CLIENT Map Application Assessment

In order to help government agencies build smarter communities with CLIENT, you must be able to deliver crucial features quickly. An influx of new customers means an influx of new and varying needs, which requires that the CLIENT application be easy to learn, maintain, update, and scale.

The purpose of this assessment is to identify opportunities for improving the performance and security of the CLIENT Map Application.

Specifically, you asked me to answer the questions:

- What's good?
- What's bad?
- What's risky?
- What's missing?
- What would you do?

Each section below describes an opportunity for improvement, my assessment of its impact on CLIENT, and my recommendation for next steps.

Map Application is Slow to Load

A few times throughout our conversations you have mentioned that “speed is a problem” with the Map application.

According to the Application Performance Management bundle in the demo account, the *average time it takes the Map Suitelet to load* is about **18 seconds**. Anecdotal testing in the browser console backs this number up.

This is a very significant load time, especially for a web application. The primary cause of this is the pattern used to load the configuration records for the application:

1. Search for all configuration records
2. Load each individual configuration record instance (~70 individual records are loaded)
3. Read relevant fields from each record

Loading a record is one of the most costly SuiteScript operations in terms of execution time, and it is placing an unnecessarily heavy burden on the Suitelet execution.

Business Impact

Long load times are a key point of frustration for users of any application. Frustrated users are less productive, and will *never* be a source of referrals, which are a key source of trust and growth for your application. Fewer referrals means a smaller pipeline where you have to work even harder to build trust and help more customers.

If this same pattern for loading configuration records is used by other applications in the CLIENT suite, your customers will be getting a *uniformly poor* experience when loading any of your applications.

Recommendation

First, refactor the Map Application by eliminating the loading of each configuration record. Instead, retrieve all relevant fields as Search Columns during the Search operation. Anecdotal performance tests in the browser console show a reduction of the load time by **99%** when eliminating the loading of configuration records.

See the provided source file for the code used for console testing.

Second, determine whether this same pattern for loading configuration data is used by any other applications in the CLIENT suite, and refactor each one similarly.

Authentication Assessment

Security is of utmost importance, particularly for government agencies and constituents. With the caveat that web security is not my expertise, I have not come across any significant red flags in the Map application's Authentication mechanism as it relates to the security of the application or its data.

I have made a concerted effort to research best practices and examples for Authentication, and the authentication mechanism used in the Map application appears to correctly conform to those examples.

What I have noticed is that the code used to authenticate is repeated several times throughout the Map application.

SPECIFIC FILES OMITTED

By repeating this code in several places throughout the application, it becomes very difficult for a developer to maintain and update the authentication mechanism.

Business Impact

Nothing of note from a security standpoint.

Repeating code multiple places throughout an application again significantly increases the developer's cognitive burden. Any time the authentication for the application needs to be modified, a developer will need to remember each and every place this code is repeated, and make the appropriate changes across all of them. This significantly increases the cost and time for maintenance of the authentication.

Repeating code also significantly adds to onboarding and training efforts. New developers that get introduced to the authentication mechanism will feel they have to memorize and retain all the possible locations where the authentication exists, in addition to any other code blocks that are repeated throughout the application. This is overwhelming and reduces morale of new developers.

Recommendation

Relocate the Authentication logic to a separate, reusable module, and leverage this single module across the locations mentioned above and any other applications in the CLIENT suite that require this same Authentication process. By moving all of the Authentication logic into *one place*, you will significantly reduce the effort required to learn, maintain, and modify the Authentication process. This will make converting the existing authentication over to SSO a much simpler because all of the logic exists in one place, not several.

You may also consider encrypting the ESRI Authentication Key on the Employee record using a Password field type instead of a Text field. SuiteScript has encryption/decryption APIs that should allow the encrypted field to be retrieved simply while keeping the data obfuscated in the UI.

Application Architecture is Monolithic

The Map application's architecture is essentially monolithic, with the vast majority of the application logic living in one single file: `GS_MapService_Dougherty.js`. The same logic is then duplicated across multiple files (`GS_MapService_DeltaCo.js`, for example).

