**Experiment challenges:** Performance testing for enterprise applications consists of multiple closely related scenarios by varying a few configurable (preferably one) parameters at a time.

**Application challenges:** In distributed software testing the applications should start efficiently and in the provably correct order by simultaneously enforcing serialization constraints and leveraging the distributed system’s inherent parallelism.

**Cloud challenges:** Selecting the most appropriate cloud from many cloud offerings is a non-trivial task. Also, migrating an application between two clouds is a complex, time-consuming, and error-prone task. Finally, communication, coordination, synchronization, monitoring, and complete management challenges; lastly, the dynamic nature of the cloud introduces extra complexity

## 4. Use Case: Testing Website (Blog) using Locust IO:

In this experiment, I will also perform performance tests (automated tests) on working websites that were designed by Dr. Angela Yu, and implemented by myself.

### a. Requirement Analyst:

The application name is Blog-Template, and it was designed using Flask Framework, and Jinja Template (Python). It’s also using PostgreSQL for the database. The database schema is used primarily to store blog content and user information. This website is fully functional and extendable. Code review was conducted by Truc Huynh. Technologies that are implemented in the application: Flask framework, Jinja template, Object-Oriented Programming, SQL Alchemy, Password Hashing (werkzeug. security), Bootstrap, CK Editor (user input), HTML, CSS, JavaScript…

Blog-Template was originally designed by Dr. Angela Yu and implemented by Truc Huynh. Application is a blog template that allows registered users to edit or comment on a post. Only the admin can create, edit or delete posts. The application was hosted at heroku.com, and the hyperlink is “<https://template-blog.herokuapp.com/>”.

The Blog-Template was designed as a template blog and post on GitHub by Truc Huynh (“<https://github.com/jackyhuynh/blog-template>”). Anyone can use the blog for any purpose. The first user will be automatically set as the admin.

### b. Structure of Backend:

**main.py** is the python script containing all components required to run the application (Blog Template). The “main.py” consists of all required packages (Python library) and the route for the front-end. The front-end was built with HTML, CSS, and Bootstrap, and stored in the templates folder (about.html, contact.html, footer.html, header.html, index.html, login.html, make-post.html, post.html, register.html). The static folder contains all the .css files, images, javascript.

The package(folders containing main.py and all the necessary files) is then pushed through GitHub and deployed to Heroku from GitHub. Therefore, every time I make a change, the application will automatically change. I also schedule automatic testings (mostly functional testing and GUI testing) for the app every time I make a change (just to make sure nothing will break when I make a new commit).

PyCharm IDE was used to develop this code but the user can use any IDE to navigate the code.

from flask import Flask, render\_template, redirect, url\_for, flash, abort  
from flask\_bootstrap import Bootstrap  
from flask\_ckeditor import CKEditor  
from datetime import date  
from functools import wraps  
from werkzeug.security import generate\_password\_hash, check\_password\_hash  
from flask\_sqlalchemy import SQLAlchemy  
from sqlalchemy.orm import relationship  
from flask\_login import UserMixin, login\_user, LoginManager, login\_required, current\_user, logout\_user  
from forms import LoginForm, RegisterForm, CreatePostForm, CommentForm  
from flask\_gravatar import Gravatar  
import os  
  
app = Flask(\_\_name\_\_)  
app.config['SECRET\_KEY'] = os.environ.get("SECRET\_KEY")  
ckeditor = CKEditor(app)  
Bootstrap(app)  
gravatar = Gravatar(app, size=100, rating='g', default='retro', force\_default=False, force\_lower=False, use\_ssl=False, base\_url=None)  
  
##CONNECT TO DB  
app.config['SQLALCHEMY\_DATABASE\_URI'] = os.environ.get("DATABASE\_URL", "sqlite:///blog.db")  
app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False  
db = SQLAlchemy(app)  
login\_manager = LoginManager()  
login\_manager.init\_app(app)  
  
  
@login\_manager.user\_loader  
def load\_user(user\_id):  
 return User.query.get(int(user\_id))  
  
  
##CONFIGURE TABLE  
class User(UserMixin, db.Model):  
 \_\_tablename\_\_ = "users"  
 # Create a User  
  
class BlogPost(db.Model):  
 \_\_tablename\_\_ = "blog\_posts"  
 # Create a BlogPost  
  
  
class Comment(db.Model):  
 \_\_tablename\_\_ = "comments"  
 # Create a comment  
db.create\_all()  
  
  
def admin\_only(f):  
 @wraps(f)  
 def decorated\_function(\*args, \*\*kwargs):  
 if current\_user.id != 1:  
 return abort(403)  
 return f(\*args, \*\*kwargs)  
 return decorated\_function  
  
  
@app.route('/')  
def get\_all\_posts():  
 posts = BlogPost.query.all()  
 return render\_template("index.html", all\_posts=posts, current\_user=current\_user)  
  
  
@app.route('/register', methods=["GET", "POST"])  
def register():  
 form = RegisterForm()  
 # ... Code to Register ...  
  
