MakerSpace Queue

Design and Planning Document

2018-2-20, version 1.1

Document Revision History

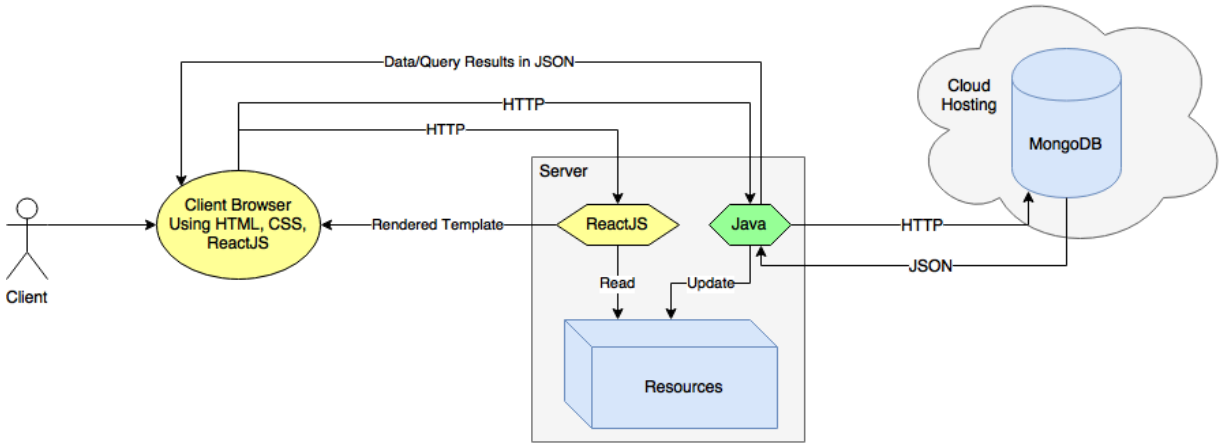
Rev. 1.1 2018-2-20: initial version

Aaron Chauvette, Gavin Kyte, Haodong Jiang, Jack Monshausen, Solomon Ghberemeskel

Index

System Architecture

We will use the Model, View, Controller architecture.



Model

We will use MongoDB to act the part of Queue and Archive of the documents associated with the print request process. MongoDB is a NoSQL database that integrates well with Java and unlike SQL databases, can easily store files that will later tie in with print requests. MongoDB is also extensively documented, easy to set up, is cloud hostable through Atlas and several other providers, and is flexible. The last point is important because the cost of schema changes is very low as no migration is needed and schemas aren't enforced on the document level. MongoDB uses an extended JSON language (BSON) for queries and has its own CLI and APIs.

View

We will use a ReactJS server that renders HTML templates to serve to client browsers. The content for these pages will be loaded asynchronously from the view at some level by making HTTP requests to our Java server with replies with JSON content that can be parsed View-side.

