8 Application-level fault injection

This chapter covers

- Building chaos engineering capabilities directly into your application
- Ensuring that the extra code doesn't affect the application's performance
- More advanced usage of Apache Bench

So far, you've learned a variety of ways to apply chaos engineering to a selection of different systems. The languages, tools, and approaches varied, but they all had one thing in common: working with source code outside your control. If you're in a role like SRE or platform engineer, that's going to be your bread and butter. But sometimes you will have the luxury of applying chaos engineering to your own code.

This chapter focuses on baking chaos engineering options directly into your application for a quick, easy, and—dare I say it—fun way of increasing your confidence in the overall stability of the system as a whole. I'll guide you through designing and running two experiments: one injecting latency into functions responsible for communicating with an external cache, and another injecting intermittent failure through the simple means of raising an exception. The example code is written in Python, but don't worry if it's not your forte; I promise to keep it basic.

NOTE I chose Python for this chapter because it hovers at the top of the list in terms of popularity, and it allows for short, expressive examples. But what you learn here is universal and can be leveraged in any language. Yes, even Node.js.

If you like the sound of it, let's go for it. First things first: a scenario.

8.1 Scenario

Let's say that you work for an e-commerce company and you're designing a system for recommending new products to your customers, based on their previous queries. As a practitioner of chaos engineering, you're excited: this might be a perfect opportunity to add features allowing you to inject failure directly into the codebase.

To generate recommendations, you need to be able to track the queries your customers make, even if they are not logged in. The e-commerce store is a website, so you decide to simply use a cookie (https://en.wikipedia.org/wiki/HTTP_cookie) to store a session ID for each new user. This allows you to distinguish between the requests and attribute each search query to a particular session.

In your line of work, latency is important; if the website doesn't feel quick and responsive to customers, they will buy from your competitors. The latency therefore influences some of the implementation choices and becomes one of the targets for chaos experiments. To minimize the latency added by your system, you decide to use an in-memory key-value store, Redis (https://redis.io/), as your session cache and store only the last three queries the user made. These previous queries are then fed to the recommendation engine every time the user searches for a product, and come back with potentially interesting products to display in a You Might Be Interested In box.

So here's how it all works together. When a customer visits your e-commerce website, the system checks whether a session ID is already stored in a cookie in the browser. If it's not, a random session ID is generated and stored. As the customer searches through the website, the last three queries are saved in the session cache, and are used to generate a list of recommended products that is then presented to the user in the search results.

For example, after the first search query of "apple," the system might recommend "apple juice." After the second query for "laptop," given that the two consecutive queries were "apple" and "laptop," the system might recommend a "macbook pro." If you've worked in e-commerce before, you know this is a form of cross-selling (https://en.wikipedia.org/wiki/Cross-selling), a serious and powerful technique used by most online stores and beyond. Figure 8.1 summarizes this process.

Learning how to implement this system is not the point of this chapter. What I'm aiming at here is to show you a concrete, realistic example of how you can add minimal code directly into the application to make running chaos experiments on it easy. To do that, let me first walk you through a simple implementation of this system, for

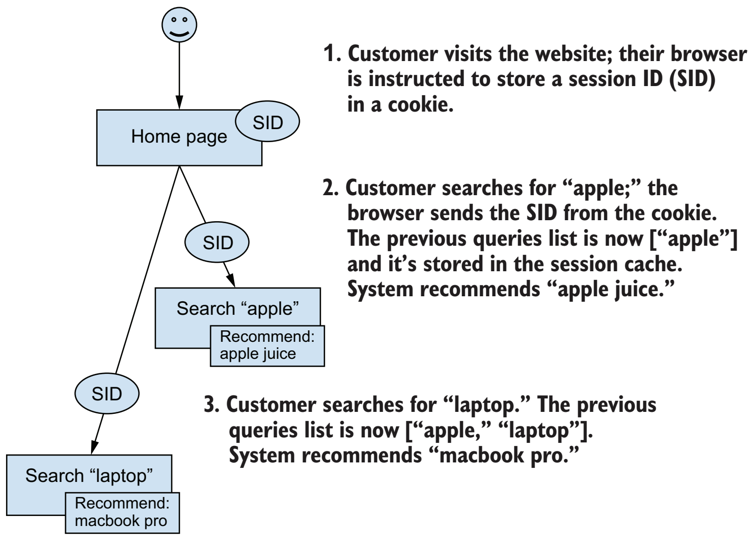
Home page
SID
1. Customer visits the website; their browser is instructed to store a session ID (SID) in a cookie.
SID
Search "apple"
Recommend: apple juice
2. Customer searches for "apple;" the browser sends the SID from the cookie. The previous queries list is now ["apple"] and it's stored in the session cache. System recommends "apple juice."
SID
Search "laptop"
Recommend: macbook pro
3. Customer searches for "laptop." The previous queries list is now ["apple," "laptop"]. System recommends "macbook pro."

