Initial Notes

Tuesday, March 3, 2020 7:19 AM

The Red Giant Diagnostics tool is currently in use, but could use updating and refactoring, as well as possibly Dockerized and launched on the web server as a Docker container.

I will only have one term if I decide to re-write it, but it could greatly benefit from a rewrite.

I would like to refactor the backend to use a standard REST API and implement a front-end using Vue.js instead of using Pug.js templates.

3/3/2020

Tuesday, March 3, 2020 12:53 PM

I have completed the UML designs for the server log files and the two controllers: Log and User
The next steps are:

- Database design
- Front-end UML
- Requirements and Epics
    - Copy the current application
    - Reach out to team for additional requirements
- Set up tracking system
    - Trello? (most likely)
    - Azure DevOps (may look into)

Once these are done, I can start implementing.

3/4/2020

Wednesday, March 4, 2020 8:12 AM

I think I might abandon the Vue.js frontend/WebAPI backend architecture and use the native .Net Core WebApp architecture. This would remove much of the complexity around user management and allow me to more easily develop the application. I can still use Vue.js in some of the pages if it would benefit, but I don't think it's necessary to use it for the entire frontend application.

Update: I really don't like the Razor/cshtml syntax, so I might try and go back to the API/Vue.js architecture.

3/7/2020

Saturday, March 7, 2020 5:34 PM

I have been following an in-depth tutorial on YouTube about how to properly implement a DotNet Core WebAPI and will be using that tutorial and project as a template/example on how to implement the backend of the new Red Giant Diagnostics application.

I am excited to implement the backend with DotNet Core and the frontend with Vue.js.

Update: 3/9/2020
The WebAPI tutorial I have been following is far too complex for my needs.
It would likely result in one of the best and most secure APIs, but I am in far above my head and will need to combine what I have learned from it with what I have learned from my Web Development course.

It may be easiest and even more desirable to use the Auth0 method from Web Development to secure my app. I wonder if there is a way to register a new user through some kind of Auth0 administrator portal.

3/12/2020

Thursday, March 12, 2020 7:42 AM

I have been playing with a basic API project and, through lots of research and trial & error, have developed a fully-functional API that authorizes users with JWT. When a user registers or logs in, they are given a JWT that will authorize their access for two weeks (this can be changed); From this same token, I can pull out their user ID so I know who is accessing a resource and can return resources requested by them; I can also identity if they belong to a certain role, such as if they are an Administrator, giving them access to more restricted resources or allowing them special privileges.

In addition to implementing token-based authentication and authorization, I also converted from a custom AppUser class to the built-in IdentityUser class, managed by the UserManager and RoleManager frameworks. I may extend the IdentityUser class to store more information about the application's users, but I am pleased and excited to actually be using Microsoft's Identity framework in a DotNet Core API application.

I have been using the test API and following documentation and tutorials to develop what I feel is a solid API application and have begun modifying my original Red Giant Diagnostics design to incorporate some of the techniques and patterns that I have found to be helpful.

I am excited to finish the design of Red Giant Diagnostics and move on to implementation. I may take the next week or two to continue playing with the test API to pull out any remaining benefits before starting work on Red Giant Diagnostics at the beginning of Spring term. I should probably meet with Sherry soon to discuss my plans and the progress I've made so far.

I know that I have a finished and working (for the most part) version of Red Giant Diagnostics already deployed, but it was a haphazard project that produced absolutely no documentation and did not follow any software design best practices. Instead of trying to recreate the documentation and design practices for an application that was already finished, I believe that rewriting and strictly following these practices will benefit me tremendously. As an added bonus, it will allow me to incorporate new features, improve existing features, and fix any problems with the existing app.

I think I'll start asking the team this week if there are any features or requirements they would like beyond what is already included in version 1.0.

3/15/2020

Sunday, March 15, 2020 3:58 PM

I think Azure DevOps (dev.azure.com) is going to be my project tracker of choice. I would like to use it as my repository, but I'm so used to GitHub it may be difficult to transition to a new system. I think I won't notice any differences on the client side, but it will be strange not having my code hosted on GitHub.

I know that DevOps can link to GitHub; Maybe I will still host my code on GitHub but use DevOps to manage everything.

UPDATE
I played around a little with DevOps and my TestAPI project; I think I'll try hosting the code for this project in DevOps, if not for more than just to try something different. The project management and CI/CD tools are pretty significant, too. I wonder if there would be a feasible way to integrate my personal web server into this to create a full continuous deployment pipeline - Make some changes on my development machine, run tests, commit and push, run full tests, build a Docker image, deploy to web server. That is my ultimate goal with any project - automated deployment and delivery.

3/16/2020

Monday, March 16, 2020 10:31 PM

I think the next phase of this project is to begin breaking down the design and requirements into the Scrum backlog.
I have created the project in Azure DevOps and can take the next week or so to create the Epics and Work Items that will cover most of the necessary steps for the project.

There will certainly be more work items that I need to add along the way and plenty of bugs to track, but I think I have a good foundation for two reasons:

1. I have built this project before - Even though I built this project haphazardly and using Node.js, it has given me a good idea of what it will take to implement this project from a design standpoint
2. I have built a DotNet Core API - The API project that I have been practicing with has given me a lot of knowledge and insight into what the backend of the project will require and how to solve many of the challenges I will face

I am excited to begin, but I would like to make sure I have as much of the pre-build planning done as I can. So far, I have the design of the backend pretty much completed; I have the requirements and basic user stories; I have the structure and frameworks chosen; and I have the version control, repository, and project management tool selected.