  
@app.route('/login', methods=["GET", "POST"])  
def login():  
 form = LoginForm()  
 # ... Code to log in...  
  
  
@app.route('/logout')  
def logout():  
 logout\_user()  
 # ... Code to Logout ...   
  
  
@app.route("/post/<int:post\_id>", methods=["GET", "POST"])  
def show\_post(post\_id):  
 form = CommentForm()  
 # ... Code to show post ...  
  
  
@app.route("/about")  
def about():  
 return render\_template("about.html", current\_user=current\_user)  
  
  
@app.route("/contact")  
def contact():  
 return render\_template("contact.html", current\_user=current\_user)  
  
  
@app.route("/new-post", methods=["GET", "POST"])  
@admin\_only  
def add\_new\_post():  
 form = CreatePostForm()  
 # ... Code to add new post ...  
  
  
@app.route("/edit-post/<int:post\_id>", methods=["GET", "POST"])  
@admin\_only  
def edit\_post(post\_id):  
 post = BlogPost.query.get(post\_id)  
 # ... Code to edit the post ...  
  
@app.route("/delete/<int:post\_id>")  
@admin\_only  
def delete\_post(post\_id):  
 post\_to\_delete = BlogPost.query.get(post\_id)  
 # ... Code to delete the post ...  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 app.run(debug=True)

### c. Test Plan Design:

**Test plan:** After analyzing Blog-Template Source Code and requirements, I design my test plan. My plan is simple to increase the number of users over time and stop at 3 million users. I will only perform 3 tasks on the websites: login, visit the contact page and visit post number 1. Even though, Locust IO allows me with much more tests such as creating tasks, editing, or deleting them; creating new users; importing new documentation into user posts… However, I just want to keep my tasks simple for this use case.

I will run an automated test starting at 1000 users at a spawn of 1.00 seconds then I will increase by 1000 for each test case. There are 2 scripts that I need to design: the performance test script and the schedule test script.

### d. Test Suite Implementation:

**Plan:** Run the test script (or use another script to schedule the test script). The schedule script should be able to increase the number of users after each successful test. The process is described below

**Test Script:** Set the wait time for each user from 1 to 300 seconds so that each user will be able to navigate and read 1 post or do some comments. Task number 1 is to navigate to the contact page (and stay there for 40 seconds). After 50 seconds navigate to post 1. PyCharm IDE is used to run the code.

This is the content of the test script (locust\_test.py)

from locust import HttpUser, task, between  
  
  
class TestCases(HttpUser):  
 host = "https://template-blog.herokuapp.com/"  
 # Set the wait time for each user from 1 to 300 seconds  
 # So that each user will be able to navigate and read 1 post or do some comment  
 wait\_time = between(1, 300)  
   
 # When a user in login the user  
 def on\_start(self):  
 self.client.post("/login", json={"Email": "jackyhuynh87@gmail.com", "Password":"1234"})  
  
 # task number 1 is to navigate to the contact page (and stay there for 40 seconds)  
 @task(10)  
 def visit\_contact(self):  
 self.client.get("/contact")  
  
 # after 50 seconds navigate to post 1  
 @task(50)  
 def visit\_post\_1(self):  
 self. client.get("/post/1")

**Execute:**

Run this script within the Terminal of PyCharm IDE

locust -f locust\_test.py

Then go to “<http://localhost:8089/>” for monitoring. This is how it looks like

**Automation Testing:**

For automation testing without the WEB UI, run this script below instead.

locust -f locust\_test.py -u 1000 -r 100 --run-time 1h30m --stop-timeout 99

To schedule a full test suite, we need to write another Python script that increments the users as our test plan and simply schedules it. Depend on the application report on peak time (e.g. 2 million people operate at the same time). The tester can simply set the user to that peak, and schedule the performance test when they have a change in the system (or just for regular maintainable).

Something like this

python3 schedule-script.py

Schedule script (schedule-script.py) can be used to scan and detect code changes, then run a performance test if needed.

### e. Test Result:

Result for the first test case 1000 users at 1.00 spawn rate. The result return with a 0% fails rate. Testers can download the PDF, CVS version for data analysis. Web UI can also be used for monitoring with real-time monitoring and interactive chart.