Figure 8.1 High-level overview of the session-tracking system

now without any chaos engineering changes, and then, once you're comfortable with it, I'll walk you through the process of building two chaos experiments into it.

8.1.1 Implementation details: Before chaos

I'm providing you with a bare-bones implementation of the relevant parts of this website, written in Python and using the Flask HTTP framework (https://flask.palletsprojects.com/). If you don't know Flask, don't worry; we'll walk through the implementation to make sure everything is clear.

Inside your VM, the source code can be found in ~/src/examples/app (for installation instructions outside the VM, refer to appendix A). The code doesn't implement any chaos experiments quite yet; we'll add that together. The main file, app.py, provides a single HTTP server, exposing three endpoints:

- Index page (at /) that displays the search form and sets the session ID cookie.
- Search page (at /search) that stores the queries in the session cache and displays the recommendations.
- Reset page (at /reset) that replaces the session ID cookie with a new one to make testing easier for you. (This endpoint is for your convenience only.)

Let's start with the index page route, the first one any customer will see. It's implemented in the `index` function and does exactly two things: returns some static HTML to render the search form, and sets a new session ID cookie, through the `set_session_id` function. The latter is made easy through Flask's built-in method of accessing cookies (`flask.request.cookies.get`) as well as setting new ones (`response.set_cookie`). After visiting this endpoint, the browser stores the random unique ID (UID) value in the `sessionID` cookie, and it sends that value with every subsequent request to the same host. That's how the system is able to attribute the further actions to a session ID. If you're not familiar with Flask, the `@app.route("/")` decorator tells Flask to serve the decorated function (in this case `index`) under the / endpoint.

Next, the search page is where the magic happens. It's implemented in the `search` function, decorated with `@app.route("/search", methods=["POST", "GET"])`, meaning that both GET and POST requests to /search will be routed to it. It reads the session ID from the cookie, the query sent from the search form on the home page (if any), and stores the query for that session by using the `store_interests` function. `store_interests` reads the previous queries from Redis, appends the new one, stores it back, and returns the new list of interests. Using that new list of interests, it calls the `recommend_other_products` function, that—for simplicity—returns a hardcoded list of products. Figure 8.2 summarizes this process.

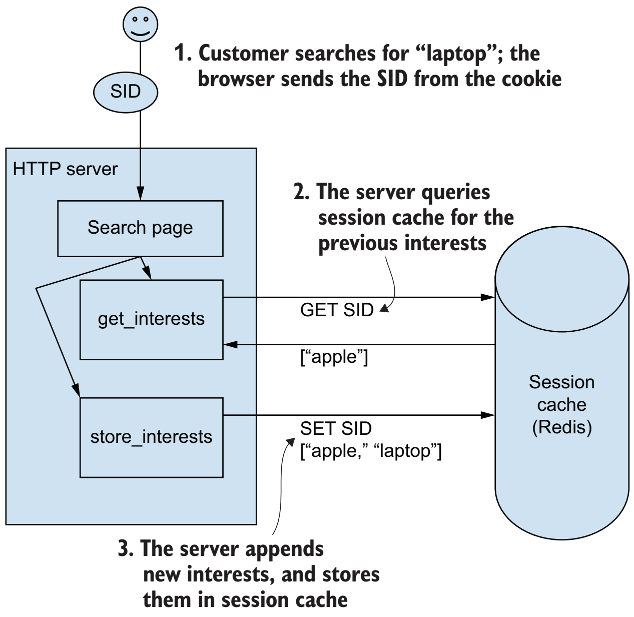
1. Customer searches for "laptop"; the browser sends the SID from the cookie
SID
HTTP server
Search page
get_interests
store_interests
2. The server queries session cache for the previous interests
GET SID
["apple"]
SET SID ["apple," "laptop"]
Session cache (Redis)
3. The server appends new interests, and stores them in session cache

Figure 8.2 Search page and session cache interactions

When that's done, the `search` function renders an HTML page presenting the search results as well as the recommended items. Finally, the third endpoint, implemented in the `reset` function, replaces the session ID cookie with a new, random one and redirects the user to the home page.