Description

Monday, March 16, 2020 10:37 PM

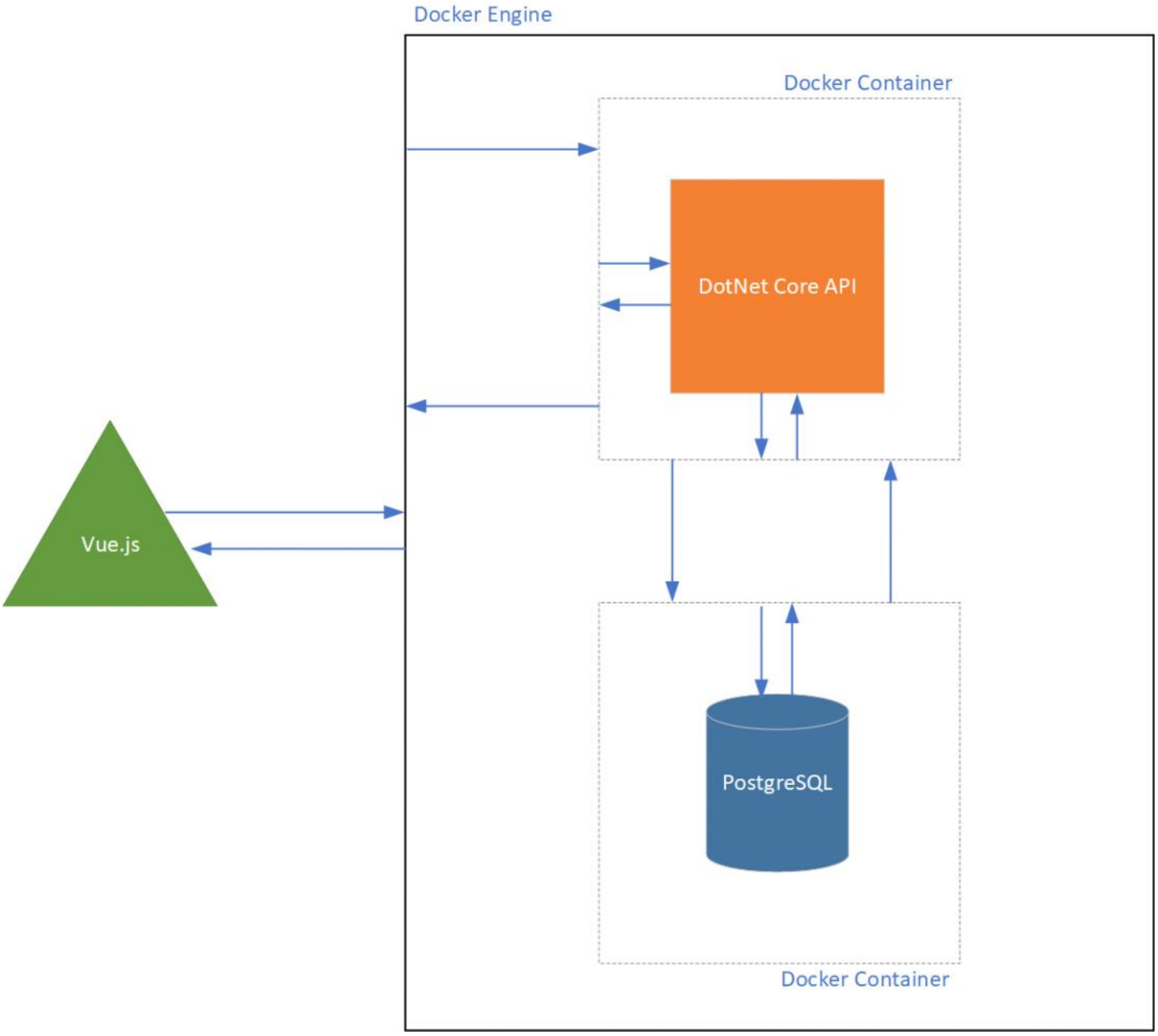
This project will consist of two main pieces:

- A DotNet Core Web API project and
- A Vue.js project

The DotNet Core API will provide the storage and processing power of the application. It is in this server-side app that the log files will be parsed and analyzed, as well as stored and retrieved in a PostgreSQL database.

The Vue.js project will provide a stylish and intuitive interface for the users of the app. My previous implementation relied on Pug.js templates and lots of Bootstrap and custom JavaScript to organize the log data and provide interactivity and animations. This time, I will use the power of Vue.js to provide a beautiful interface with lots of power, while keeping it simple and easy to use for anyone.

The image below describes the architecture and interaction between the three main pieces of the application.



In this image, we can see the Vue.js front-end living outside and separated from the rest of the application. It interacts with the Docker Engine running on the web server; The Docker Engine then carries this communication to the Dockerized DotNet Core API application. From here, if database access is required, the DotNet Core API application speaks with the PostgreSQL database via the Docker Engine again. Any communication back to the Vue.js frontend is carried out via the Docker Engine and out through the webserver.

The PostgreSQL database will be entirely managed by Entity Framework from within the DotNet Core application. In the previous version of this application, the database was MongoDB and I used the Mongoose.js library to manage the data; In other projects in school I used ADO.Net and manually wrote all of the SQL queries to manage the database. I used Entity Framework in the practice API I built and it was incredibly easy and very helpful.

Server Log Design

Tuesday, March 3, 2020 7:22 AM

The application design won't change much, but it would benefit from upfront design and requirements vs building and refactoring on-the-fly like I did in the first iteration.

The main models will, again, revolve around the log files themselves and will be similar in structure.

Server Log Requirements

The server log models will need to be able to contain all the information in a server log file, this includes:

- Date and Time
- RLM Version
- RLM Platform
- OS Version
- Hostname
- User
- Working Directory
- Environmental Variables
- RLM Host ID List
- License Files
    - License UUID
    - Assigned Host
        - IP/Hostname
        - Ethernet Address
        - RLM Port
    - ISV
        - Name
        - Port
    - Licensed Suites
        - Name
        - Seats
        - Expiration
        - Issue Date
- RLM Options
- Statistics
    - Server (ISV or RLM)
        - Name
        - Port
        - Running
        - Restarts
    - Debug Log File
    - Start Times
    - Messages
    - Connections
    - Checkouts
    - Denials
    - Log Removals
- License Pool
    - ISV

        - Pools
            - Product
            - In Use
            - Users/Machines
- Debug Logs
    - ISV
    - Log Data
- RLM Instances
    - RLM Version
    - Command
    - Working Directory
    - Process ID
    - Main TCP Port
    - Alternative Port(s)
    - Web Port
    - ISV Servers
    - Current Instance Flag

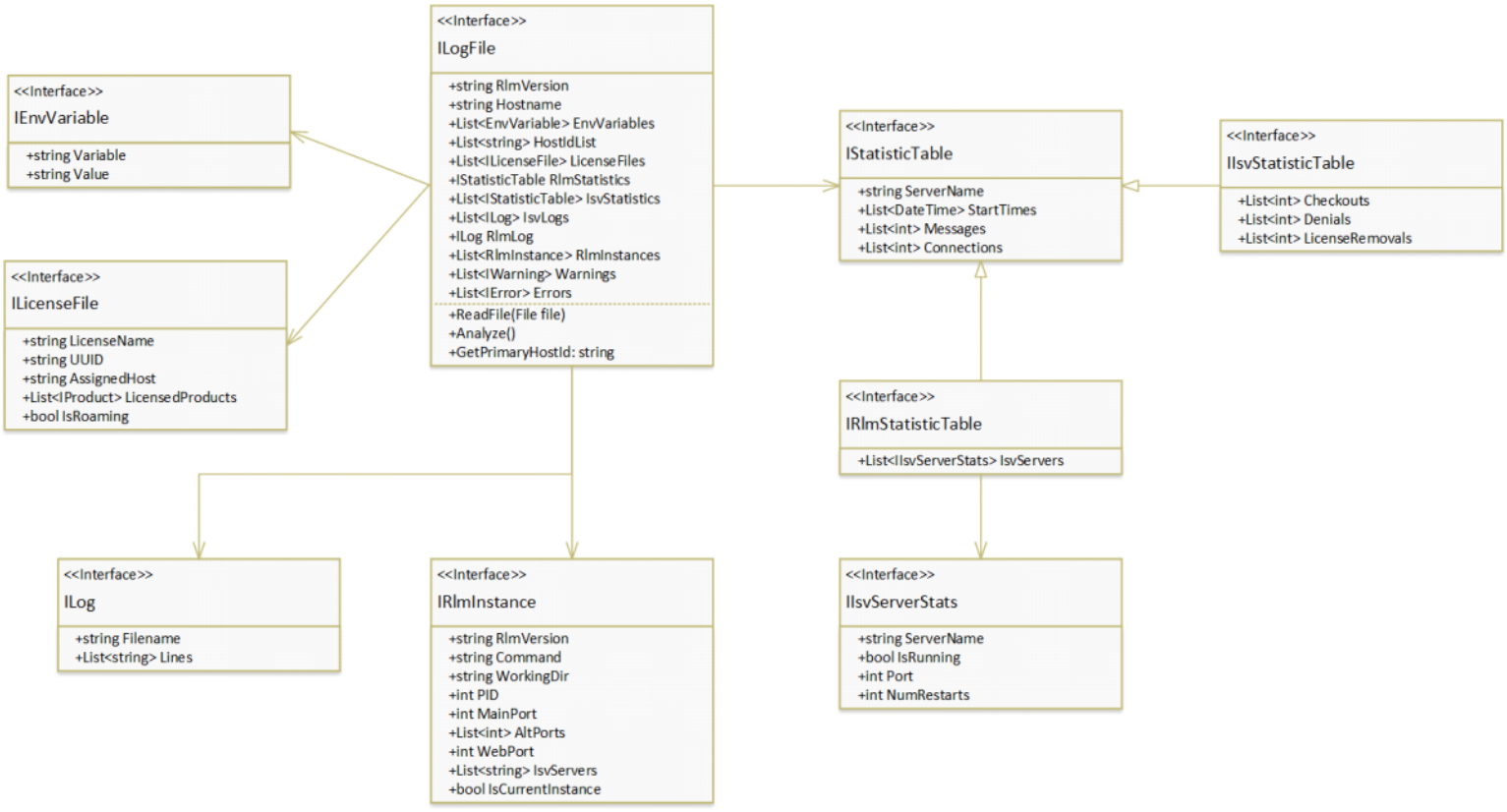
All of this data will be necessary to represent a single log file and linked to one another. Because this is such a complicated model, it will require a complex system of classes in .Net Core and Entity Framework Core to accurately represent in a SQL database.

