Overengineering Hackathon Projects for Fun and Absolutely No Profit

Author: Dan Subak

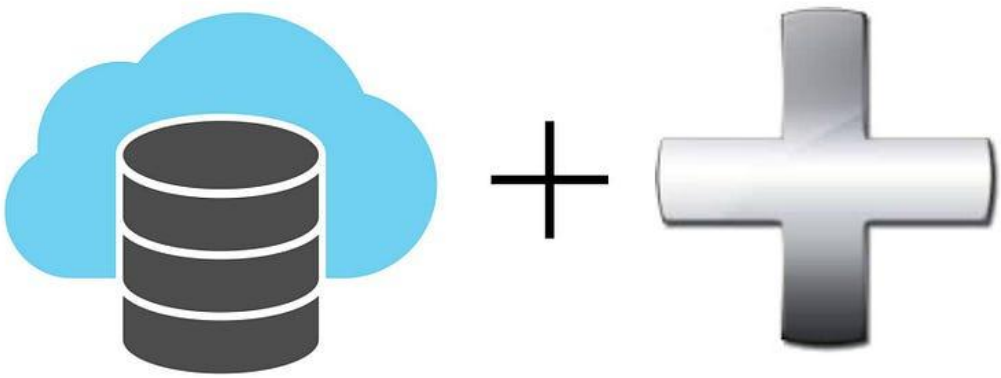
Claps: 80

Date: Jul 20, 2019

In early May, myself and three other intrepid engineers participated in the Boston Stupid Shit No One Needs & Terrible Ideas Hackathon. The theme of this hackathon is slightly different from other hackathons you may have heard of or participated in before â€” rather than taking a day to focus and develop something useful, this event asks participants to focus and build something that is â€œbad, useless, terrible, and probably should have never been made.â€ Participants then demo their terrible ideas at the end of 8 hours of work. This event was hosted a few blocks from our office at the Cambridge Innovation Center and seemed like a great opportunity to put our professional skills to good (bad?) use!

We began by crowdsourcing our terrible ideas â€” while only 4 of us ended up participating, the entire engineering organization had a great time thinking of ideas which were deliberately illogical, nonsensical, unusable, harmful and in one case potentially illegal. No idea was too stupid, but unfortunately we did have to settle on one. Some projects (such as this one) were too trivial to occupy 4 engineers for 8 hours. Others far too difficult, such as training drones to steal packages from other delivery drones. Some never really made that much sense (edible shoes?) All of them were spectacularly terrible and Iâ€™d urge anyone considering participating in such a hackathon to spend time brainstorming with your peers â€” it was almost as much fun as the event itself! Some of the highlights were:

- **OpenBox**: A single-owner-at-a-time inbox that everyone can access and use, but only by one person at a time.
- **NightOwl**: Like f.lux but in reverse, increasing blue light as it gets closer to bed time.
- **Sacrifice**: When downloading a file, deletes the file on the filesystem which is closest in size to the download.
- **MailPager**: A custom PagerDuty integration with a direct mail API, so you receive a letter in the mail for PagerDuty incidents.



The terrible logo for our terrible idea

The idea we decided to pursue was a project we affectionately called â€œMath-as-a-Serviceâ€. Our next step involved scaffolding out a proposed workflow for adding two numbers â€” we did this a week ahead of the event. This was a useful exercise as it informed both the API weâ€™d eventually expose as well as the database models we would need to create. We also created an OpenAPI spec which we used as a guideline while developing the actual server. These efforts allowed us to have a clear set of deliverables going into the day of the hackathon.

At the end of the day, our prototype was complete; a RESTful, Flask-based microservice for doing math. **In seven short HTTP requests, you can perform the calculation of algebraic expressions such as 1+1 or 2*8.** The possibilities really are endless! Find the (horrible) code at https://github.com/math-as-a-service/configurations. By way of example, hereâ€™s how youâ€™d go about adding 2 + 2 and getting the result:

1. First, our client would issue a POST request to the `/expression` endpoint. This creates an expression that can be associated with multiple operators and operands and eventually is evaluated, generating a result.
2. Next, with the ID of our created `expression`, weâ€™d issue a POST request to the `/operand` endpoint. In the body of the request, weâ€™ll provide the `expression_id`, value

of the operand (in this case 2), and rank. Rank is the position of the operand relative to other operands â€” the first number in the expression is rank 0, the second is rank 1 and so on.
3. Step 3 is to create our second operand. In the case of adding 2+2, this will involve making the same POST request to the `/operand` endpoint to create the second operand, but this time, weâ€™ll give it `rank=1`.
4. Create an `operator` object by issuing a `POST` request to the `/operator` endpoint. This takes a very similar payload to the `/operand` endpoint â€” an `expression_id`, `rank` and `value`. The main difference is that `value` in this case refers to the mathematical (or logical) operator you want to use â€” for this example, that would be the string `â€™+â€™`.
5. Now that our expression has the relevant operator and operands created and ordered, we are ready to evaluate it. To do so, weâ€™ll issue a POST request to the `/evaluation` endpoint and specify the `expression_id` to evaluate â€” this creates an `evaluation` which indicates the status of your pending `result`.
6. In order to determine if your math has been evaluated yet, our client can poll the `/evaluation` endpoint using the ID from step 5 and issuing GET requests. On initial creation, the evaluation will be in a `STARTING` state. During the evaluation itself the status will be reflected as `EVALUATING`. Finally, if everything evaluates without issue, itâ€™ll be marked as `FINISHED`; otherwise, it will get an `ERRORED` status.
7. Once the evaluation is in a `FINISHED` state, the corresponding `result` object will contain the output of your algebraic expression and can be retrieved by issuing a GET to the `/result` endpoint! Simple