set production values or a way to toggle efficiently between local and production.

There are a number of ways to tackle this challenge. Our approach will be to create a dedicated docker-compose-prod.yml file that we can use to test the production environment locally. When it comes down to deploy our project into production, we will manually update the environment variables to match.

To start, in the base directory create a docker-compose-prod.yml file. It will be located right next to the existing docker-compose.yml file. Then immediately add it to our exiting .gitignore file. We definitely do not want our production secrets to be stored in Git and available in the source code.

.gitignore

.venv

\_\_pycache\_\_/

db.sqlite3

.DS\_Store # Mac only

docker-compose-prod.yml

If you’re curious, Github maintains an official Python gitignore file containing additional configurations worthy of further exploration.

Run git status again and the docker-compose-prod.yml file is not visible, even though it is still in our project. That’s what we want!

For now, copy the contents of the docker-compose.yml file over into docker-compose-prod.yml.

docker-compose-prod.yml

services:

web:

build: .

command: python /code/manage.py runserver 0.0.0.0:8000

volumes:

- .:/code

ports:

- 8000:8000

depends\_on:

- db

environment:

- "DJANGO\_SECRET\_KEY=django-insecure-kdhq8uv91b4(ae==g+v6dd41)=re54dovd^1xzv+7d\*t6m0d=("

- "DJANGO\_DEBUG=True"

- "DATABASE\_URL=postgres://postgres@db/postgres"

db:

image: postgres:15

volumes:

- postgres\_data:/var/lib/postgresql/data/

environment:

- POSTGRES\_HOST\_AUTH\_METHOD=trus

volumes:

postgres\_data:

To run our new file, spin down the Docker container and restart it via the -f flag to specify an alternate compose file. By default, Docker assumes a docker-compose.yml so adding the -f flag is unnecessary in that case.

docker-compose down

docker-compose -f docker-compose-prod.yml up -d

Navigate to the website and everything should run as before, even though we are using a different compose file.

# docker-compose-prod.yml

environment:

- "DJANGO\_SECRET\_KEY=django-insecure-kdhq8uv91b4(ae==g+v6dd41)=re54dovd^1xzv+7d\*t6m0d=("

- "DJANGO\_DEBUG=False" # new

- "DATABASE\_URL=postgres://postgres@db/postgres"

Spin down the Docker container and start it up again after making the change to reload the changed environment variable.

docker-compose down

docker-compose -f docker-compose-prod.yml up -d

The website should run the same as before, but to check that DEBUG is set to False, visit a page that doesn’t exist like <http://127.0.0.1:8000/debug>.



Debug Page Not Found

And there is a generic “Not Found” message, confirming we have DEBUG set to False. Because if it were True, there would be a detailed error report instead.

Let’s run the Django deployment checklist again now that DEBUG has been changed. Recall that when we ran it earlier in the chapter there were 6 issues.

docker-compose exec web python manage.py check --deploy

System check identified some issues:

WARNINGS:

...

System check identified 5 issues (0 silenced).

We’re down to 5 since DEBUG is set to False. Progress!

**Defaults**

Environment variables serve two purposes in our Django project: they keep items like SECRET\_KEY actually secret and they act as a way to toggle between local and production settings. While there is nothing wrong with having two environment variables for a setting like DEBUG, it is arguably cleaner to use a default value when we don’t need to keep something secret.

For example, let’s rewrite the DEBUG configuration to look as follows:

# django\_project/settings.py

DEBUG = env.bool("DJANGO\_DEBUG", default=False # new

This means default to a production value of False if no environment variable is present. If there is one available, DJANGO\_DEBUG, then use that instead.

This approach of defaulting to the most secure, production-only settings is more secure because if for some reason environment variables are not loading in properly, we don’t want to the website to just use insecure local development variables. In other words, controls needed to be loosened to work rather than tightened. This is similar to the approach we took with our user permissions earlier in the book.

Go ahead now and update docker-compose-prod.yml by removing DJANGO\_DEBUG so that only the DJANGO\_SECRET\_KEY is under the environment section.

# docker-compose-prod.yml

environment:

- "DJANGO\_SECRET\_KEY=django-insecure-hv1(e0r@v4n4m6gqdz%dn(60o=dsy8&@0\_lbs8p-v3u^bs\

4)xl"

If you spin down Docker again with docker-compose down and restart the production version the website will still work properly.

docker-compose down

docker-compose -f docker-compose-prod.yml up -d

**SECRET\_KEY**

One of the five current issues that Django’s deployment checklist complains about is the SECRET\_KEY, which is randomly generated by Django each time the startproject command is run. It is used to provide cryptographic signing throughout your project and it is very, very important to keep secure.

