Programming manifesto:  
**1. Use Drew's skematics as a END GOAL. Not a waterfall template.**  
Planned out the path of small goals and iterations that will slowly build up  
to the final vision. Always have a working product at the end of the day.  
That means, front end, back end.  
  
**2. Sparing is a sucky replacement for real fighting.**  
I took karate and got my ass kicked a lot in school dispite this.  
2.1: I like Mockito. But do not think it is a substitute for testing live software.  
2.2: I am developing in as real of an environment as I can. Hence on a real server using openshift.  
 Deploying to openshift is actually faster than deploying to localhost. I didn't think that would be the case, but it is.  
  
**3. I disagree with leaving the front-end and back-end in different projects/code bases.**I do not disagree that they should be separated concerns. It would be a good idea to give the UI  
people their own folder in the repo.   
  
The only reason I can think of to keep UI and Back-End in separate code bases is if we do not TRUST the UI people. Example: A code competition where thousands of people are trying to stick a UI onto our back end.  
  
Also: Allowing the UI people to put their code in our code base would NOT prevent us from doing the above also if we wanted to later.  
  
**4. Error responses use same data structure as non-error responses:**Using a valve (company) style way to find errors at runtime.  
If FRONT END makes bad HTTP request, we return the  
data in the format expected by the UI, but all the fields will  
be populated with an error message.  
  
Reasoning:   
1: Front end programmers don't have to write extra code to handle another  
data schema for the error.  
2: Helps prevent the UI from breaking, but still makes the error known.

5. If you anticipate the error happening, you should code for it. You should also code with a paranoid style of "I am going to make mistakes".

|  |
| --- |
| CandidateExamsController.java |
| @JsonView(CandidateExamViews.ConfirmEndpoint.**class**)  @RequestMapping(value = "confirm", method = RequestMethod.***GET***)  @ApiOperation(value = "A small subset of data to confirm the Candidate's identity", response = CandidateExam.**class**,  produces = "application/json",  notes = "The information provided will consist of only the name of the candidate and the skill being tested, along with the token. "  + "This should be enough for the candidate to confirm they are taking the correct test. "  + "Please note, if the token is not found, an error\_response will be sent back through Json.")  **public** Object confirm(@RequestParam("token") **final** String token) {  **final** Optional<Object> byToken = Optional.*ofNullable*(**this**.testManager.read(token));  **return** byToken.orElse(ErrorResponse.*report*(**this**.missingToken(token)));  }  @JsonView(CandidateExamViews.StartEndpoint.**class**)  @RequestMapping(value = "start", method = RequestMethod.***GET***)  @ApiOperation(value = "The full Candidate Exam, with the timer started", response = CandidateExam.**class**,  produces = "application/json",  notes = "This will provide the test proper and the 1 hour time limit will have started. "  + "Once started, a test cannot be taken again.")  **public** CandidateExam read(@RequestParam("token") **final** String token) {    //If token is bogus, we want to return a friendly JSON error message  //to our UI developers, rather than crash application with null pointer exception.  **final** Optional<Object> byToken = Optional.*ofNullable*(**this**.testManager.read(token));  **if**(byToken.isPresent()){  **return** ErrorResponse.*report*( **this**.missingToken(token) );  }  **return** **this**.testManager.start(token);  } |

I was trying to fix the read() function because I got a null pointer exception when I called the API using  
a bogus token. Mike says not to because the UI people should never be calling with a bad token.  
If we all did exactly what we were supposed to do... we wouldn't need any error checking.  
  
Also, I program in a style where I want MY custom error checking to find the error BEFORE the JVM  
finds the error. My experience is doing this makes it very easy to fix the bug if it happens. Especially since I include probable causes of the error in my error message. (When the code is fresh in my head.)  
When an error pops up a month later after you've forgotten all the details of the class, you will thank yourself.  
  
My proposed solution for this project is to have some type of error class that extends BaseEntity.  
That way the confirm() function wouldn't have its output type as generic Object.  
I despise generic Objects floating around in code. I am a fan of being as strict with typing as possible.  
  
My personal solution: Do not have an error class. But have configurations within all entities that are  
indicative of an error.  
  
Example: If your UI looks like:  
-----------------------------------------------------------------------------  
QUESTION#1: ERROR:API CALL ERROR: Invalid Token Used  
[ ] : ERROR:API CALL ERROR: Invalid Token Used  
[ ] : ERROR:API CALL ERROR: Invalid Token Used  
[ ] : ERROR:API CALL ERROR: Invalid Token Used  
[ ] : ERROR:API CALL ERROR: Invalid Token Used  
------------------------------------------------------------------------------

You've successfully:  
1. Kept strict typing.  
2. Avoided a null pointer exception from bad api call.  
  
We are doing NEITHER of that. And it pisses me off.  
  
Also. Our UI people can't do anything with a null pointer exception.  
And a new Error-Type response is just another JSON schema that our UI  
people now need to support and know about.   
  
**5. Live debug info is a good idea in the final product.**  
Premature optimization is stupid. This is a Java Enterprise app. Not a real-time video game.  
Because of this, the maintainability gained by having live debug information in the final product  
can outstrip any performance benifit of leaving it out.  
  
What would you rather have:  
An app that is a few milliseconds faster?  
Or an app that takes a exponentially less time to debug when a problem occurs?  
  
Analogy of live debug info in a final product:  
Look at the pallet of color on a Mountain Dew can sometime.