The following listing provides the full source code for this application. For now, ignore the commented out section on chaos experiments.

Listing 8.1 app.py

```
import uuid, json, redis, flask

COOKIE_NAME = "sessionID"

def get_session_id():
    """ Read session id from cookies, if present """
    return flask.request.cookies.get(COOKIE_NAME)

def set_session_id(response, override=False):
    """ Store session id in a cookie """
    session_id = get_session_id()
    if not session_id or override:
        session_id = uuid.uuid4()
    response.set_cookie(COOKIE_NAME, str(session_id))


CACHE_CLIENT = redis.Redis(host="localhost", port=6379, db=0)

# Chaos experiment 1 - uncomment this to add latency to Redis access
#import chaos
#CACHE_CLIENT = chaos.attach_chaos_if_enabled(CACHE_CLIENT)

# Chaos experiment 2 - uncomment this to raise an exception every other call
#import chaos2
#@chaos2.raise_rediserror_every_other_time_if_enabled
def get_interests(session):
    """ Retrieve interests stored in the cache for the session id """
    return json.loads(CACHE_CLIENT.get(session) or "[]")

def store_interests(session, query):
    """ Store last three queries in the cache backend """
    stored = get_interests(session)
    if query and query not in stored:
        stored.append(query)
    stored = stored[-3:]
    CACHE_CLIENT.set(session, json.dumps(stored))
    return stored


def recommend_other_products(query, interests):
    """ Return a list of recommended products for a user,
    based on interests """
    if interests:
        return {"this amazing product":
     "https://youtube.com/watch?v=dQw4w9WgXcQ"}
    return {}


app = flask.Flask(__name__)

@app.route("/")
def index():
    """ Handle the home page, search form """
    resp = flask.make_response("""
    <html><body>
        <form action="/search" method="POST">
            <p><h3>What would you like to buy today?</h3></p>
            <p><input type='text' name='query'/>
            <input type='submit' value='Search'/></p>
        </form>
        <p><a href="/search">Recommendations</a>. <a href="/reset">Reset</a>.
     </p>
    </body></html>
    """)
    set_session_id(resp)
    return resp

@app.route("/search", methods=["POST", "GET"])
def search():
    """ Handle search, suggest other products """
    session_id = get_session_id()
    query = flask.request.form.get("query")
    try:
        new_interests = store_interests(session_id, query)
    except redis.exceptions.RedisError as exc:
        print("LOG: redis error %s", str(exc))
        new_interests = None
    recommendations = recommend_other_products(query, new_interests)
    return flask.make_response(flask.render_template_string("""
    <html><body>
        {% if query %}<h3>I didn't find anything for "{{ query }}"</h3>{%
     endif %}
        <p>Since you're interested in {{ new_interests }}, why don't you
     try...
        {% for k, v in recommendations.items() %} <a href="{{ v }}">{{ k
     }}</a>{% endfor %}!</p>
        <p>Session ID: {{ session_id }}. <a href="/">Go back.</a></p>
    </body></html>
    """,
        session_id=session_id,
        query=query,
        new_interests=new_interests,
        recommendations=recommendations,
    ))

@app.route("/reset")
def reset():
    """ Reset the session ID cookie """
    resp = flask.make_response(flask.redirect("/"))
    set_session_id(resp, override=True)
    return resp
```

Let’s now see how to start the application. It has two external dependencies:

- Flask (https://flask.palletsprojects.com/)
- redis-py (https://github.com/andymccurdy/redis-py)

You can install both in the versions that were tested with this book by running the following command in your terminal window:

```
sudo pip3 install redis==3.5.3 Flask==1.1.2
```

You also need an actual instance of Redis running on the same host, listening for new connections on the default port 6379. If you're using the VM, Redis is preinstalled (consult appendix A for installation instructions if you're not using the VM). Open another terminal window, and start a Redis server by running the following command:

```
redis-server
```