Server Log UML v1

Tuesday, March 3, 2020 12:37 PM

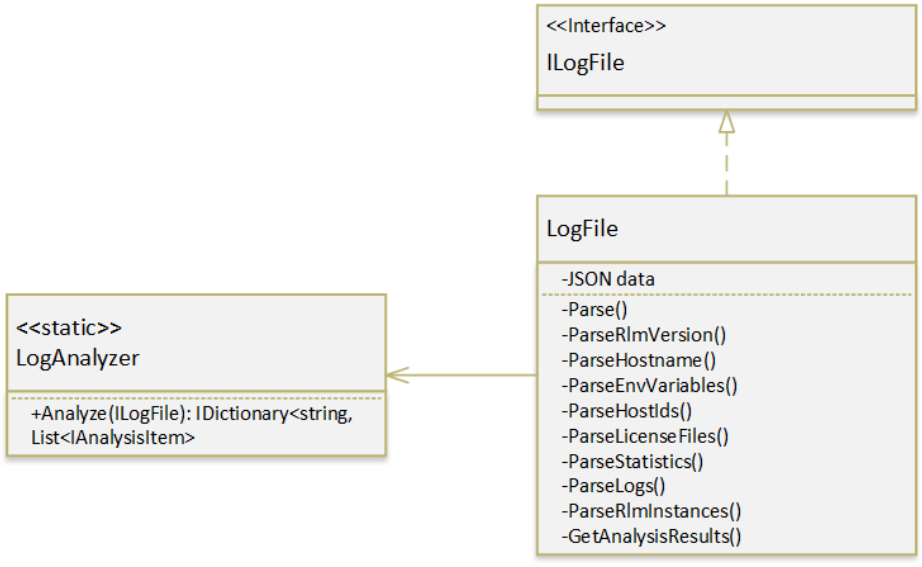
Server Log UML - Overview

The UML design below displays the total UML design for containing a server log file in C# classes.



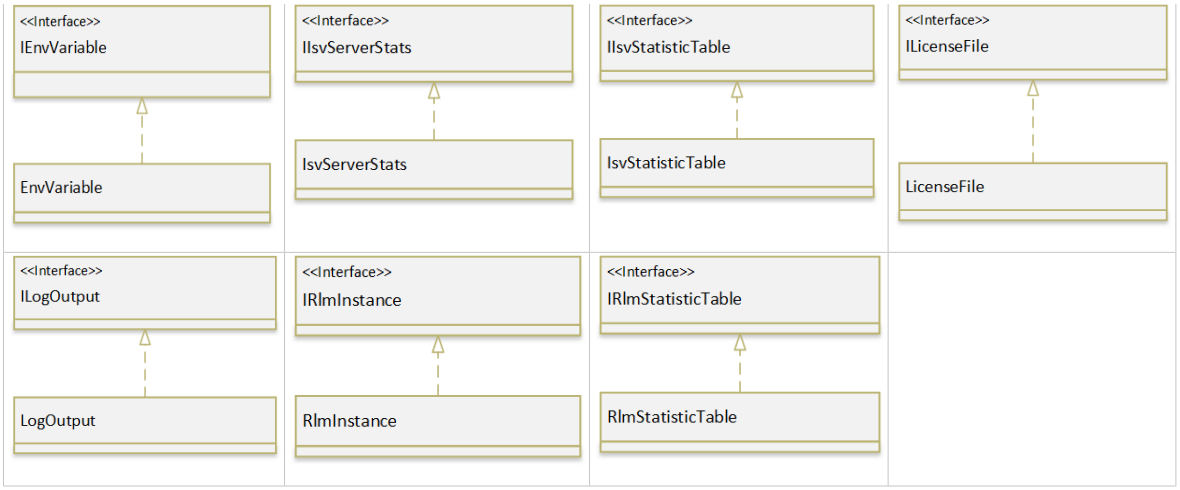
Implementations - LogFile

Each of the interfaces will be implemented by a concrete class. None of these classes will perform any special tasks or contain any special data, except for the LogFile class. Each class simply represents a concrete version of its interface.



The concrete LogFile class will need to either implement an analysis method itself or, more appropriately, this behavior should be encapsulated in a different (potentially static) class - The LogAnalyzer.