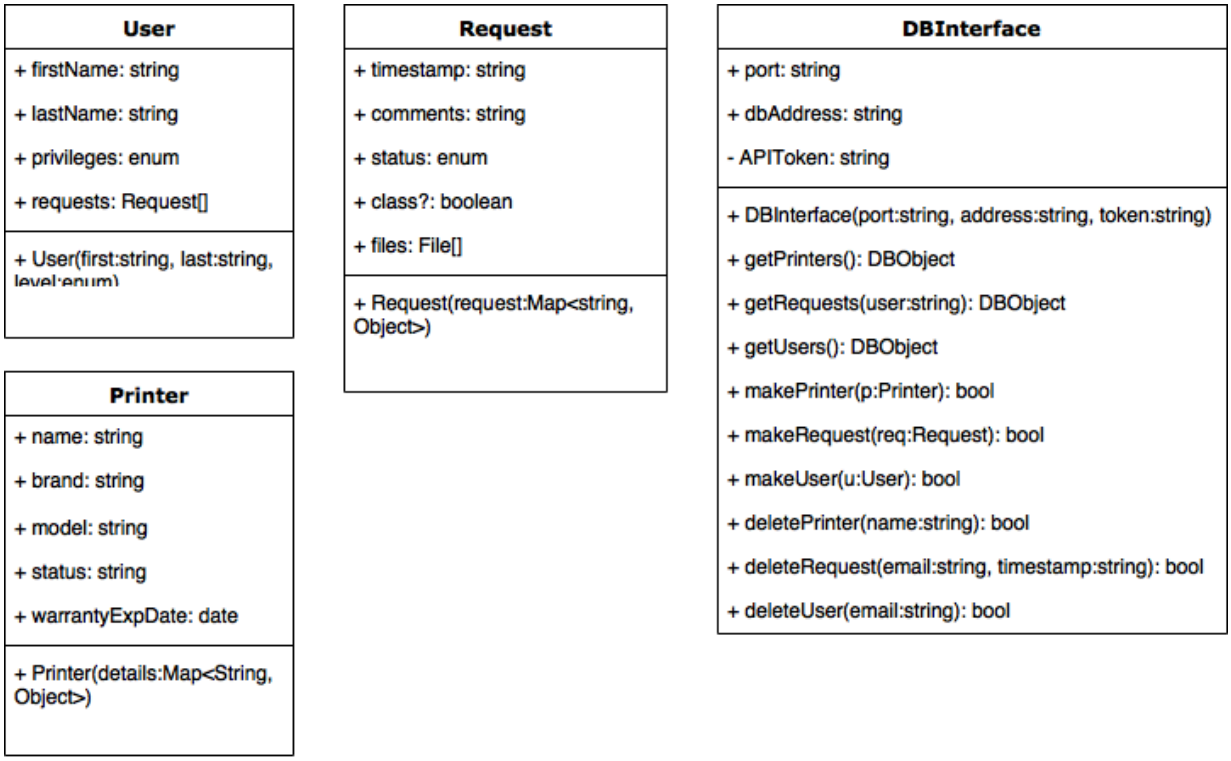
Controller

We will use Java as our main backend language. A java server will handle the transfer of data from the Model into parsable content for the View. The files that cannot be transferred over JSON in HTTP responses can be placed into a Resources bin that the View can access and

subsequently load into the page. This currently is going to be a file that the Java backend writes to the image src file of the project. This file will be accessible by both servers, and since react renders from the backend we will be able to rerender the page when the image gets added to the folder. This will be important for the download or direct printing of .STL files, as well as the splicing of those files into thumbnails for previewing in the view.

To interact with the Model, we will need the Java-MongoDB driver. We can also integrate some open sourceware to help map domain-level objects into DBObjects and back to JSON. See https://www.mongodb.com/blog/post/getting-started-with-mongodb-and-java-part-i for references.

Design Details

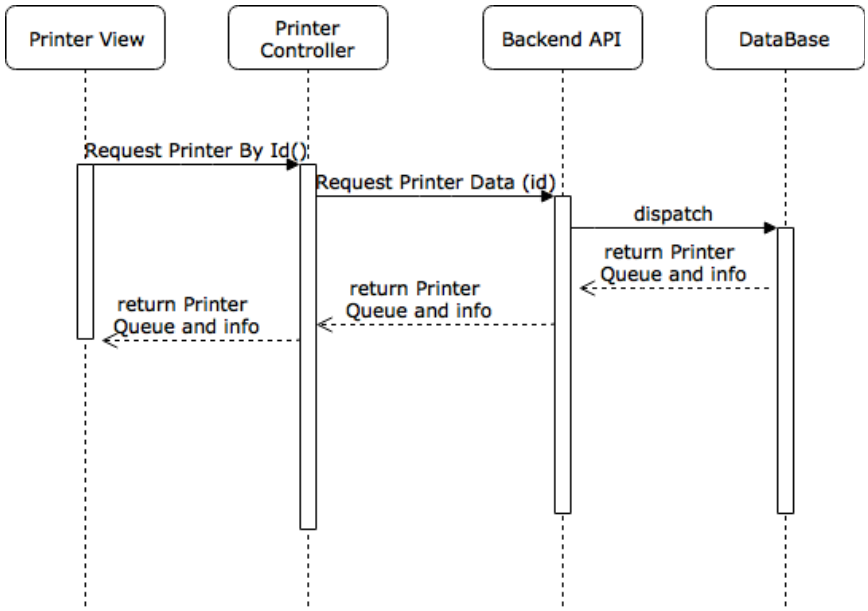


Since MongoDB is a key-value, document-based database with denormalized data, the schema differs from relational models most may be familiar with. Here is the schema that should accommodate our needs:

```
[{
  “email”: “<string>”,
  "type": "USER",
  “firstName”: “<string>”,
  “lastName”: “<string>”,
  "privileges": "<ADMIN/STAFF/REQUESTER>",
  "requests": [
    {
      "_id": "<timestamp>",
      "comments": "<string>",
      "status": "<ORDERED/IN_PROCESS/COMPLETED>",
      "class?": "<true/false>",
```

```
        "requestItems": [{"file": <blob>, "qty": <int>,
“status”:<ORDERED/IN_PROCESS/COMPLETE>}]
      }
   ]
},
{
   "name": "<string>",
   "brand": "<string>",
   "model": "<string>",
   "status": "<string>",
   “currentFileId”: “<string>”
}]
```

Printer View Object:



Login Page:

Content	Component	Description
Email	Text Field	User’s Login Key
Password	Password Field	User’s Password
Login	Button	Submits users Email and password to the back end

Continue as Guest	Button	Redirect user to next page

Requester View:

Content	Component	Description
Make A Request	Button	Redirects person to the requesting page

