Password Hashing

Introduction

You've decided to do something a bit different and try some hacking. Beginners luck! You were able to hack into BookFace and download their database table containing usernames and hashed passwords! They're stored in `resources/hackedDatabases` of this package.

BookFace wasn't storing their customers passwords very securely, and you were able to find the salt value and hashing configuration the service used for their passwords. A salt is a random string that should be unique to each user to make the user's passwords harder to discover. However, we've discovered that BookFace is using the same salt for all their users, so it was easy to crack. (For this activity you do not need to know how password hashing and salting work, these details are implemented for you!)

We decide to see if we can crack into anyone's account. If we can guess their password correct, we can append the user's salt value we stole, hash it, and we're in! People are a lot more predictable than you'd think when it comes to passwords, so we downloaded a list of common passwords from the internet and saved them under `resources/commonPasswords`.

For this activity, we are going to go through all the passwords in one of commonPasswords files, pre-compute their hash, and store them in a new file. We'll use these hashed passwords to see if they match against the hacked database table of usernames and passwords.

Currently, we have a `PasswordHasher` class which has methods for generating hashes, and writing passwords and hashes to a new file.

In `PasswordHasher` is a method called `generateAllHashes`, which is responsible for generating hashes for a given list of passwords. Currently, we pass all the passwords as one batch to one `BatchPasswordHasher`. However, this is taking forever! You think it'd be more efficient if you split the work of generating hashing for all the passwords among multiple threads, so you decide to update the code!

Before Starting

We'll be using csv files as *resources* to test our compiler. To make sure we can run and debug our tests in IntelliJ, verify the directories `src/main/resources` and `src/test/resources` are marked as *Resources Root* and *Test Resources Root* respectively. You can do so by looking at the directories in IntelliJ's project panel and looking for a small yellow icon on the folders.

If you don't see an icon, right click the directories and select 'Mark Directory as -> Resources Root' and 'Mark Directory as -> Test Resources Root' from the context menu, depending on whether it's in the `src/main` or `src/test` directory. Do **not** select 'unmark as...' since that means they already are marked as resource roots, and IntelliJ should be able to access files within those folders.

Lastly, make sure to run the `main` method in `PasswordCracker` to see how it works and what its current output is. If you want to try processing larger lists of passwords you can also change the file names stored in the `HACKED_DATABASE_FILE` and `COMMON_PASSWORD_FILE` constants at the top of the `PasswordUtils` class to match the different `.csv` file names in the `resources` directory.

You can use the following files: Passwords:

- commonPasswords1K.csv
- commonPasswords10K.csv
- commonPasswords100.csv

Databases:

- hackedDatabase1Kx100x1900.csv
- hackedDatabase100x10x190.csv
- hackedDatabases10Kx1Kx19K.csv

Here's what needs to be done:

Phase 1 - Make BatchPasswordHasher a Runnable

1. Modify BatchPasswordHasher to implement Runnable so that its hashing logic can be run on a separate thread
2. Write a unit test to test your code.

You can verify that your changes are working as expected by passing the tests in `BatchPasswordHasherIntegrationTest`.

Phase 2 - Split the work in generateAllHashes

Now that you've made BatchPasswordHasher a Runnable, you can run multiple threads of `BatchPasswordHasher`, which will allow us to split the work of generating hashes for all the passwords passed into `PasswordHasher`'s `generateAllHashes`. `generateAllHashes` is called in `PasswordCracker`'s main method to get the hashes for all of the common passwords, which is then used to match against the usernames and passwords in the hacked databases.

Modify `generateAllHashes` in `PasswordHasher` so that we split the work between four `BatchPasswordHasher`s that run in four separate threads. Take a look at how `generateAllHashes` currently works to help figure out where you'll need to make changes.

To do this, you'll need to do make the following changes in `generateAllHashes`:

1. Split the list of passwords into four sublists so each thread will do a portion of the work. There are multiple ways to do this -- one way is to use `com.google.common.collect.Lists`'s `partition` method, which takes in a list and the number of entries you want each sublist to contain. An example using `partition` might look like:

```
// we want each list to have 1/4 of the total entries in the list of
passwords
List<List<String>> partitionedLists = Lists.partition(passwords,
passwords.size() / 4);
for (List<String> sublist : partitionedLists) {
  ... concurrently hash each sublist
}
```

   You can view the javadoc for `partition` here.
2. Create four `BatchPasswordHasher` objects, that each take a part of one of the sublists.
3. Create and start four threads to run your `BatchPasswordHasher` objects. These threads should replace the `hashPasswords()` call currently being made in the method.
4. Once all the threads have completed, combine all of the hashed passwords each `BatchPasswordHasher` generated, and return them. By waiting for the threads to complete, what you are doing is allowing each run method to complete. Only then will the map of passwords be fully populated. Knowing when threads are complete isn't that straight forward. In a future lesson we'll learn more about this. For today, we have provided you with a method called `waitForThreadsToComplete` in `PasswordHasher`, which takes in a list of threads. You can call this method and pass in the four threads you've created. A call to this method will force your application to pause and wait until all threads complete. When the `waitForThreadsToComplete` returns your application will resume.

To test, you can run the main method in `PasswordCracker`. In the console, you should be able to see print statements from each `BatchPasswordHasher` that says how many hashes it's generated, and then it prints out any users from the database that are using the common passwords. You can also try running your PasswordCracker with different files of passwords. You just need to update the `COMMON_PASSWORD_FILE` variable in `PasswordUtil` with a different file name or the `HACKED_DATABASE_FILE` variable with a file from `resources`.

You can also verify that your changes are working as expected by running the tests in `PasswordHasherIntegrationTest`.