Implementations - Others

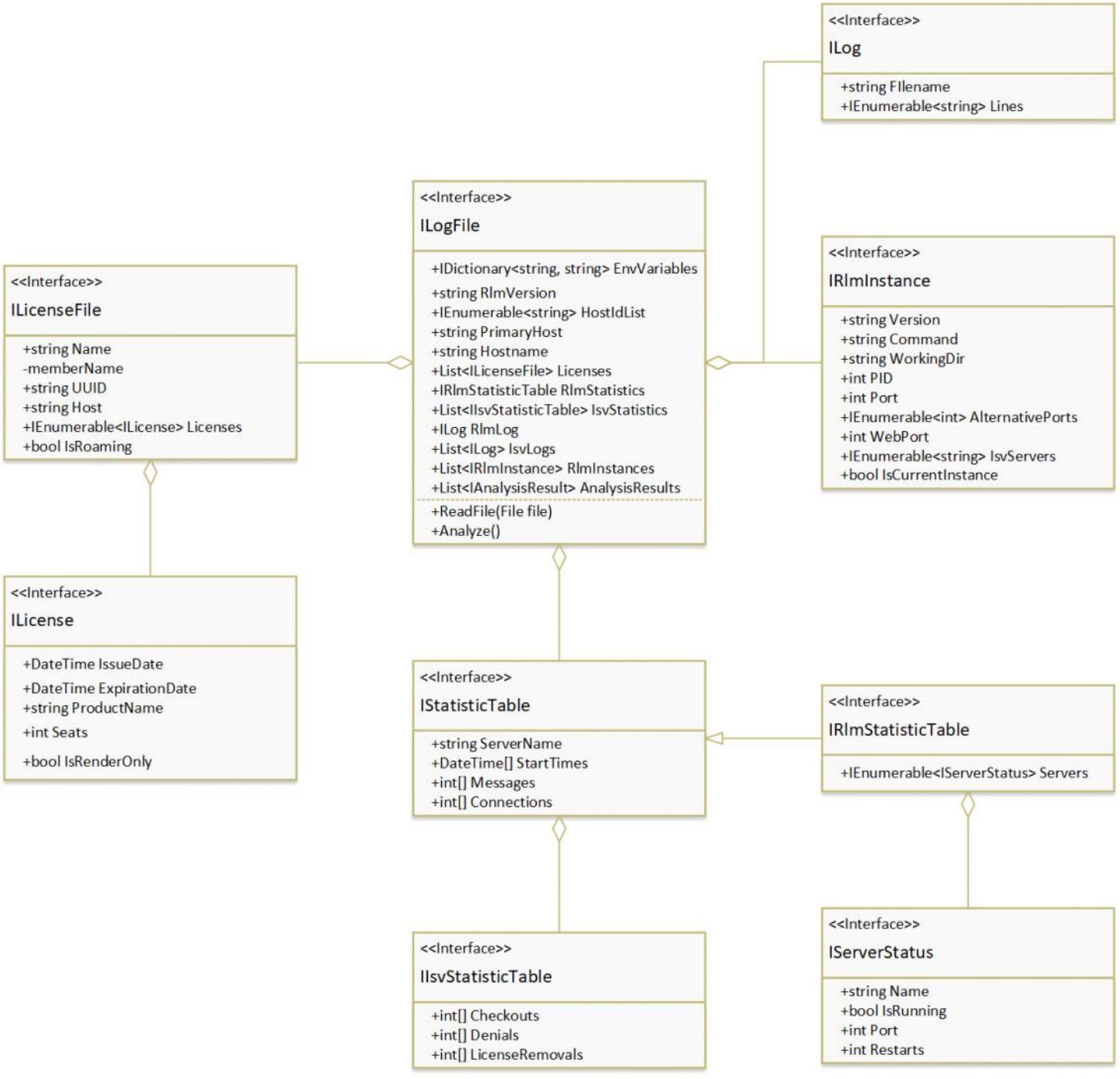

<<Interface>>
IEnvVariable
EnvVariable
<<Interface>>
IIsvServerStats
IsvServerStats
<<Interface>>
IIsvStatisticTable
IsvStatisticTable
<<Interface>>
ILicenseFile
LicenseFile
<<Interface>>
ILogOutput
LogOutput
<<Interface>>
IRlmInstance
RlmInstance
<<Interface>>
IRlmStatisticTable
RlmStatisticTable

Server Log UML v2

Sunday, March 8, 2020 11:29 AM

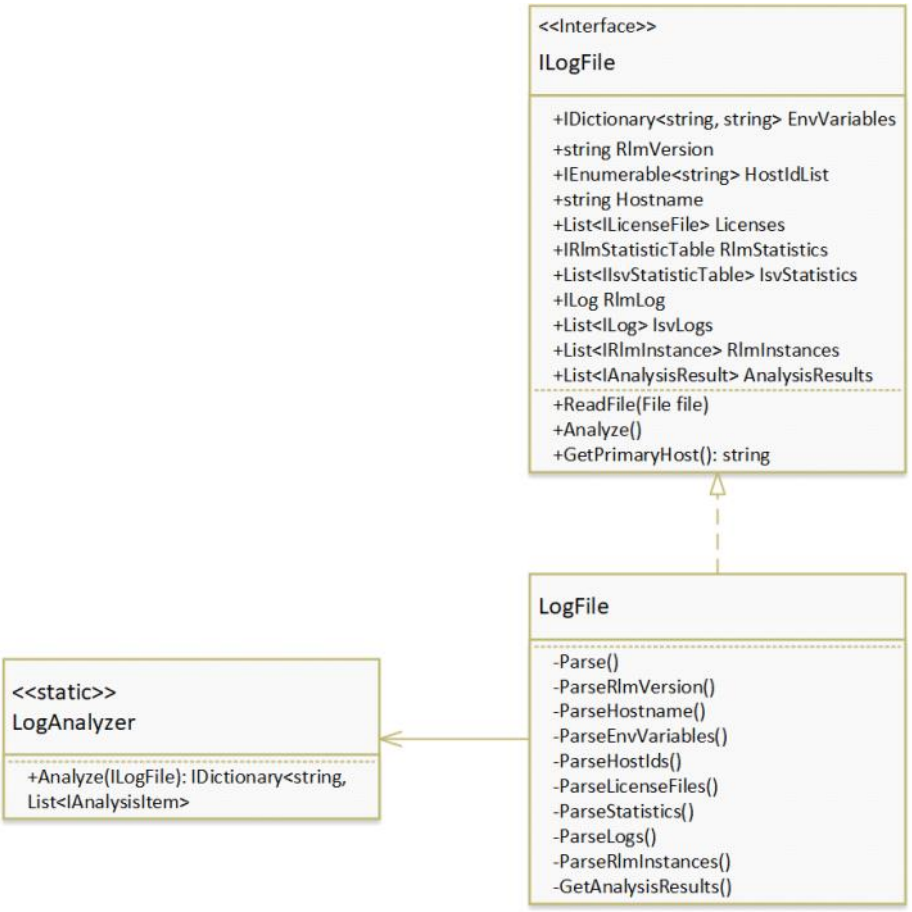
Version 2

I refactored the main family UML for the log file, correcting the relationship types and removing some unnecessary classes (like Environment Variables - They can be simply stored as a dictionary of <string, string> values.