You will see the characteristic output of Redis, similar to the following:

```
54608:C 28 Jun 2020 18:32:12.616 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
54608:C 28 Jun 2020 18:32:12.616 # Redis version=6.0.5, bits=64,
     commit=00000000, modified=0, pid=54608, just started
54608:C 28 Jun 2020 18:32:12.616 # Warning: no config file specified, using the
     default config. In order to specify a config file use ./redis-server
     /path/to/redis.conf
54608:M 28 Jun 2020 18:32:12.618 * Increased maximum number of open files to
     10032 (it was originally set to 8192).
                _._
           _.-``__ ''-._
      _.-``    `.  `_.  ''-._           Redis 6.0.5 (00000000/0) 64 bit
  .-`` .-```.  ```\/    _.,_ ''-._
 (    '      ,       .-`  | `,    )     Running in standalone mode
 |`-._`-...-` __...-.``-._|'` _.-'|     Port: 6379
 |    `-._   `._    /     _.-'    |     PID: 54608
  `-._    `-._  `-./  _.-'    _.-'
 |`-._`-._    `-.__.-'    _.-'_.-'|
 |    `-._`-._        _.-'_.-'    |           http://redis.io
  `-._    `-._`-.__.-'_.-'    _.-'
 |`-._`-._    `-.__.-'    _.-'_.-'|
 |    `-._`-._        _.-'_.-'    |
  `-._    `-._`-.__.-'_.-'    _.-'
      `-._    `-.__.-'    _.-'
          `-._        _.-'
              `-.__.-'
```

With that, you are ready to start the application! While Redis is running in the second terminal window, go back to the first one and run the following command, still from ~/src/examples/app. It will start the application in development mode, with detailed error stacktraces and automatic reload on changes to the source code:

```
cd ~/src/examples/app              ❶
FLASK_ENV=development \            ❷
FLASK_APP=app.py \                 ❸
    python3 -m flask run           ❹
```

❶ **Goes to the location with the source code of the application**
❷ **Specifies development environment for easier debugging and auto-reload**
❸ **Specifies FLASK_APP environment variable, which points Flask to run the application**
❹ **Runs the flask module, specifying run command to start a web server**

The application will start, and you'll see output just like the following, specifying the app it's running, the host and port where the application is accessible, and the environment (all in bold font):

```
 * Serving Flask app "app.py" (lazy loading)
 * Environment: development
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 289-495-131
```

You can now browse to http://127.0.0.1:5000/ to confirm it's working. You will see a simple search form, asking you to type the name of the product you're interested in. Try searching for "apple." You are taken to a second page, where you will be able to see your previous queries as well as the recommendations. Be absolutely sure to click the recommendations; they are great! If you repeat this process a few times, you will notice that the page retains the last three search queries. Finally, note that the page also prints the session ID, and if you're curious, you can see it in the cookies section in your browser.

OK, so now you have a simple, yet functional application that we'll pretend you wrote. Time to have some fun with it! Let's do some chaos engineering.

8.2 Experiment 1: Redis latency

In the e-commerce store scenario I described at the beginning of the chapter, the overall latency of the website is paramount: you know that if you slow the system down too much, customers will start leaving the website and buying from your competitors. It's therefore important that you understand how the latency communicating with the session cache (Redis) affects the overall speed of the website. And that's where chaos engineering shines: we can simulate some latency and measure how much it affects the system as a whole.

You have injected latency before in different ways. In chapter 4, you used Traffic Control (`tc`) to add latency to a database, and in chapter 5 you leveraged Docker and Pumba to do the same. So how is this different this time? In the previous scenarios, we tried hard to modify the behavior of the system without modifying the source code. This time, I want to add to that by showing you how easy it is to add chaos engineering when you *are* in control of the application's design. Everyone can do that—you just need to have a little bit of imagination! Let's design a simple experiment around the latency.

8.2.1 Experiment 1 plan

In the example application, it's easy to establish that for each request, the session cache is accessed twice: first to read the previous queries, and second to store the new set. You can therefore hypothesize that you will see a double of any latency added to the Redis calls in the overall latency figure for the website.

Let's find out whether that's true. By now, you're well versed in using Apache Bench (`ab`) for generating traffic and observing latencies, so let's leverage that once again. Here's one possible version of a chaos experiment that will help test that theory:

1. Observability: generate traffic and observe the latency by using `ab`.
2. Steady state: observe latency without any chaos changes.
3. Hypothesis: if you add a 100 ms latency to each interaction with the session cache (reads and writes), the overall latency of the /search page should increase by 200 ms.
4. Run the experiment!

That's it! Now, all you need to do is follow this plan, starting with the steady state.

8.2.2 Experiment 1 steady state

So far, you've used `ab` to generate GET requests. This time, you have a good opportunity to learn how to use it to send POST requests, like the ones sent from the search form on the index page that the browser sends to the /search page. To do that, you need to do the following:

1. Use the POST method, instead of GET.
2. Use the `Content-type` header to specify the value used by the browser when sending an HTML form (`application/x-www-form-urlencoded`).
3. Pass the actual form data as the body of the request to simulate the value from a form.
4. Pass the session ID (you can make it up) in a cookie in another header, just as the browser does with every request.