Our current SECRET\_KEY is not secure because it has been added to the Git source control multiple times. Even if we moved its current value to an environment variable now, the value will live on in our Git history forever. So a new SECRET\_KEY must be generated and stored properly.It needs to have at least 50 characters, five or more unique characters, and not be prefexed with“django-insecure-“!

Luckily it is easy enough to generate a new random key. We can use Python’s built-in secrets module. The parameter token\_urlsafe returns the number of bytes in a URL-safe text string. With Base64 encoding on average each byte has 1.3 characters. So using 38 results in 51 characters in this case.

docker-compose exec web python -c "import secrets; print(secrets.token\_urlsafe(38))"

LHanzMtuuenbzKGXGBSHZYHXrydW3\_4zBeEE7WRtZGIrO0NNCbs

(mine going to be different)

|  |
| --- |
| A quick reminder that since we’re working with Docker, if your SECRET\_KEY includes a dollar sign, $, then you need to add an additional dollar sign, $$. This is due to how docker-compose handles variable substitution. Otherwise you will see an error! |

Add the new SECRET\_KEY to the docker-compose-prod.yml file so it looks as follows:

docker-compose-prod.yml

# docker-compose-prod.yml

environment:

- "DJANGO\_SECRET\_KEY=LHanzMtuuenbzKGXGBSHZYHXrydW3\_4z

BeEE7WRtZGIrO0NNCbs"

Because docker-compose-prod.yml is in our .gitignore file it will not appear in our source code.

Restart the Docker container which now uses a truly secret SECRET\_KEY.

docker-compose down

docker-compose -f docker-compose-prod.yml up -d

The website should work just as before. Run the deployment checklist again:

docker-compose exec web python manage.py check --deploy

System check identified some issues:

WARNINGS:

...

System check identified 4 issues (0 silenced).

We’re down to 4 issues now but first a brief dive into web security so we can understand why these settings are so important.

**Web Security**

Even though Django handles most common security issues by default, it is still vital to understand frequent attack methods and the steps Django takes to mitigate them. You can find an overview on the Django security page, but we’ll go into further depth here.

Django comes by default with a number of additional security middlewares that guard against other request/response cycle attacks.

A full explanation of each is beyond the scope of this book, but it is worth reading about theprotections provided by the Django security team over the years. Do not change the defaultswithout good cause.

**SQL injection**

Let’s start with a SQL injection attack which occurs when a malicious user can execute arbitrarySQL code on a database. Consider a log in form on a site. What happens if a malicious user instead types DELETE from users WHERE user\_id=user\_id? If this is run against the database without proper protections it could result in the deletion of all user records! Not good. This XKCD comic provides a humorous though potentially accurate example of how this can occur.

Fortunately the Django ORM automatically sanitizes user inputs by default when constructing querysets to prevent this type of attack. Where you need to be careful is that Django does provide the option to execute custom sql or raw queries. These should both be used with extreme caution since they could open up a vulnerability to SQL injection.

The non-profit Open Web Application Security Project (OWASP) has a fantastic and very detailed SQL Injection Cheat Sheet that is recommended for further reading.

**XSS (Cross Site Scripting)**

Cross-site scripting (XSS) is another classic attack that occurs when an attacker is able to inject small bits of code onto web pages viewed by other people. This code, typically JavaScript, if stored in the database will then be retrieved and displayed to other users.

For example, consider the form used for writing book reviews on our current site. What if instead of typing, “This book was great” a user typed something with JavaScript? For example, <script>alert('hello');</script>. If this script were stored on the database then every future user’s page would have a pop-up saying “hello”. While this particular example is more annoying than dangerous, a site vulnerable to XSS is very dangerous because a malicious user could insert any JavaScript into the page, including JavaScript that steals pretty much anything from an unsuspecting user.

To prevent an XSS attack Django templates automatically escape specific characters that are potentially dangerous including brackets (< and >), single quotes ', double quotes ", and the ampersand &. There are some edge cases where you might want to turn autoescape off but this should be used with extreme caution.

OWASP’s XSS Cheat Sheet is recommended for further reading.