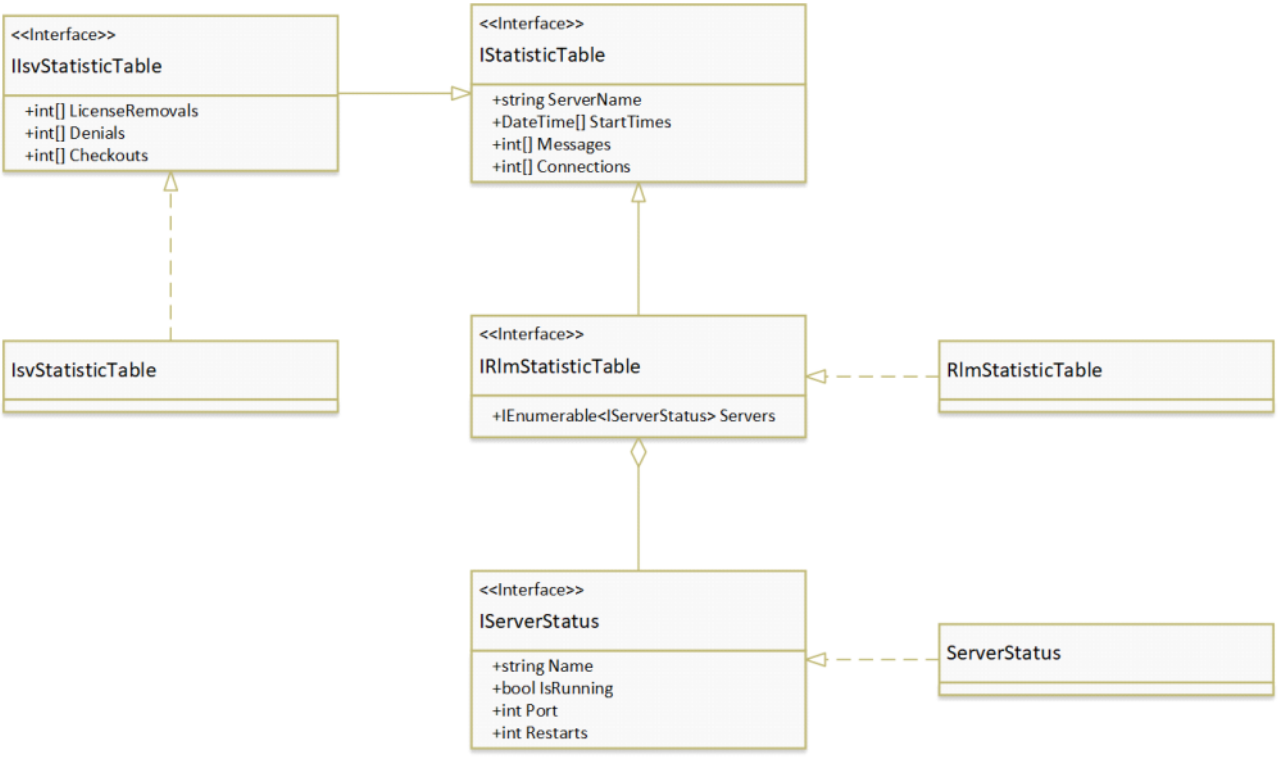
LogFile v2

I have filled out the UML for the LogFile concrete class, as well. I will likely encapsulate the analysis of a LogFile to a s tatic method/class, removing this responsibility from the data model.



Statistic Tables v2

The various interfaces and implementations of the statistic tables are more complicated than I originally thought; It's a good thing I took the time to think it out and make a solid design.

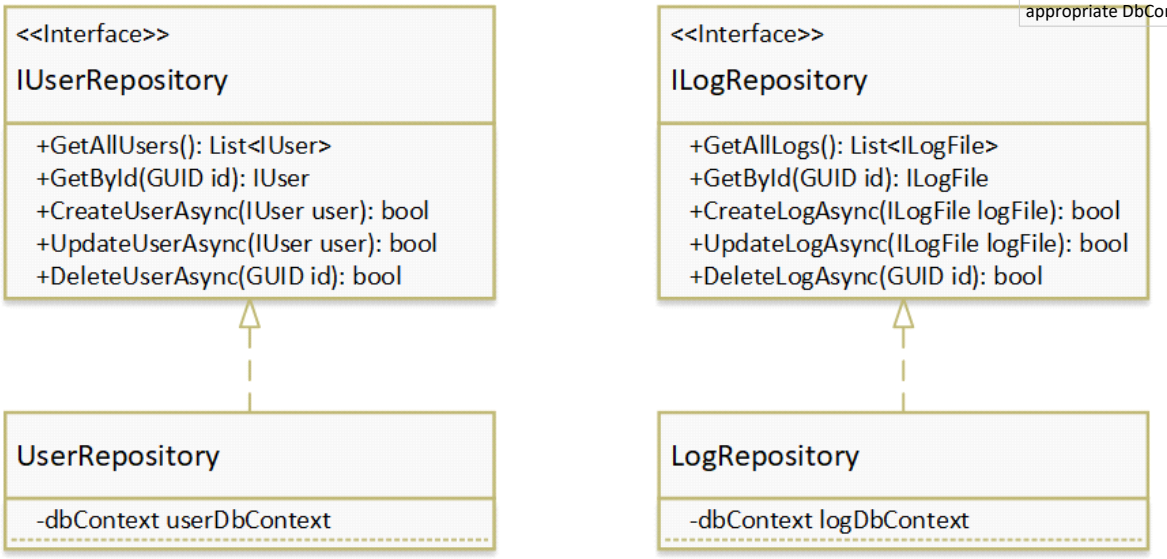


Repository v2

The repositories have been simplified, in that each interface is its own separate interface instead of inheriting from some top-level "IRepository" object - This was unnecessary and didn't offer any real benefit.

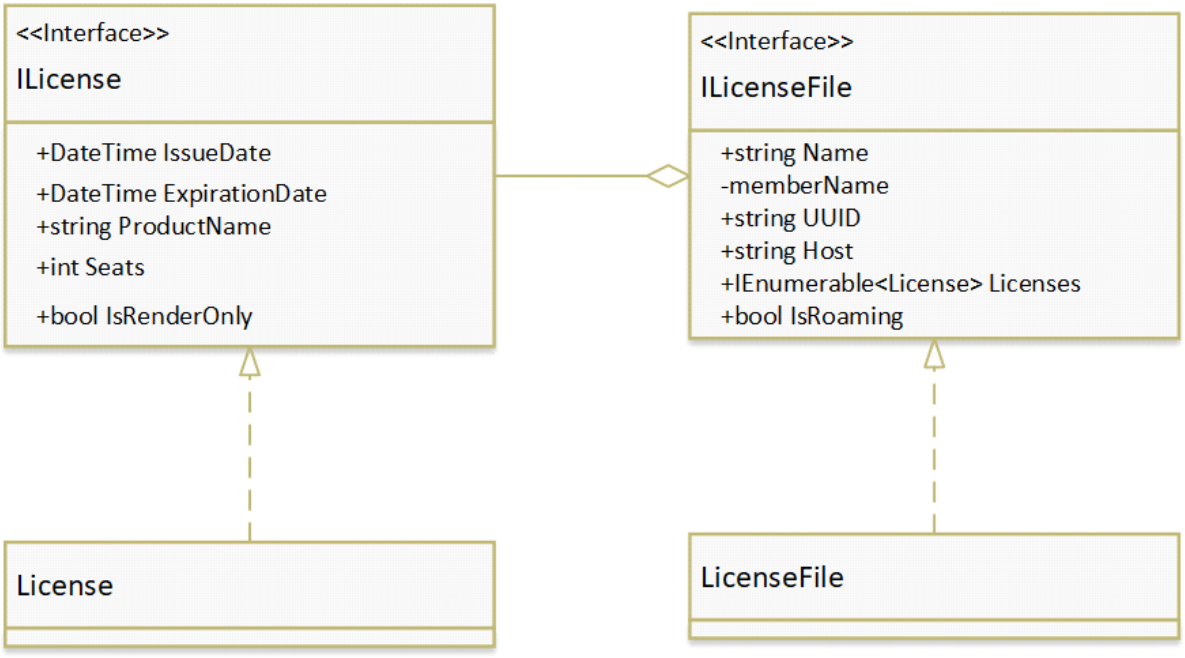
I also updated the design to acknowledge the fact that the Create, Update, and Delete methods will likely be asynchronous, while the Read methods will be synchronous (not sure why). This has been a repeated pattern in tutorials I've been following as well as in the Web Development class. I would like to standardize the methods so that they are *all* asynchronous, but it doesn't appear that the DbContext class (EF Core) offers asynchronous GET methods.

UPDATE 3/12/2020
I don't think I will end up using the repository pattern.
The controllers will each have a Service that will talk to the
appropriate DbContext.



License File v2

I also realized I had forgotten to even design the "ILicense" interface altogether.