Business Impact

In my experience, monolithic architectures carry the following disadvantages:

- Higher cost of maintenance and troubleshooting as code is difficult to read and navigate. More time spent troubleshooting is less time spent adding valuable features. This means much slower time to value for customers and typically reduces developer morale because they're spending most of their time fixing instead of creating.
- Higher cost of onboarding new developers. Monolithic architectures require more effort to learn because you need to learn the entire system at once and retain it all. By contrast, with a modular architecture, developers can focus and learn only their immediate responsibility without worrying about the rest of the application until they need to. When an application is simpler to learn, developers become more productive in less time, increasing their morale and helping you break even on their compensation much faster.
- Not scalable as the maintenance effort and effort required to collaborate multiplies with each new feature added. When all of the application's logic is contained in one file, it will be impossible for multiple developers to work on that same file at once due to the nature of NetSuite's shared environments.

Recommendation

I advise a concerted effort toward refactoring the Map's monolithic architecture into a more modular approach. Break down the monolithic application into smaller, single-focus modules. “Less code” is easier to learn, easier to fix, easier to improve. As you scale your user base, you'll need to scale your development team. A more modular architecture will be easier for those new developers to learn, and they'll be much quicker becoming fully productive.

Separating the concerns of the application into small modules will help scale your developers' ability to collaborate and work on the application simultaneously.

Built on SuiteScript 1.0

I have been informed by one member of the SuiteScript team that: “Within the next few releases, although not within one year, we will be disallowing creation of new 1.0 scripts”.

Business Impact

If the above information is accurate, you can expect that before NetSuite version 2019.1, you will need to be writing all of your new scripts in SuiteScript 2.0. Because 1.0 and 2.0 cannot be intermingled in the same script, you may be limited on what you can accomplish or add to your existing 1.0 script. You will not be able to take advantage of the latest features of SuiteScript, and you will be missing out on any continued performance or security improvements that NetSuite makes with SuiteScript 2.0.

I consider delaying a transition to 2.0 extremely risky for any SuiteScript application. Your maintenance efforts will grow exponentially as 1.0 fades into obscurity. You will effectively need to maintain two distinct code bases, and very soon developers who come in to the NetSuite market won't even learn 1.0; thus, there will be even less available talent to help you maintain and improve your applications.

Recommendation

In conjunction with the effort to modularize the application, write those modules in SuiteScript 2.0 right away.

Given that most of the application is standard JavaScript and not SuiteScript, this should not require much additional effort. The majority of the effort will be in learning the 2.0 syntax and converting the Suitelet itself to its 2.0 counterpart.

For more on my recommendations regarding transitioning to SuiteScript 2.0, see https://stoic.software/why-ss2/

Lack of Meaningful Documentation

There are very few comments at all, and no comments of substance, in the almost 7000-line `FILENAME OMITTED` file, which houses the lion's share of the Map application logic.

Business Impact

Very little documentation makes for a very high cognitive load when ramping up on the application. Longer ramp-up time means less productive developers. Less productive developers are more frustrated, and it takes much longer for the business to break even on their compensation.

Recommendation

By refactoring into a more modular architecture, much of the need for documentation will dissipate as small, focused modules are much easier to read and understand than a single massive file. Documentation efforts can then be geared towards explaining why a module exists and how it fits in to the larger application, rather than filling up with inane comments like `/* End Authentication block */`, which make up the majority of the current comments.

Coinciding with an effort to refactor the application to a modular approach should be an effort to document the larger purpose and higher functions of the modules that get created in order to ease and decrease onboarding efforts.

Significant Amounts of Repeated Code

In addition to the repetition of the Authentication block, there are many other blocks of logic that are duplicated across many files, like the definition of LODs or the manipulation of Parcels.

Business Impact

Repeated code makes for a massive time sink (and thus, a money sink) in maintenance efforts and drastically increases onboarding time for new developers.

When the same code needs to be updated in multiple places, it becomes very easy for developers to miss one, meaning your application will behave inconsistently, and there is a much higher risk of either releasing new bugs or only *partially* fixing an existing one.