**Cross-Site Request Forgery (CSRF)**

A Cross-Site Request Forgery (CSRF) is the third major type of attack but generally lesser known than SQL Injection or XSS. Fundamentally it exploits that trust a site has in a user’s web browser. When a user logs in to a website, let’s call it a banking website for illustration purposes, the server sends back a session token for that user. This is included in the HTTP Headers of all future requests and authenticates the user. But what happens if a malicious actor somehow obtains access to this session token?

For example, consider a user who logs into their bank in one browser tab. Then in another tab they open their email and click on an email link from a malicious actor. This link looks legitimate, but in fact it is pointing to the user’s bank which they are still logged into! So instead of leaving a blog comment on this fake site, behind the scenes the user’s credentials are used to transfer money from their account to the hacker’s account.

In practice there are multiple ways to obtain a user’s credentials via a CSRF attack, not just links, but hidden forms, special image tags, and even AJAX requests.

Django provides CSRF protection by including a random secret key both as a cookie via CSRF Middleware and in a form via the csrf\_token template tag. A 3rd party website will not have access to a user’s cookies and therefore any discrepancy between the two keys causes an error.

As ever, Django does allow customization: you can disable the CSRF middleware and use the csrf\_protect() template tag on specific views. However, undertake this step with extreme caution.

The OWASP CSRF Cheat Sheet provides a comprehensive look at the issue. Almost all major websites have been victims of CSRF attacks at some point in time.

A good rule of thumb is whenever you have a form on your site, think about whether you need to include the csrf\_token tag in it. Most of the time you will!

Clickjacking Protection

Clickjacking is yet another attack where a malicious site tricks a user into clicking on a hidden frame. An internal frame, known as an iframe, is commonly used to embed one website within another. For example, if you wanted to include a Google Map or YouTube video on your site you would include the iframe tag that puts that site within your own. This is very convenient.

But it has a security risk which is that a frame can be hidden from a user. Consider if a user is already logged into their Amazon account and then visits a malicious site that purports to be a picture of kittens. The user clicks on said malicious site to see more kittens, but in fact they

click an iFrame of an Amazon item that is unknowingly purchased. This is but one example of clickjacking.

To prevent against this Django comes with a default clickjacking middleware that sets a X-Frame- Options HTTP header that indicates whether a resource is allowed to load within a frame or iframe. You can turn this protection off, if desired, or even set it at a per view level. However, do so with a high degree of caution and research.

**HTTPS/SSL**

All modern websites should use HTTPS, which provides encrypted communication between a client and server. HTTP (Hypertext Transfer Protocol) is the backbone of the modern web, but it does not, by default, have encryption.

The “s” in HTTPS refers to its encrypted nature first due to SSL (Secure Sockets Layer) and these days its successor TLS (Transport Layer Security). With HTTPS enabled, which we will do in our deployment chapter, malicious actors can’t sniff the incoming and outgoing traffic for data like authentication credentials or API keys.

One of the 4 remaining issues in our Django deployment checklist is that SECURE\_SSL\_REDIRECT is currently set to False. For security reasons, it’s far better to force this to be True in production. Let’s change that now. At the bottom of the django\_project/settings.py file add a configuration for SECURE\_SSL\_REDIRECT: set it to either the environment variable DJANGO\_SECURE\_SSL\_REDIRECT

or the default value of True.

# django\_project/settings.py

SECURE\_SSL\_REDIRECT = env.bool("DJANGO\_SECURE\_SSL\_REDIRECT", default=True)

Then we will add the environment variable to docker-compose.yml so that for local development it defaults to False which is the less secure setting.

# docker-compose.yml

environment:

- "DJANGO\_SECRET\_KEY=django-insecure-hv1(e0r@v4n4m6gqdz%dn(60o=dsy8&@0\_lbs8p-v3u^bs4)xl"

- "DJANGO\_DEBUG=True"

- "DJANGO\_SECURE\_SSL\_REDIRECT=False" # new

Restart Docker and run the deployment checklist again.