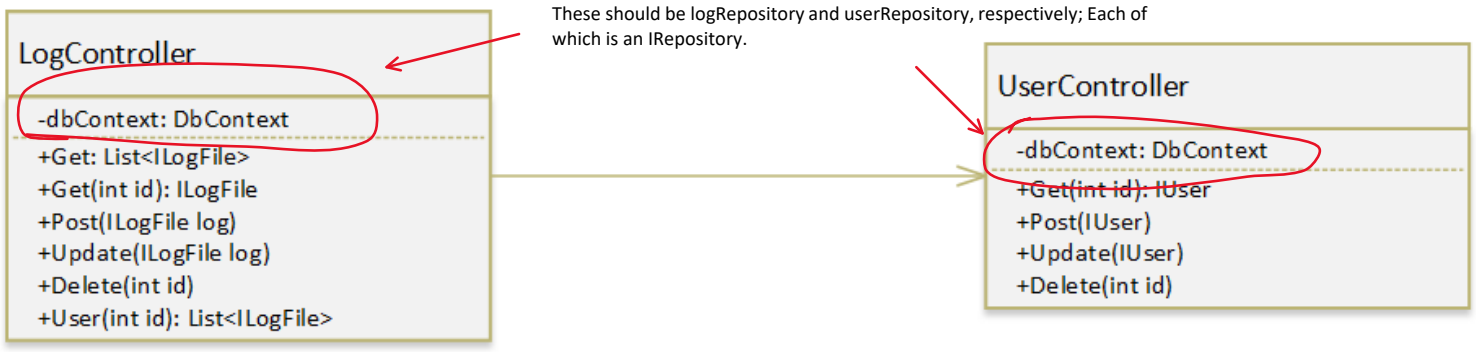


Backend Design

Tuesday, March 3, 2020 12:06 PM

The application will consist of two pieces: front-end and back-end.
The backend will consist of a .Net Core "webapi" application utilizing the REST architecture.

There will only need to be two endpoints (controllers) - The application focuses entirely on the server logs, so there will be one controller for the LogFiles and one controller for Users.



This may turn out to be a simple REST API with a highly complicated parsing algorithm for the log data itself. Luckily, the database work should be taken care of by Entity Framework and the Repository Pattern.

Repository Pattern

The repository pattern will encapsulate the database actions necessary to manipulate the data objects (Log and User). This means that, instead of calling the userContext directly, I will use an intermediate userRepository and call simplified methods like userContext.Get or userContext.Delete; This will be similar for the Log files.

This decouples the Controllers from the database layer and makes it much easier to test.

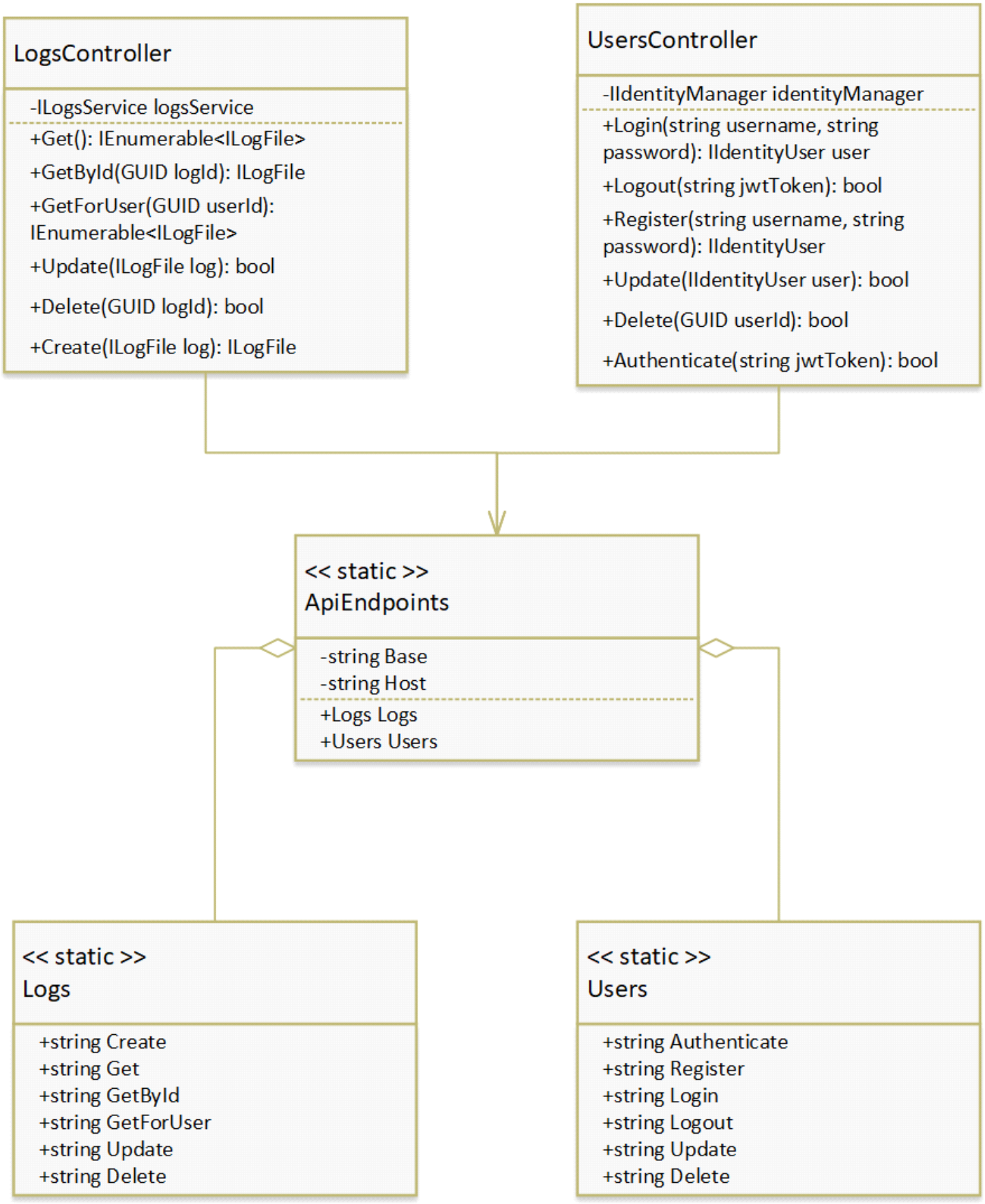
UPDATE:
This information is no longer accurate; See the next pages.

Controllers v2

Sunday, March 8, 2020 10:25 PM

For the time being, I believe we will only need two basic API controllers for the webapi backend: One for the logs and one for the application users. Most of the user management and authentication will be handled by DotNet Core, but I will need to expose the endpoints to allow users to login and logout, as well as endpoints to allow administrative users to create, update, and remove users.

The logs controller will be a basic RESTful API, providing the necessary CRUD methods, as well as a method to get all the logs belonging to a particular user - This will be useful for basic users to see all of their logs and for administrators to filter by user.



