ACF OPRE

Technical Evaluation MAPS

May 12, 2021

Team

Nina Mak, nina.mak@gsa.gov
Carly Jugler, carly.jugler@gsa.gov
Amy Mok, amy.mok@gsa.gov
Randy Hart, randy.hart@gsa.gov
Nikki Lee, nicoleb.lee@gsa.gov
Elie Berkowitz, elie.berkowitz@gsa.gov
Elizabeth Ayer, elizabeth.ayer@gsa.gov

Methods and criteria

The goals of the technical evaluation of MAPS were to conduct a thorough investigation of the code and architecture to make a determination about whether or not MAPS, as it is currently designed and built, can be extended and maintained or if it should be rebuilt. The criteria we looked at were the characteristics of a codebase that typically predict how maintainable and extendable it is. These criteria are:

- Quality of the documentation of the code and the system
- Presence and quality of code tests
- Structure and security of the data
- Readability and complexity of the code
- Organization of the code
- Quality of the architecture of the overall system.

We also identified a few sub-criteria that are related to some of the previously listed criteria which are particularly critical to the maintainability and/or extensibility of the system:

- Does the documentation contain instructions for how to setup locally and/or deploy the system
- Is the data in a logical and maintainable format
- Is the code written such that it is clear what it's intended to do for someone who didn't write it.

The possible scores for each of the noted criteria were excellent, good, fair, and poor.

Summary of the scores

Criterion	Score*
Documentation ● Is there documentation? ● Is the documentation clear? ● Does it explain how to run and deploy the code? ● Is there a system architecture included in the documentation?	Poor
Testing Practices ● Are there tests? ● Do they run? ● Is the test coverage > 90%?	Poor
Data Practices ● Is the data in a logical and maintainable format?	Fair

• Is the database proprietary or OS? • Is the data secure?	
Code Readability & Complexity • Are names of things clear? • Is it clear what things are meant to do? • Is there inline documentation where needed? • Does code follow DRY principles? • Is the boolean logic easy to understand? • Is there usually one return statement per function? • Does it require many calls throughout the codebase to carry out a single set of functionality?	Fair
Code Structure & Organization • Are functions or methods concise? • Are source files concise? • Are source files organized and neat? • Is the structure of the repository intuitive and easy to navigate?	Good
System Design • Is the code broken out into modules and/or APIs? • Is it clear how the system is deployed?	Fair

Overall, given all of these factors, the extensibility and reusability of the codebase is really poor. We recommend that MAPS be rebuilt using modern programming languages and standard best practices for both documentation and code development for the sustained success of the product long term.

Details of the Scores

Documentation: Poor

There is no documentation on how to run the application locally and no documentation for why certain decisions were made in the code. There are very few inline comments to explain complex/hard-to-understand logic and/or decision making to future developers. Methods and functions don't use a standard in-line notation that can be programmatically parsed, which is a good practice to let others who did not write the code know what the code is intended to do. Furthermore, external dependencies and system architecture are not documented; the boundary diagram is the most we have in terms of dependency documentation, but it does not include information about Twilio and Oracle. Lastly, all knowledge about how to get the system set up lives in the head of one developer.

Given that the code dates back to 2012 and it is not a small codebase, it could be a really big effort to document the codebase. Additionally, histories for why certain decisions were made in the code are most likely lost to time at this point and cannot be properly documented. It is hard to thoroughly document code that has a long history.

Testing practices: Poor

Most tests in the codebase are commented out (this usually indicates the code/tests are either unusable or deprecated) and there are no instructions on how to run the tests that are not commented out. Further, the current developers are not using a testing framework, and assertions are instead printing the results of specific functions or method calls (test cases) to standard out. This method of testing is not standard and not best practice because the tests are not asserting an expected behavior or functionality for a given piece of code. Real and useful software tests compare expected outputs to given outputs and then pass or fail based on the comparison. Writing tests like they are currently written for MAPS means the developers are comparing what they expect in their heads to the actual output, which is not helpful for people who are new to the codebase. As they're currently written, these tests could not be run in an automated build pipeline because whether or not the tests pass or fail is up to the discretion of the person reading the results.

Because there are no coded expectations in the tests for how certain parts of the code should function, new code added could potentially break existing working code without anyone's knowledge. Further, when code does not have tests, it is highly likely that code is not written in a modular manner that allows for writing tests, and as a result, the code is hard to change and add new features to.

Data practices: Fair

The data is currently stored in an enterprise database, Oracle, which requires a proprietary license to use. We would recommend against maintaining the Oracle database in the long run due to the opacity in how the database software works under the hood (the source code is not open source) and due to its unnecessarily high operational costs. There are plenty of well-documented, highly usable, highly secure open source database softwares available that do not require a license to run and operate.

