Sprint One: Semester 3-Spring

In this sprint we will be exploring how integrated back-end services can be developed as applications which can be used by websites and other services. As before, you are welcome to propose your own sprint idea to me and implement that. The basic requirements we have for this project are as follows:

* It must use Node.js.
* It must include the use of at least **two** data structures (in some form) that we’ve learned about so far (arrays, queues, stacks)
* It must store and retrieve data from a database or a file.

If you have an idea that falls outside of this scope, please propose it to me anyway, and I’ll help you scope it.

If you cannot think of anything, or if you want some inspiration, the following is a complete project that you could do for this sprint.

Default Project

For this sprint, imagine that you belong to a top-secret government agency. Your organization has agents all over the world who are actively deployed into dangerous situations, and who need to get access to information, as well as provide information back to the organization. In the past, agents used dead drops – messages hidden in physical locations in the world – to relay information back and forth between headquarters and agents. But this is a dangerous process! What if malicious actors became aware of the location of a dead drop, and compromised the message? Or worse, apprehended the agent!

Luckily, we are programmers, and we are going to solve this problem using our programming skills. The basic idea is that we are going to create a service that has at least two endpoints: one that allows users to end a request to a “file” and store a message, and another that allows the user to retrieve a previously stored message from a file. Critically, when a message is retrieved, it should be removed from circulation so that it cannot be retrieved again, since, as we all know from watching many cheesy spy movies, when secret agents send messages, they always self-destruct after you read them.

We’re going to provide two versions of these end points, one that works like a stack, and one that works like a queue. The stack version will allow us to post a message, and then when someone retrieves a message, the **most recent unretrieved message** will be returned to the user. For the queue version, when someone retrieves a message, it will instead be **the oldest unretrieved message**. These two endpoints can be used for different purposes, for simple status updates that are not mission critical, agents can use the queue. For situations where the information is critical and more-up-to-date information is more important than older information, the agents can use the stack.

When we send requests to this endpoint, it will be in the form of (for example):

{

data: “Data Here”,

AgentId: “SomeNumberHere”,

StructureId: “SomeNumberHere” – actually location id?

}

Let’s break this down. The “data” portion of the request is the message that the agent is trying to send. The “agent\_id” is a unique number identifying the agent sending the information – this number will be stored with the message in the database but will be kept secret from retrievers. The StructureId is a number that identifies a particular “dead drop” queue or stack. If your system gets a request for a StructureId that it does not have a queue or stack associated with already, it should create one and add the new message to that structure.

When we send a retrieve request, it is in the form of (for example):

{

AgentId: “SomeNumberHere”,

StructureId: “SomeNumberHere”

}

When this occurs, the file should store a record of who retrieved the message for the structure, and then return some of the data to the user. If it uses the stack endpoint, it should be the most recent message added to the structure, with the queue endpoint it should be the oldest. Note that we only return the “data” field from the stored message to the user of this endpoint, we keep the agent id a secret that only the file knows.