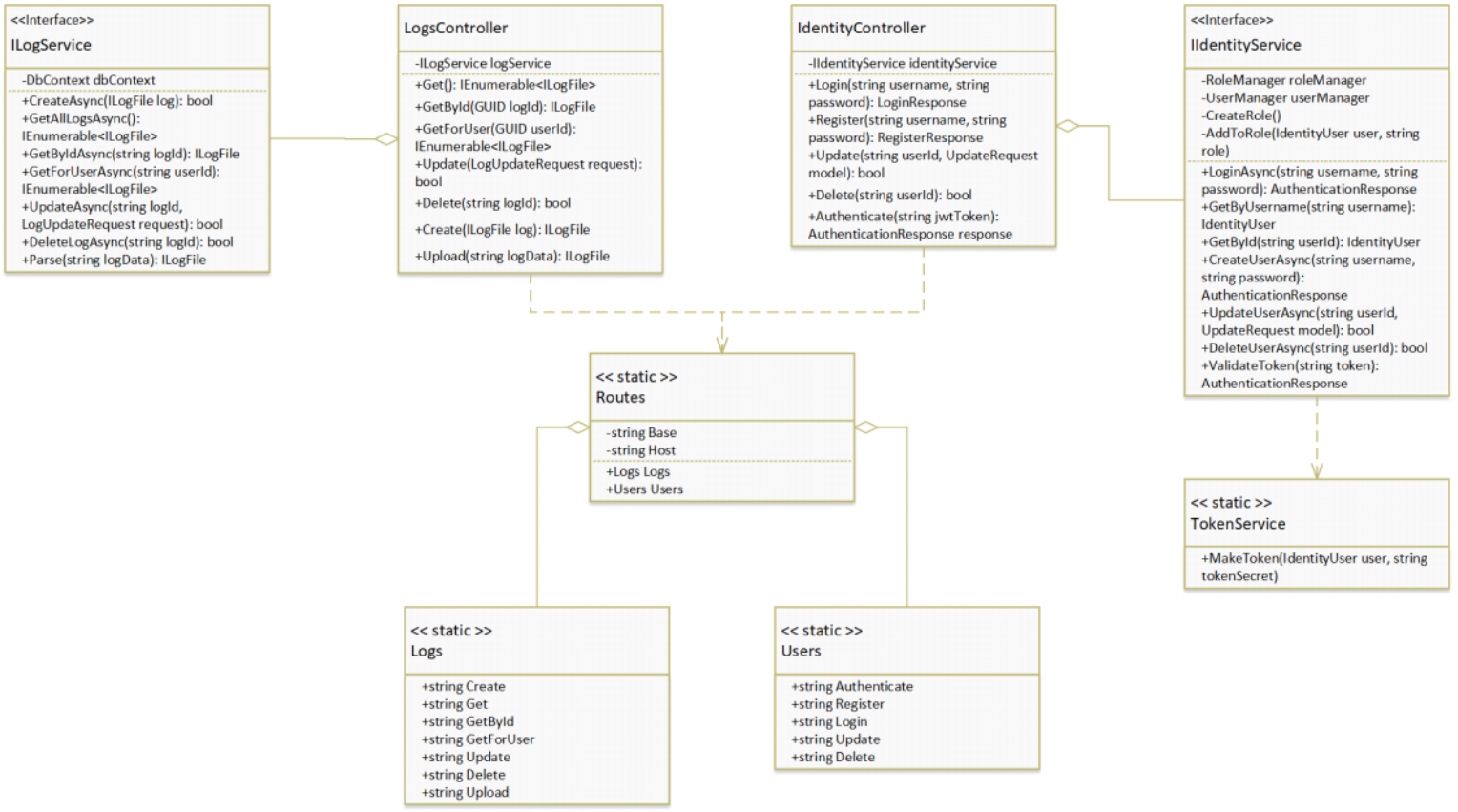
One technique that I picked up from a DotNet Core tutorial was storing the API endpoint routes in static classes, so that any references to these routes are encapsulated, reducing the chance of making errors and reducing duplicated strings in code around the app.

With this setup, each endpoint in the Controller would have a Route attribute similar to this:
`[Route(ApiEndpoints.Logs.Create)]`

Controllers v3

Wednesday, March 11, 2020 10:41 PM

After learning more about DotNet Core and building a test API, I have developed a more thorough and accurate design of the two controller classes and their service interfaces.

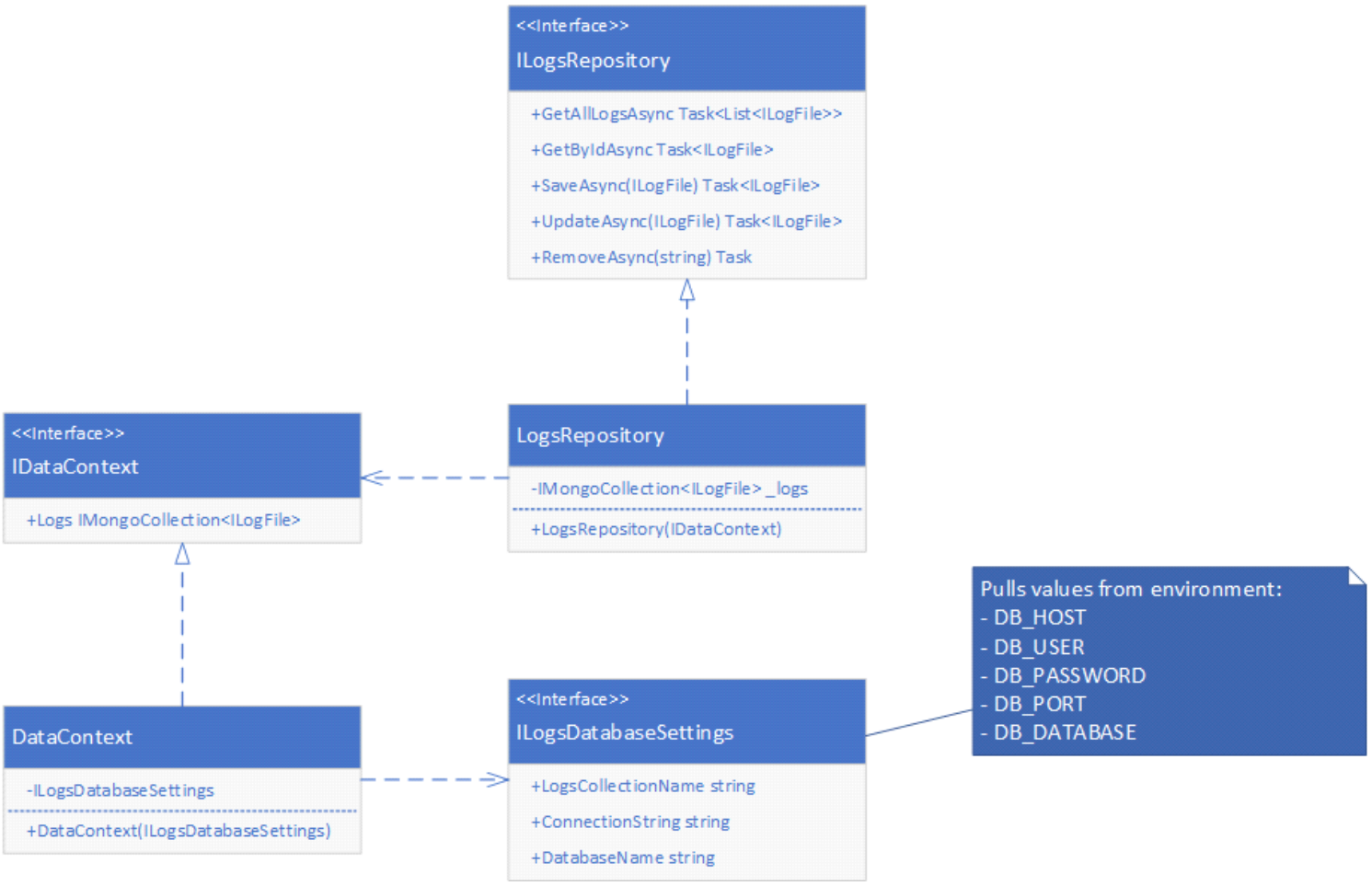


It's a very large diagram, but we essentially use the controllers to direct the endpoints to the appropriate service method, performing authorization where necessary and packaging the responses for the client application.

