Redbubble Coding Test

Brief

The Redbubble system has many digital images, often taken with a camera. We have exported EXIF data from a selection of these images. This data is available via an API.

Instructions

Your task is to create a set of static HTML files to allow a user to browse the images contained in the API.

The API is available at: /api/v1/works.xml.

Create a batch processor that takes data from the API, and produce a single HTML page (based on this output template), for each camera make, camera model and also an index page.

The index page must contain:

- Thumbnail images for the first 10 work;
- Navigation that allows the user to browse to all camera makes.

Each camera make HTML page must contain:

- Thumbnail images of the first 10 works for that camera make;
- Navigation that allows the user to browse to the index page and to all camera models of that make.

Each camera model HTML page must contain:

- Thumbnail images of all works for that camera make and model;
- Navigation that allows the user to browse to the index page and the camera make.

The batch processor should take the API URL and the output directory as parameters.

The data returned from the API contains a small sample set of works.

Guidance

At Redbubble, our first significant hiring hurdle is a coding test. Often the experience and achievements of the candidate don't match at all with the quality of code they send through to us. This page attempts to give you some guidance on what to think about beyond the actual instructions. Please read this **very carefully**, it is important.

tl;dr

Consider the person who has to read/run your coding test and do everything you can to make their job easier. Represent your very best professional practices with the code you submit.

The people reading your code test will be working through hundreds of them. As we are language agnostic, they'll be seeing solutions written in a wide variety of languages and demonstrating the entire breadth of the software quality spectrum. The few who submit readable, simple, extendable, reliable and performant applications make it through to the interview stage. Filtering down to these few is an incredibly time-consuming and often very frustrating process. As you write your coding test remember to treat the end users of that test (the hirers) as the very important and time-poor people that they are.

Read the Instructions, Meet the Requirements

This may seem obvious, but even the better quality tests we've received often leave out a requirement or two. If they say submit your code via a github repo then do it. If they say write an application that takes two parameters, a username and a shoe size, then do that. If they say output JSON files do that. If you have

questions about the requirements don't be afraid to ask. When we receive a test that doesn't meet the straightforward test requirements we lose faith in that candidate's ability to build code to satisfy our requirements.

Include a README

Just because you know how to run your app doesn't mean the person looking at your test does. They can spend 20 minutes of their precious time reading your source code and reverse engineering it to work out where your main class is and the command to run it. Or you can just tell them. By not spelling it out you are moving your application closer to the reject pile. If you're doing the test in a language they're not currently working in then be more explicit with your instructions. For example, if you write in Ruby, tell them which version of Ruby to install along with some instructions on how to install it. Likewise with C compilers, Java package managers, .NET versioning and so on can all turn into a big time-sink of Googling, reading man pages and frustration. So be considerate, explicit and accurate in your README. Better yet, use the tools that are appropriate for your platform (`sbt`, `maven`, `bundler`, `cocoapods`, etc.) that make this simple(r).

It's also a good idea to include some other stuff about what you've built in your README. Things like why you made certain decisions, how you architected your code, why you chose a certain language or framework for your solution, any issues or shortcomings with your code, etc. We generally read this first, and this gives us a good impression of what you were thinking, and helps us gauge your code.

Write Production-Quality Code

Write code that is as similar to production-level code as possible, given the constraints on your time. If the coding test is the first stage in the application process, all the hirer knows about you is what you tell them in your code. One giant class named `Test` with three epic methods named `process_1`, `process_2` and `process_3` and no tests tells us everything we need to know; you aren't getting anywhere near our precious production code. Likewise, leave out the TODOs. We don't like them in our code base and they don't do you any favours in a test. `// TODO: Implement error handling` is not a selling point for your coding skills. These may seem like extreme examples, but too many of the coding tests we receive look like something hacked together by a second year Comp Sci student, and not the production-quality code we're looking for.

Write Tests!

We like TDD, one of the things we look for in a candidate is the ability to write good tests. What makes a good test?

- Make sure your test makes it easy to identify where the problem in your code is. A simple way to do this is to make only one assertion per test block.
- Unit test where necessary, e.g. complex methods, code we might find obscure or crucial functions. Just to be sure you know what we mean by unit testing: "a unit [is] the smallest testable part of an application"Wikipedia.
- Integration tests for our coding exercise are optional, but whether to include them or not will depend on the type of test you are doing. If you're not sure, put them in anyway. So far we haven't rejected anyone for putting in too many tests.
- Test everything. TDD, BDD, unit, integration - there are different schools of thought on how to best approach testing and most of them have merit. At the end of the day the important thing is that any breaking changes to your code break your tests and clearly identify where the problem is.

Design an App

Even the simplest of applications can be split into a data model class, a display/runner class and some logic between the two. Design your solution to be robust, extensible and simple and then we'll know you can do the same thing for us. Yes, those principles involve trade-offs, and how you make those trade-offs will tell us a lot about what you think is important. Of course your audience will differ, but the best thing you can do is try to strike a balance between them. If you pick one and optimise for that at the expense of the others you're making assumptions about what we're looking for that may end up with your code in the reject pile.

Include Some Error Handling

We run your code from the command line, it exits immediately with no error messages or any other kind of output. If we can, we'll spend the time inserting random debugging statements into your code in the hope of finding the error. More than likely we won't have that kind of time to spare, so we put you straight in the reject pile. Help us out by adding some error handling, preferably with stack traces (it's okay, we're coders, we can read stack traces) and you'll greatly increase your chances of making it onto the "interview" pile.

Pay Attention

A key quality of a good developer is attention to detail. Here's a list of detail

that some developers thought were not important, but we reckon they are:

- don't hardcode pathnames to your computer;
- make sure the app compiles before you send it in;
- make sure the app runs before you send it in;
- tidy up your spelling everywhere - comments, project names, email, variable names, class names and all of the other places;
- use a recent version of your chosen language, not an old one;
- don't abbreviate variable names. We have no idea what `c << a if g` means and neither will you in six months;
- finish the coding test before you submit it, a partial solution is not enough. If you feel it is taking too long, get in touch and we can negotiate a better submission date.

Going for a job is a nerve-wracking process; no one likes to be judged. Writing code does have a strong creative element so what appeals to one person isn't necessarily going to appeal to another. However, it's our hope that by following these guidelines you can maximise your chances of getting through to the next stage in the hiring process. Have fun and good luck!

Disclaimer: Of course not all this advice will apply to all coding tests or reviewers so use with a dollop of your own common sense.

This guidance was originally taken from Lonely Planet.