Recommendation

Eliminate repeated code by abstracting the repeated logic into its own module, then leveraging that module wherever appropriate. Any code that is needed repeatedly is a good candidate for its own module, or at the very least its own reusable function inside of a module.

This effort can be undertaken concurrently with the refactoring of the monolithic architecture.

Significant Amounts of Unused/Dead Code

There are large amounts of declared variables that are never used and huge swaths of code blocks that are simply commented out (dead code) for no documented reason.

Business Impact

Unused variables and dead code again increase the cognitive burden for developers, complicating onboarding and maintenance efforts by obscuring what code is actually meaningful. They also add technical bloat to the application by making unnecessarily large files, which take longer to load, and by taking up additional memory space for no practical reason, which means less available memory for the meaningful parts of the application to carry out their tasks.

A slower application takes us back to our frustrated users, who will never refer your application to their counterparts.

Recommendation

Eliminate all unused variables and dead code blocks.

Utilize a tool like ESLint to quickly identify unused variables (along with many, many other code quality improvements).

Once a block of code has been deemed unnecessary (i.e. by commenting it out), it should be deleted. Proper source control procedures ensure that any block of code from any point in the application's life is still accessible, so there is no need for it to remain in the source file.

If you do not have disciplined source control processes in place, then that is a critical process to add before continuing *any* development on *any* of your applications.

HTML Content of Application

The HTML for the Map Application is written inline within a Script; it is a block of more than 60 consecutive lines of code that looks, in part, like this:

snippet.javascript

```
CLIENT CODE OMITTED
```

Business Impact

This approach is incredibly hard to read, debug, and maintain for any developer, no matter how experienced or patient.

Any time the markup of your application needs to be modified, a developer will have to dig through all of these String concatenation operations, quotes, and escapes in order to locate the appropriate tag and modify accordingly. This creates a significant and unnecessary “cognitive burden” on the developer during the troubleshooting or improvement process.

This approach to markup also makes it incredibly difficult for, say, a graphic/web designer to improve the user experience of the application because the markup is buried within code they are typically not familiar with.

Sections of code like this that increase the cognitive burden not only make development and maintenance much more difficult, but they also make onboarding new developers on to the application take much longer. Longer onboarding efforts are more costly to the business and more frustrating for the developer.

Recommendation

1. Move all of the static HTML for the application into its own separate HTML file that is stored in the File Cabinet.
2. Add text placeholders to this file where the dynamic HTML will be placed.
3. The Suitelet will then load this HTML file and dynamically replace the placeholders with the appropriate data.

This is the pattern I follow myself when building similar applications.

Static Data Usage

Static data, specifically for the Map's LODs, is defined directly within the application source file.

Business Impact

Similarly to the drawbacks of the HTML content pattern described above and the drawbacks of repeated code, putting large amounts of static data inside the application logic adds a heavy and unnecessary cognitive burden. It clutters up the file, and it is completely inaccessible by any other part of the application.

Defining static data directly in source files also makes it very difficult to locate later on, particularly if the application gets broken up across multiple modules.

Recommendation

Move static data like LODs to JSON files that are stored in the File Cabinet. These files can then be read and parsed by any piece of the application that needs them.

Store all data files in a dedicated `data/` (or other aptly named) directory, and developers will never need to guess at where they put that data six months ago.

[Question] Is there a max length or a max parameter count when passing URL parameters?

In our discovery conversation, you had indicated that a lot of data gets passed around via URL parameters and were wondering whether or not there is any sort of limitation or maximum restriction placed on this.

The answer is “Technically No”, but there is an advised length of 2000 characters. Some servers have limitations on the URLs they can process, but according to the HTTP specification, there is no limit. I have not seen any place within the Map Application that you might be exceeding this advised limit.

Overall Assessment

Whenever I build or assess software, I follow this mantra from Wes Dyer:

“Make it correct. Make it clear. Make it concise. Make it fast. In that order.”

Overall, I believe the Map application is in the “Correct” stage. The application currently accomplishes what it is intended to achieve, but it is not easy to learn, navigate, maintain, or improve.