Queue View:

Content	Component	Description
Enqueue	Button	Add a project from the queue to a printer that can be selected
Job Tile	Job Element	Gets an element from the Job Element

Job Element:

Content	Component	Description
Delete	Button	Remove Job from the queue and moves it to the archive
Expand	Button	Expands all of the File Element inside the Job

File Element:

Content	Component	Description
Delete	Button	Remove Job from the queue and moves it to the archive
Queue	Button	Selects the object ready to be qued to the printer

Printer Object:

Content	Component	Description
Cancel	Button	Remove Job from the the

		printer and moves it to the queue at position 1
Finish	Button	Remove Job from the the printer and moves it to the archive

Printer Array:

Content	Component	Description
Printer	Button	Is an iteration of the printer object
Edit	Button	Adds the ability to add another printer and remove printers from the database

Information Bar:

Content	Component	Description
Login/Logged in as	Button	Depending on weather or not you are logged in, if you are then it will give option to logout if not then brings the user to the login page
Archive	Button	Brings you to the archive sub page
Printers	Button	Brings you to the printer sub page

Archive SubPage:

Content	Component	Description
Data Table	Table	Table of all of the archived projects
Search Bar	Text Field	Querys the database for all of the files that fit within the paramaters
Full name	Button	Sorts the current parse of the

		database with names
Date Requested	Button	Sorts the current parse of the database with date requested
School	Button	Sortss the list on weather or not the project is school related
LinkToFile	TextField	PlaceHolder for showing the location of file/ a link to download the file again

Security and Privacy:

This part of the implementation procedure includes an overview of the system security and requirements that must be followed during implementation. During the printing process, we may receive user's email, their full name and the job that's sent. It is still a personal information; however, it is less likely that we would deal with Privacy Act.

Regarding MongoDB, we will be referencing https://docs.mongodb.com/manual/security/ on best practices and authentication of users interacting with the database.

System Security Features:

In this section, we will provide an overview and discussion of the security features that must be addressed when it is implemented. It should include the determination of system sensitivity and the actions necessary to ensure that the system meets all the criteria that would a web server could possibly have.

- ➔ Since web print allows any user with access to the 3D Maker space user web interface the ability to upload a design for printing, it will inherently increase surface area for several attacks on the web interface. Users could spam the form and fill up the database— Could limit requests per day.

o Things we could do to minimize the amount of spams coming to our system would be:

- ➔ Using a Human-Friendly Bot-Unfriendly Questions
- ➔ Using a CAPTCHA, which is a script to block spam bots from accessing our forms while our real users can get through
- ➔ More specifically, third party applications can result in remote vulnerabilities. For instance, in Tinkercad one can download unverified. This is because these applications are online and are used to render print jobs on the server after the user has uploaded their file.
    - Ways to prevent remote vulnerabilities

- ❖ Never using an arbitrary input data in file literal (anything that's interpreted as a sequence number of data. It could be integers and float numbers)
- ❖ Dynamic whitelisting would be another option, but it is not easy to implement as this is used in some big organizations. It deals with compliance as well.

One way to alleviate the security risk could be applying security updates to the orchestrated application. In addition, since Xavier's network is vulnerable to the public, we could use a Sandbox security mechanism which uses a dedicated virtual machine that is isolated from the rest of the print system. It is a security mechanism for separating running programs, usually in an effort to mitigate system failures or software vulnerabilities from spreading. It can also serve to execute untested or untrusted programs or code, possibly from unverified or untrusted third parties, suppliers, users, or websites. Then, we can render the job using the application on the web interface server.

Another concern is the topic of SQL injections. We will allow the admin to query the database on some level to get proper analytics from it, but we must be cautious to prevent users from running SQL commands inside print requests and other actions that would allow them to escalate their own privileges or modify the DB. SQL injection attacks are also used to delete records from the database. We could prevent that from happening by blocking URLs from our web server. We know that it's not possible to fully prevent SQL injection attacks, however, we could be as minimally vulnerable to SQL injection attacks as possible. We could just try to find some common SQL query keywords in URLs and block them from entering to our web server. This is very possible with regular expressions to parse URLs and the forms that Requesters can submit.

- ➔ We implement a solution to encrypt our sensitive data in our database. This might include our users' passwords, security questions and answers, and others information that would be a turning point for our attackers. The hope is that even if SQL injections get into our system, encrypting our data will give us a time to discover the breach and maybe take some other actions like enforcing the rule of resetting passwords for our users.
- ➔ Another way would be not storing our passwords and other very valuable data in our database. That way, even if these malicious attacks get into the system, it's unlikely that they will access our data.

We can also utilize detailed Firewall systems on our server. Additionally, we will consider separating our data, web application firewalls and HTTP requests when files are sent. In general, even though these methodologies don't guarantee the prevention of vulnerabilities and

injection attacks, they will serve to combat and minimize the amount of attacks we may encounter.