docker-compose down

docker-compose -f docker-compose-prod.yml up -d

docker-compose exec web python manage.py check --deploy

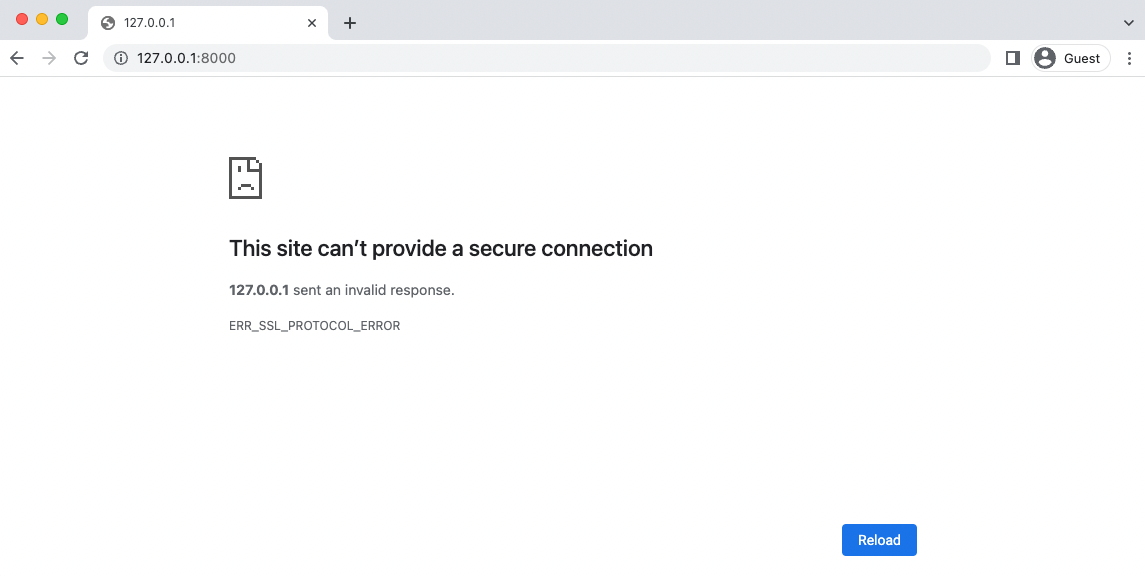
...

System check identified 3 issues (0 silenced).

We’re down to 3 issues now.

**SSL Error**

If you try to visit the website now there is an issue though.



SSL Error

It seems there is some sort of SSL Protocol Error, which makes sense because we just updated an SSL setting. Where could we find more info though? How about trying the logs:

docker-compose logs

...

You're accessing the development server over HTTPS, but it only supports HTTP.

Ah, that’s more descriptive. Since we switched the default over to True our website is automatically enforcing SLL. But the problem is that the local Django server only works over HTTP. We can run the deployment checklist to gain confidence in our production settings in docker-compose-prod.yml but in terms of actually using using the production site locally, we won’t be able to enforce SSL.

That’s ok though. It’s one of many reasons why we have separate local and production configurations. Companies typically actually have three different environments set up: one for local, one for production, and a staging server that mimics production but allows for more actual testing before switching things over completely. The staging server is often deployed but with a hidden URL and additional permissions so only team developers can try it out.

Going forward if you want to try out the local website with production settings be aware you will have to toggle off DJANGO\_SECURE\_SSL\_REDIRECT.

One more common gotcha is that even if you restart the Docker containers and start the local configuration, your web browser might still complain of an SSL error.

docker-compose down

docker-compose up -d

This is because most web browsers once they find out an HTTPS version of a site exists–even if it’s a localhost like http://127.0.0.1:8000/–will cache that and continually revert to it. You can look up different ways to clear the cache but the most effective in this case is to restart your web browser. As ever, when something isn’t working try turning it off and restarting.

**HTTP Strict Transport Security (HSTS)**

HTTP Strict Transport Security (HSTS) is a security policy that lets our server enforce that web browsers should only interact via HTTPS by adding a Strict-Transport-Security header.

There are three implicit HSTS configurations in our settings.py file that need to be updated for production:

• SECURE\_HSTS\_SECONDS = 0

• SECURE\_HSTS\_INCLUDE\_SUBDOMAINS = False

• SECURE\_HSTS\_PRELOAD = False

The SECURE\_HSTS\_SECONDS setting is set to 0 by default but the greater the better for security purposes. We will set it to the recommended 30 days, or 2,592,000 seconds, in our project.

SECURE\_HSTS\_INCLUDE\_SUBDOMAINS forces subdomains to use SSL. We will set it to True in production.

SECURE\_HSTS\_PRELOAD only has an effect when there is a non-zero value for SECURE\_HSTS\_SECONDS, but since we just set one, we’ll need to set this to True.

Add these three lines to the bottom of the django\_project/settings.py file.

# django\_project/settings.py

SECURE\_HSTS\_SECONDS = env.int("DJANGO\_SECURE\_HSTS\_SECONDS", default=2592000) # 30 days

SECURE\_HSTS\_INCLUDE\_SUBDOMAINS = env.bool("DJANGO\_SECURE\_HSTS\_INCLUDE\_SUBDOMAINS", default=True)

SECURE\_HSTS\_PRELOAD = env.bool("DJANGO\_SECURE\_HSTS\_PRELOAD", default=True)

Then update docker-compose.yml with the local development values.