Fortunately, this all can be done with `ab` by using the following arguments:

- `-H "Header: value"` to set custom headers, one for the cookie with the session ID and one for the content type. This flag can be used multiple times to set multiple headers.
- `-p post-file` to send the contents of the specified file as the body of the request. It also automatically assumes the POST method. That file needs to follow the HTML form format, but don't worry if you don't know it. In this simple use case, I'll show you a body you can use: `query=TEST` to query for "TEST." The actual query in this case doesn't matter.

Putting this all together, and using our typical concurrency of 1 (`-c 1`) and runtime of 10 seconds (`-t 10`), you end up with the following command. Assuming that the server is still running, open another terminal window and run the following:

```
echo "query=Apples" > query.txt                                   ❶
ab -c 1 -t 10 \
    -H "Cookie: sessionID=something" \                            ❷
    -H "Content-type: application/x-www-form-urlencoded" \        ❸
    -p query.txt \                                                ❹
    http://127.0.0.1:5000/search
```

1. **Creates a simple file with the query content**
2. **Sends a header with the cookie specifying the sessionID**
3. **Sends a header specifying the content type to a simple HTML form**
4. **Uses the previously created file with the simple query in it**

You will see the familiar output of `ab`, similar to the following (abbreviated). My VM managed to do 1673 requests, or about 167 requests per second (5.98 ms per request) with no errors (all four in bold font):

```
Server Software:        Werkzeug/1.0.1
Server Hostname:        127.0.0.1
Server Port:            5000
(...)
Complete requests:      1673
Failed requests:        0
(...)
Requests per second:    167.27 [#/sec] (mean)
Time per request:       5.978 [ms] (mean)
```

So far, so good. These numbers represent your steady state, the baseline. Let's implement some actual chaos and see how these change.

8.2.3 Experiment 1 implementation

It's time to implement the core of your experiment. This is the cool part: because you own the code, there are a million and one ways of implementing the chaos experiment, and you're free to pick whichever works best for you! I'm going to guide you through just one example of what that could look like, focusing on three things:

- Keep it simple.
- Make the chaos experiment parts optional for your application and disabled by default.
- Be mindful of the performance impact the extra code has on the whole application.

These are good guidelines for any chaos experiments, but as I said before, you will pick the right implementation based on the actual application you're working on. This example application relies on a Redis client accessible through the `CACHE_CLIENT` variable, and then the two functions using it, `get_interests` and `store_interest`, use the `get` and `set` methods on that cache client, respectively (all in bold font):

```
CACHE_CLIENT = redis.Redis(host="localhost", port=6379, db=0)              1

def get_interests(session):
    """ Retrieve interests stored in the cache for the session id """
    return json.loads(CACHE_CLIENT.get(session) or "[]")                    2

def store_interests(session, query):
    """ Store last three queries in the cache backend """
    stored = get_interests(session)
    if query and query not in stored:
        stored.append(query)
    stored = stored[-3:]
    CACHE_CLIENT.set(session, json.dumps(stored))                   ❸
    return stored
```

❶ **An instance of Redis client is created and accessible through the CACHE_CLIENT variable.**

❷ **get_interests is using the get method of CACHE_CLIENT.**

❸ **stores_interests is using the set method of CACHE_CLIENT (and get by transition, through the call to get_interests).**

All you need to do to implement the experiment is to modify `CACHE_CLIENT` to inject latency into both of the `get` and `set` methods. There are plenty of ways of doing that, but the one I suggest is to write a simple wrapper class.

The wrapper class would have the two required methods (`get` and `set`) and rely on the wrapped class for the actual logic. Before calling the wrapped class, it would sleep for the desired time. And then, based on an environment variable, you'd need to optionally replace `CACHE_CLIENT` with an instance of the wrapper class.

Still with me? I prepared a simple wrapper class for you (`ChaosClient`), along with a function to attach it (`attach_chaos_if_enabled`) in another file called chaos.py, in the same folder (~/src/examples/app). The `attach_chaos_if_enabled` function is written in a way so as to inject the experiment only if an environment variable called `CHAOS` is set. That's to satisfy the "disabled by default" expectation. The amount of time to inject is controlled by another environment variable called `CHAOS_DELAY_SECONDS` and defaults to 750 ms. The following listing is an example implementation.