In my latest test, Template-Blog was able to handle 20,000 users at a 0.5 spawn rate.

**Test Result** This is printed within my IDE (Py Charm).

2022-03-10T17:00:41Z  
[2022-03-10 12:00:41,483] DESKTOP-J4K30PB/INFO/locust.main: Shutting down (exit code 1)  
 Name # reqs # fails | Avg Min Max Median | req/s failures/s  
----------------------------------------------------------------------------------------------------------------------------------------------------------------  
 GET //contact 1095 1(0.09%) | 120 27 19214 130 | 0.81 0.00  
 POST //login 1000 0(0.00%) | 144 114 1134 140 | 0.74 0.00  
 GET //post/1 5162 4(0.08%) | 125 28 25672 140 | 3.83 0.00  
----------------------------------------------------------------------------------------------------------------------------------------------------------------  
----------------------------------------------------------------------------------------------------------------------------------------------------------------  
 2 GET //post/1: ConnectionResetError(10054, 'An existing connection was forcibly closed by the remote host', None, 10054, None)  
 2 GET //post/1: RemoteDisconnected('Remote end closed connection without response')  
 1 GET //contact: ConnectionResetError(10054, 'An existing connection was forcibly closed by the remote host', None, 10054, None)  
----------------------------------------------------------------------------------------------------------------------------------------------------------------  
 Name # reqs # fails | Avg Min Max Median | req/s failures/s  
----------------------------------------------------------------------------------------------------------------------------------------------------------------  
 GET //contact 1 0(0.00%) | 42 42 42 42 | 0.01 0.00  
 POST //login 10 0(0.00%) | 162 137 181 160 | 0.05 0.00  
 GET //post/1 18 0(0.00%) | 106 40 194 66 | 0.09 0.00  
----------------------------------------------------------------------------------------------------------------------------------------------------------------  
 Aggregated 29 0(0.00%) | 123 40 194 150 | 0.15 0.00  
  
Response time percentiles (approximated)  
 Type Name 50% 66% 75% 80% 90% 95% 98% 99% 99.9% 99.99% 100% # reqs  
--------|--------------------------------------------------------------------------------|---------|------|------|------|------|------|------|------|------|------|------|------|  
 GET //contact 42 42 42 42 42 42 42 42 42 42 42 1  
 POST //login 170 170 170 170 180 180 180 180 180 180 180 10  
 GET //post/1 69 170 170 170 180 190 190 190 190 190 190 18  
--------|--------------------------------------------------------------------------------|---------|------|------|------|------|------|------|------|------|------|------|------|  
 None Aggregated 150 170 170 170 180 180 190 190 190 190 190 29

**Number of User Chart** This is the WEB UI chart (1000 users at 1.00 spawn rate)

**Number of Response Time** This is the WEB UI chart (1000 users at 1.00 spawn rate)

**Total Request per Second** This is the WEB UI chart (1000 users at 1.00 spawn rate)

## 5. Advantages/Disadvantage of Locust

### a. Advantage:

Verifies the speed, accuracy, and stability of the software match expectation. Assists the system by authenticating the responsiveness and managing the scalability and reliability of software features. It is free and open-source.

### b. Disadvantage:

Locust can be a costly mistake if done haphazardly, leading to inaccurate results and conclusions. Need Python programming knowledge to perform the test (must be very familiar with Python to design complicated test cases)

## Reference:

E. J. Weyuker and F. I. Vokolos, “Experience with performance testing of software systems: issues, an approach, and case study,” in IEEE Transactions on Software Engineering, vol. 26, no. 12, pp. 1147-1156, Dec. 2000, DOI: 10.1109/32.888628. [1]

D. Jayasinghe et al., “Expertus: A Generator Approach to Automate Performance Testing in IaaS Clouds,” 2012 IEEE Fifth International Conference on Cloud Computing, 2012, pp. 115-122, DOI: 10.1109/CLOUD.2012.98. [4]

Kumar. K., “Write your first performance/load test in Python.”, Retrieved from <https://medium.com/@kundan3034/write-your-first-performance-load-test-in-python-e8e2132ef775> [2]

Tutorial Points (n.d.), “Performance Testing”, Retrieved from <https://www.tutorialspoint.com/software_testing_dictionary/performance_testing.htm> [3]

Bystron. C., Heyman. J., Hamren J., Heyman. H. and Holmberg. L., Locust, Retrieved from <https://github.com/locustio/locust> [5]

Retrieved from [http://docs.locust.io/en/stable/running-without-web-ui.html#](http://docs.locust.io/en/stable/running-without-web-ui.html) [6]