# docker-compose.yml

environment:

- "DJANGO\_SECRET\_KEY=django-insecure-hv1(e0r@v4n4m6gqdz%dn(60o=dsy8&@0\_lbs8p-v3u^bs4)xl"

- "DJANGO\_DEBUG=True"

- "DJANGO\_SECURE\_SSL\_REDIRECT=False"

- "DJANGO\_SECURE\_HSTS\_SECONDS=0" # new

- "DJANGO\_SECURE\_HSTS\_INCLUDE\_SUBDOMAINS=False" # new

- "DJANGO\_SECURE\_HSTS\_PRELOAD=False" # new

Restart Docker and run the deployment checklist again.

docker-compose down

docker-compose -f docker-compose-prod.yml up -d

docker-compose exec web python manage.py check --deploy

...

System check identified 2 issues (0 silenced).

Only 2 issues left!

**Secure Cookies**

An HTTP Cookie is used to store information on a client’s computer such as authentication credentials. This is necessary because the HTTP protocol is stateless by design: there’s no way to tell if a user is authenticated other than including an identifier in the HTTP Header!

Django uses sessions and cookies for this, as do most websites. But cookies can and should be forced over HTTPS as well via the SESSION\_COOKIE\_SECURE config. By default Django sets this value to False for local development; in production it needs to be True.

The second issue is CSRF\_COOKIE\_SECURE, which defaults to False but in production should be True so that only cookies marked as “secure” will be sent with an HTTPS connection.

Add these two lines again to the bottom of django\_project/settings.py.

django\_project/settings.py

SESSION\_COOKIE\_SECURE = env.bool("DJANGO\_SESSION\_COOKIE\_SECURE", default=True)

CSRF\_COOKIE\_SECURE = env.bool("DJANGO\_CSRF\_COOKIE\_SECURE", default=True)

Then update the docker-compose.yml file.

# docker-compose.yml

environment:

- "DJANGO\_SECRET\_KEY=django-insecure-kdhq8uv91b4(ae==g+v6dd41)=re54dovd^1xzv+7d\*t6m0d=("

- "DJANGO\_DEBUG=True"

- "DJANGO\_SECURE\_SSL\_REDIRECT=False"

- "DJANGO\_SECURE\_HSTS\_SECOND=0"

- "DJANGO\_SECURE\_HSTS\_INCLUDE\_SUBDOMAINS=False"

- "DJANGO\_SECURE\_HSTS\_PRELOAD=False"

- "DJANGO\_SESSION\_COOKIE\_SECURE=False" # new

- "DJANGO\_CSRF\_COOKIE\_SECURE=False" # new

- "DATABASE\_URL=postgres://postgres@db/postgres"

Restart Docker and run the deployment checklist again.

docker-compose down

docker-compose -f docker-compose-prod.yml up -d

docker-compose exec web python manage.py check --deploy

System check identified no issues (0 silenced).

No more issues. Woohoo!

**Admin Hardening**

So far it may seem as though the general security advice is to rely on Django defaults, use HTTPS, add csrf\_token tags on forms, and set a permissions structure. All true. But one additional step Django does not take on our behalf is hardening the Django admin.

Consider that every Django website sets the admin, by default, to the /admin URL. This is a prime suspect for any hacker trying to access a Django site. Therefore, an easy step is to simply change the admin URL to literally anything else! Open up and change the URL path. In this example it is anything-but-admin/.

**Git**

There’s been a lot of code changes in this chapter so make sure to commit everything with Git.

Remove-Item -Recurse -Force .git

git init

git status

git add .

git commit -m “Chapter 17. Security”

If you have any errors make sure to look at your logs with docker-compose logs and compare your code with the official source code on Github

**Conclusion**

Security is a major concern for any website. By using a docker-compose-prod.yml file we can accurately test, within Docker, our production settings before deploying the site live. And by using default values we can both simplify the environment variables in the file as well as ensure that if something goes awry with environment variables we will default to secure production values. Django comes with many built-in security features and with the addition of the deployment checklist we can now deploy our site now with a high degree of confidence that it is secure.

Ultimately, security is a constant battle and while the steps in this chapter cover most areas of concern, keeping your website up-to-date with the latest Django version is vital for continued safety.

The end.