Listing 8.2 chaos.py

```
import time
import os


class ChaosClient:
    def __init__(self, client, delay):
        self.client = client                                    ❶
        self.delay = delay
    def get(self, *args, **kwargs):                             ❷
        time.sleep(self.delay)                                  ❸
        return self.client.get(*args, **kwargs)
    def set(self, *args, **kwargs):                             ❹
        time.sleep(self.delay)
        return self.client.set(*args, **kwargs)

def attach_chaos_if_enabled(cache_client):
    """ creates a wrapper class that delays calls to get and set methods """
    if os.environ.get("CHAOS"):                                 ❺
        return ChaosClient(cache_client,
     float(os.environ.get("CHAOS_DELAY_SECONDS", 0.75)))
    return cache_client
```

1. **The wrapper class stores a reference to the original cache client.**
2. **The wrapper class provides the get method, expected on the cache client, that wraps the client's method of the same name.**
3. **Before the method relays to the original get method, it waits for a certain amount of time.**
4. **The wrapper class also provides the set method, exactly like the get method.**
5. **Returns the wrapper class only if the CHAOS environment variable is set**

Now, equipped with this, you can modify the application (app.py) to make use of this new functionality. You can import it and use it to conditionally replace `CACHE_CLIENT`, provided that the right environment is set. All you need to do is find the line where you instantiate the cache client inside the app.py file:

```
CACHE_CLIENT = redis.Redis(host="localhost", port=6379, db=0)
```

Add two lines after it, importing and calling the `attach_chaos_if_enabled` function, passing the `CACHE_CLIENT` variable as an argument. Together, they will look like the following:

```
CACHE_CLIENT = redis.Redis(host="localhost", port=6379, db=0)
import chaos
CACHE_CLIENT = chaos.attach_chaos_if_enabled(CACHE_CLIENT)
```

With that, the scene is set and ready for the grand finale. Let's run the experiment!

8.2.4 Experiment 1 execution

To activate the chaos experiment, you need to restart the application with the new environment variables. You can do that by stopping the previously run instance (press Ctrl-C) and running the following command:

```
CHAOS=true \                     ❶
CHAOS_DELAY_SECONDS=0.1 \        ❷
FLASK_ENV=development \          ❸
FLASK_APP=app.py \               ❹
python3 -m flask run             ❺
```

1. **Activates the conditional chaos experiment code by setting the CHAOS environment variable**
2. **Specifies chaos delay injected as 0.1 second, or 100 ms**
3. **Specifies the Flask development env for better error messages**
4. **Specifies the same app.py application**
5. **Runs Flask**

Once the application is up and running, you're good to go to rerun the same `ab` command you used to establish the steady state once again. To do that, run the following command in another terminal window:

```
echo "query=Apples" > query.txt && \                          ❶
ab -c 1 -t 10 \
    -H "Cookie: sessionID=something" \                        ❷
    -H "Content-type: application/x-www-form-urlencoded" \    ❸
    -p query.txt \                                                   ❹
    http://127.0.0.1:5000/search
```

❶ **Creates a simple file with the query content**
❷ **Sends a header with the cookie specifying the sessionID**
❸ **Sends a header specifying the content type to a simple HTML form**
❹ **Uses the previously created file with the simple query in it**

After the 10-second wait, when the dust settles, you will see the `ab` output, much like the following. This time, my setup managed to complete only 48 requests (208 ms per request), still without errors (all three in bold font):

```
(...)
Complete requests:      48
Failed requests:        0
(...)
Requests per second:    4.80 [#/sec] (mean)
Time per request:       208.395 [ms] (mean)
(...)
```

That's consistent with our expectations. The initial hypothesis was that adding 100 ms to every interaction with the session cache should result in an extra 200 ms additional latency overall. And as it turns out, for once, our hypothesis was correct! It took a few chapters, but that's a bucket list item checked off! Now, before we get too narcissistic, let's discuss a few pros and cons of running chaos experiments this way.

8.2.5 Experiment 1 discussion

Adding chaos engineering code directly to the source code of the application is a double-edged sword: it's often easier to do, but it also increases the scope of things that can go wrong. For example, if your code introduces a bug that breaks your program, instead of increasing the confidence in the system, you've decreased it. Or, if you added latency to the wrong part of the codebase, your experiments might yield results that don't match reality, giving you false confidence (which is arguably even worse).

You might also think, "Duh, I added code to sleep for *X* seconds; of course it's slowed down by that amount." And yes, you're right. But now imagine that this application is larger than the few dozen lines we looked at. It might be much harder to be sure about how latencies in different components affect the system as a whole. But if the argument of human fallibility doesn't convince you, here's a more pragmatic one: doing an experiment and confirming even the simple assumptions is often quicker than analyzing the results and reaching meaningful conclusions.