We didn't get to see the actual data, but the vendor did provide a database schema and model classes. Since we didn't have access to the database, we're unsure if there are triggers or other special procedures defined. If so, it would make it harder and take more effort to migrate the database to another non-Oracle database.

The tables and schema are clear, but the type of information contained at the table level is highly specific. For example: there is a table called `PSC_CODE` and `PSC_FEE`. The idea here being that if you were to add another contracting shop, you'd create two more tables with their name and fees when instead this could be genericized to `PROCUREMENT_SHOP` with fields for the specific name of the shop and their fees. A highly specific database structure like this can have its perks in terms of time for retrieval of data, but there are perhaps better ways to optimize for retrievability such as data indexing. Additionally, having a lot of tables with similar types of information in them in a single database can lead to unnecessary redundancy of data throughout the database which requires business logic in the code to keep many versions of the same information up to date.

Relationships are defined in the model classes but not seen from the schema. We are unsure whether this is due to the relationship not being defined properly in the model classes or the output of the schema being incomplete. Further, the data models are extremely flat and do not have a lot of inheritance to leverage similar models. Therefore, similar models create a lot of duplicated code without making use of inheritance. But model classes seem logical and maintainable.

There seem to be users/roles as a concept in the MAPS application, but we cannot determine by looking at the backend code if each service has a guard against who can access them in the backend or if this is merely being done as hide/show functionality on the client side. This is a potential security concern.

There are raw SQL statements in the Java files and Jasper reports. Calling hardcoded tables/field names in SQL statements makes it hard to make changes to fields/tables without breaking them, and may potentially be a security concern.

Code readability and complexity: Fair

Functions and methods are clearly named and generally easy to read with fairly easy-to-follow logic and there is generally one return statement per function. However, there are a lot of abbreviations in the names of functions and variables that make it unnecessarily difficult to know what is meant, and a lot of commented out code with no explanation as to why.

Wicket and Spring (the two Java frameworks MAPS is built on) are fairly opinionated about how things are structured within the two frameworks and the code seems to follow their required structure, which is helpful for someone completely new to the system because they can read the documentation for both Wicket and Spring to figure out what is what and why certain types of files exist. However, as mentioned previously, we wonder about perhaps extracting more functionality out of specific classes and into parent classes which the specific classes could then implement. We are not sure if the seemingly high level of repetition throughout the code here is just inherent to Wicket and Spring or if there are ways to follow DRY (Don’t Repeat Yourself) principles in the code even while using the frameworks.

Common functionality, for example the functionality to schedule and send notifications and emails, uses the creation of custom messaging queues and custom cron jobs as opposed to OS libraries or services inside the AWS cloud that are designed and built to carry out these types of tasks really well.

Code structure and organization: Good

Source files are concise and the top level structure of the repository is organized (dao/dto/model,etc), but within each top level structure (within dao, etc) it is hard to understand if there should be more hierarchy in organizing them. An example: in the `web/wicket` subdir, there are subdirs called `components`, `models`, and `pages`, and there's also another subdir called `document` which contains its own subdirs called `components`, `models`, and `pages`. Without proper context as to why the repository is structured this way, it is difficult to navigate.

System design: Fair

This is a monolithic application, and the architecture does not leverage APIs. The code is organized in terms of layers. There is a model layer, DAO layer, DTO layer, service layer, etc. But the service layer functions seem similar to the DAO layer, which makes it hard to understand where functions should go as a new developer to the codebase. There are benefits of using monolithic applications vs. breaking out front-end applications and back-end applications. Monolithic applications can make it easier to set up and run at the beginning of development, but as it evolves and the codebase grows bigger, it becomes harder to make a change or add new features because everything is tightly coupled. In a monolith, every layer may need to be touched and changed to make a new feature. The codebase is currently rather large, and as the front end interface becomes hard to use, it is time to think about decoupling the front-end from the back-end.

Additionally, there's no CI/CD configuration files to understand how the system is deployed. Configurations for deployment are done manually and are not documented.

Recommendation

With no documentation and no tests, it would be very hard to implement a new feature in MAPS without a lot of help from developers who originally wrote the code. Additionally, it is not easy to come into the MAPS codebase without prior experience with the specific frameworks (Wicket and Spring) and understand how the disparate parts interact with one another. A lack of documentation exacerbates this.

Further, given that there is no documentation on how to get the app up and running locally and that doing so requires access to an expensive proprietary Oracle database to do so, ease of development by software developers who are new to the codebase is both expensive and very difficult.

Given all of these factors, the overall extensibility and reusability of the codebase is really poor. We recommend that MAPS be rebuilt using modern programming languages and standard best practices for both documentation and code development for the sustained success of the product long term. Additionally, we feel that, because MAPS is a rather large system, a frontend that’s separate from the data layer would allow for more changeability and ease of development in the future.