Implementation Plan

Units - The Perceived complexity or difficulty on a scale of 1 (Easy) - 10 (Hard)
Est. Time - In hours to research, code, implement

Iteration 1

Task Name	Units	Est. Time	Responsible	Dependencies
1. Database			Gavin	
1.1 Refine schema details	2	1		N/A
2. Basic Web Interface			Jack, Solomon	
2.1 View Queue	2	6		Database, Server
2.2 Login page (Auth)	2	2		Server
2.3 Basic visuals of request interactions	3	4		Server
2.4 Dynamic content from HTTP response	5	3		Database
2.5 Link to download files	2	2		Server
3. Server implementation			Gavin, Jack	
3.1 Setup Java server	3	1	-	N/A
3.2 Setup ReactJS server	3	1	-	N/A
3.3 Connect to database	1	1	~	Database hosting
3.4 Find hosting (XU/local network?)	1	2		Privileges
4. Backend API			Gavin, Aaron, Haodong	
4.1 DBObject <-> JSON mapping	4	5		OM Modules, Java-Mongo Driver

Iteration 2

Task Name	Units	Est. Time	Responsible	Dependencies
1. Create Rest of Domain Objects			Aaron, Gavin	SpringBoot, Database (?)
1.1 User	1	1		
1.2 Request	1	1		
1.3 Request-Item	1	1		
1.4 File	1-5	1-4		Tool for verifying and converting 3D print files
2. Server Connection			Gavin, Jack	
2.1 HTTP calls from V to C				
2.2 Async Calls				Allowing us to keep all different parts of data to be called from the front end and not have the webpage slowly load
2.3 Deal with image Displaying				We need to figure out a clever way to display the images from the thumbnails of stl files
3. Requester Queue View				
3.1 Render printers on page				
3.2 Render queue bar				
3.3 Display status and print on printer				
3.4				
4.				

Testing Plan

Unit Testing:

Our goal with unit testing is to be able to reliably check to see if every function inside of our codebase does exactly what is intended and nothing more to the system. Unit testing allows us to track if the core functionality that we implement changed potentially making unwanted bugs or code errors that are initially unaccounted for when we change functions.

We will also be running test driven development where our project is completely covered by unit tests. We will also be going through test driven development. Considering we will have all of our functions laid out in this document, our tests will be very easy to write. The majority of tests will be for internal methods, and there will be very few for all of the services that we write.

JUnit tests:

We are going to be using Java for our backend and MongoDB. The test should cover the database as well as the backend. This will allow us to check the backend for any discrepancies that arise while writing code, not allowing for things to break without letting us know.

Jest||Moca Unit Tests:

Our front end will be written in a JavaScript module called ReactJS. We will be unit testing all of the code and methods that we write. This should allow us to follow what the program is doing in real time knowing that our functions are doing what they are supposed to do. This will be mostly the logic behind parsing json and then changing the database with all of the data that is entered.

Integration Testing:

High Level Integration Testing will currently be done manually by testing all of the functions capable of being called on every system used in the project. We will make dummy objects on the front end that are capable of hitting all of the interactions where the front end interacts with the backend. We will repeat this with dummy calls on the backend to see what happens to the database. Through the calls we can completely test all of the services and how they interact with everything else.

We will be testing all of the interfaces multiple times over the course of our project, mainly when we need to put out any kind of release of the project. We will also check this a lot throughout early development. During mid development we will only do these tests whenever one of the interfaces ends up breaking or if we need to change one of the interfaces.

System Testing:

This will simulate the user and allow for us to test all of the elements of the application we are building. We will then go and test everything with using all of the buttons and everything on the front end. This should cover all of the functions within the front and backend and also change the database and test that

Ease of Use Testing:

Ease of use testing will let us take outside input and constructive feedback to further develop our front end. To test this we will have people observe the mock ups for the project early on and after some time we can test this again with getting user input form the user interface after it is programmed.

Smoke Test:

This test will be as simple as an initial compile and run tests to make sure that there is no further app breaking, bugs, or any other errors that will stop the functionality of the application. After this has been completed, we will proceed to regression testing.

Regression Testing:

Based on the datas in the queue, we will check if there are any new bugs/mistakes after modifying to protect the original version of the program. This will be done by running the app and seeing if anything broke while the code was being written. To test, this we will have a simple protocol to go through that will allow us to see if our application still functions with the changes. Afterwards, we will test for the initial bug that we were attempting to fix. If all seems fine, we will push this into the development branch.

Beta Testing:

For beta testing we will end up deploying the code to work on the Inside of the Xavier System. This includes running the frontend and backend servers and directly interacting with each how a normal user would. Then we will have the staff at the makerspace also look at the webpage and give their input on the beta and see what they think about it.