I'm also sure you noticed that reading and writing to Redis in two separate actions is not going to work with any kind of concurrent access and can lose writes. Instead, it could be implemented using a Redis set and atomic add operation, fixing this problem as well as the double penalty for any network latency. My focus here was to keep it as simple as possible, but thanks for pointing that out!

Finally, there is always the question of performance: if you add extra code to the application, you might make it slower. Fortunately, because you are free to write the code whatever way you please, there are ways around that. In the preceding example, the extra code is applied only if the corresponding environment variables are set during startup. Apart from the extra `if` statement, there is no overhead when running the application without the chaos experiment. And when it's on, the penalty is the cost of an extra function call to our wrapper class. Given that we're waiting for times at a scale of milliseconds, that overhead is negligible.

That's what my lawyers advised me to tell you, anyway. With all these caveats out of the way, let's do another experiment, this time injecting failure, rather than slowness.

8.3 Experiment 2: Failing requests

Let's focus on what happens when things fail rather than slow down. Let's take a look at the function `get_interests` again. As a reminder, it looks like the following. (Note that there is no exception handling whatsoever.) If the `CACHE_CLIENT` throws any exceptions (bold font), they will just bubble up further up the stack:

```
def get_interests(session):
    """ Retrieve interests stored in the cache for the session id """
    return json.loads(CACHE_CLIENT.get(session) or "[]")
```

To test the exception handling of this function, you'd typically write unit tests and aim to cover all legal exceptions that can be thrown. That will cover this bit, but will tell you little about how the entire application behaves when these exceptions arise. To test the whole application, you'd need to set up some kind of *integration* or *end-to-end* (*e2e*) tests, whereby an instance of the application is stood up along with its dependencies, and some client traffic is created. By working on that level, you can verify things from the user's perspective (what error will the user see, as opposed to what kind of exception some underlying function returns), test for regressions, and more. It's another step toward reliable software.

And this is where applying chaos engineering can create even more value. You can think of it as the next step in that evolution—a kind of end-to-end testing, while injecting failure into the system to verify that the whole reacts the way you expect. Let me show you what I mean: let's design another experiment to test whether an exception in the `get_interests` function is handled in a reasonable manner.

8.3.1 Experiment 2 plan

What should happen if `get_interests` receives an exception when trying to read from the session store? That depends on the type of page you're serving. For example, if you're using that session date to list recommendations in a sidebar to the results of a search query, it might make more economic sense to skip the sidebar and allow the user to at least click on other products. If, on the other hand, we are talking about the checkout page, then not being able to access the session data might make it impossible to finish the transaction, so it makes sense to return an error and ask the user to try again.

In our case, we don't even have a buy page, so let's focus on the first type of scenario: if the `get_interests` function throws an exception, it will bubble up in the `store_interests` function, which is called from our search website with the following code. Note the `except` block, which catches `RedisError`, the type of error that might be thrown by our session cache client (in bold font):

```
try:
    new_interests = store_interests(session_id, query)
except redis.exceptions.RedisError:                     ❶
    print("LOG: redis error %s", str(exc))
    new_interests = None
```

❶ **The type of exception thrown by the Redis client you use is caught and logged here.**

That error handling should result in the exception in `get_interests` being transparent to the user; they just won't see any recommendations. You can create a simple experiment to test that out:

1. Observability: browse to the application and see the recommended products.
2. Steady state: the recommended products are displayed in the search results.
3. Hypothesis: if you add a `redis.exceptions.RedisError` exception every other time `get_interests` is called, you should see the recommended products every other time you refresh the page.
4. Run the experiment!

You've already seen that the recommended products are there, so you can jump directly to the implementation!

8.3.2 Experiment 2 implementation

Similar to the first experiment, there are plenty of ways to implement this. And just as in the first experiment, let me suggest a simple example. Since we're using Python, let's write a simple decorator that we can apply to the `get_interests` function. As before, you want to activate this behavior only when the `CHAOS` environment variable is set.

I prepared another file in the same folder, called chaos2.py, that implements a single function, `raise_rediserror_every_other_time_if_enabled`, that's designed to be used as a Python decorator (https://wiki.python.org/moin/PythonDecorators). This rather verbosely named function takes another function as a parameter and implements the desired logic: return the function if the chaos experiment is not active, and return a wrapper function if it is active. The wrapper function tracks the number of times it's called and raises an exception on every other call. On the other calls, it relays to the original function with no modifications. The following listing provides the source code of one possible implementation.

Listing 8.3 chaos2.py

```
import os
import redis
```

```
def raise_rediserror_every_other_time_if_enabled(func):
    """ Decorator, raises an exception every other call to the wrapped
     function """
    if not os.environ.get("CHAOS"):
        return func                                                 ❶
    counter = 0
    def wrapped(*args, **kwargs):
        nonlocal counter
        counter += 1
        if counter % 2 == 0:
            raise redis.exceptions.RedisError("CHAOS")              ❷
        return func(*args, **kwargs)                                ❸
    return wrapped
```

❶ **If the special environment variable CHAOS is not set, returns the original function**
❷ **Raises an exception on every other call to this method**
❸ **Relays the call to the original function**

Now you just need to actually use it. Similar to the first experiment, you'll modify the app.py file to add the call to this new function. Find the definition of the `get_interests` function, and prepend it with a call to the decorator you just saw. It should look like the following (the decorator is in bold font):

```
import chaos2
@chaos2.raise_rediserror_every_other_time_if_enabled
def get_interests(session):
    """ Retrieve interests stored in the cache for the session id """
    return json.loads(CACHE_CLIENT.get(session) or "[]")
```

Also, make sure that you undid the previous changes, or you'll be running two experiments at the same time! If you did, then that's all you need to implement for experiment 2. You're ready to roll. Let's run the experiment!

8.3.3 Experiment 2 execution

Let's make sure the application is running. If you still have it running from the previous sections, you can keep it; otherwise, start it by running the following command:

```
CHAOS=true \                       ❶
FLASK_ENV=development \            ❷
FLASK_APP=app.py \                 ❸
python3 -m flask run               ❹
```

❶ **Activates the conditional chaos experiment code by setting the CHAOS environment variable**
❷ **Specifies the Flask development env for better error messages**
❸ **Specifies the same app.py application**
❹ **Runs Flask**

This time, the actual experiment execution step is really simple: browse to the application (http://127.0.0.1:5000/) and refresh it a few times. You will see the recommendations every other time, and no recommendations the other times, just as we predicted, proving our hypothesis! Also, in the terminal window running the application, you will see logs similar to the following, showing an error on every other call. That's another confirmation that what you did worked:

```
127.0.0.1 - - [07/Jul/2020 22:06:16] "POST /search HTTP/1.0" 200 -
127.0.0.1 - - [07/Jul/2020 22:06:16] "POST /search HTTP/1.0" 200 -
LOG: redis error CHAOS
```

And that's a wrap. Two more experiments under your belt. Pat yourself on the back, and let's take a look at some pros and cons of the approach presented in this chapter.

8.4 Application vs. infrastructure

When should you bake the chaos engineering directly into your application, as opposed to doing that on the underlying layers? Like most things in life, that choice is a trade-off.

Incorporating chaos engineering directly in your application can be much easier and has the advantage of using the same tools that you're already familiar with. You can also get creative about the way you structure the code for the experiments, and implementing sophisticated scenarios tends to not be a problem.

The flip side is that since you're writing code, all the problems you have writing any code apply: you can introduce bugs, you can test something other than what you intend, or you can break the application altogether. In some cases (for example, if you wanted to restrict all outbound traffic from your application), a lot of places in your code might need changes, so a platform-level approach might be more suitable.

The goal of this chapter is to show you that both approaches can be useful and to demonstrate that chaos engineering is not only for SREs; everyone can do chaos engineering, even if it's only on a single application

Pop quiz: When is it a good idea to build chaos engineering into the application?

Pick one:

1. When you can't get it right on the lower levels, such as infrastructure or syscalls
2. When it's more convenient, easier, safer, or you have access to only the application level
3. When you haven't been certified as a chaos engineer yet
4. When you downloaded only this chapter instead of getting the full book!

See appendix B for answers.

Pop quiz: What is not that important when building chaos experiments into the application itself?

Pick one:

1. Making sure the code implementing the experiment is executed only when switched on
2. Following the best practices of software deployment to roll out your changes
3. Rubbing the ingenuity of your design into everyone else's faces
4. Making sure you can reliably measure the effects of your changes

See appendix B for answers.

Summary

- Building fault injection directly into an application can be an easy way of practicing chaos engineering.
- Working on an application, rather than at the infrastructure level, can be a good first step into chaos engineering, because it often requires no extra tooling.
- Although applying chaos engineering at the application level might require less work to set up, it also carries higher risks; the added code might contain bugs or introduce unexpected changes in behavior.
- *With great power comes great responsibility*—the Peter Parker principle (http://mng.bz/Xdya).