Data Context UML

Tuesday, April 14, 2020 8:04 AM

On April 13, I built the data context classes allowing the LogsRepository class to receive a fully-connected MongoCollection of ILogFile through dependency injection.



Testing

Tuesday, April 14, 2020 9:56 PM

I will try to be utilizing a Test-Driven Development (TDD) approach as much as I can by creating interfaces that expose methods and properties (contracts) that are the most useful, then designing and writing unit tests that probe these interfaces in both sunny- and rainy-day scenarios.

I will be using NUnit and FakeItEasy as my testing frameworks. I believe NUnit is provided by Microsoft while the FakeItEasy package had to be added through NuGet.

The FakeItEasy package makes it incredibly easy to mock interfaces and function calls, allowing a set of unit tests to focus on the actual System Under Test (SUT) without worrying about the implementation of its dependencies.

For example, to create a Mocked instance of an interface, IExampleInterface, we would just use this line of code:

```
var fakeInterface = A.Fake<IExampleInterface>();
```

Then, if the SUT relies on a method exposed by this interface, we can mock that specific method or property and determine its return value:

```
A.CallTo(() => fakeInterface.MethodOrProperty()).Returns(SomeReturnItem);
```

If the method exposed on the interface requires an argument:

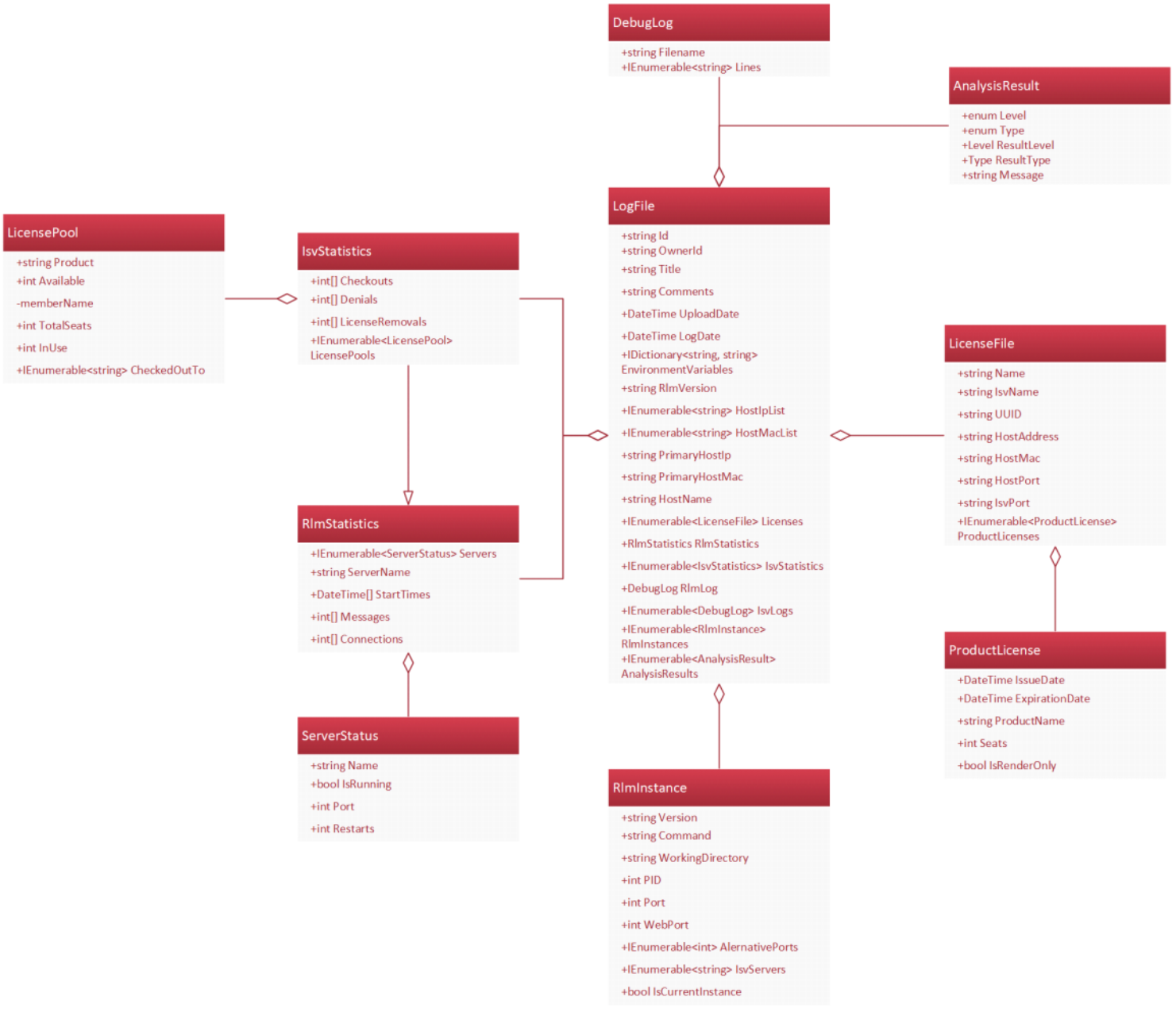
```
A.CallTo(() =>
fakeInterface.Method(A<ObjectType>.Ignored)).Returns(SomeReturnItem);
```

FakeItEasy Website: https://fakeiteasy.readthedocs.io/en/stable/

Final Server UML

Wednesday, June 10, 2020 10:53 PM

This image describes the final LogFile class and all its aggregate components.



EPICs

Thursday, March 12, 2020 7:59 AM

The following Epics should be fulfilled by the new Red Giant Diagnostics application:

- User Auth[oriz|entic]ation
  - Authenticate via Registration/Login
  - Authorization - Restrict access to resources to authorized users
- Administration
  - Administrative roles can modify user and log data
- Log Analysis
  - Users can upload log files to app
  - Log data is parsed by the app
  - Log data is analyzed by the app
  - Log data and analysis results are presented in easy-to-read format
- Log Save & Retrieve
  - Log data can be saved to database
  - Log data can be retrieved from database at a later time
  - Log data and analysis results can be shared between users
- Metadata
  - Data about the application itself is available
  - Trends from logs are analyzed and presented

Features

Thursday, March 12, 2020 10:10 PM

In addition to the EPICs, some basic features must be included:

- Logs are parsed and analyzed; errors and complications are presented in a noticeable, readable manner
- Basic log analysis does not require a login, but only registered users can save logs to the database
- Saved logs can be shared via a URL or shared directly in the app

Stories

Thursday, March 12, 2020 10:12 PM

1. As a registered or unregistered user, I want to upload a log to be analyzed, so that I can quickly and easily discover any problems and form a resolution for the customer.
2. As a registered user, I want to save logs to the app, so that I can retrieve them later.
3. As a registered user, I want to save logs to the app, so that I can share them with my colleagues.
4. As a registered user, I want to be able to edit my profile, so that my colleagues have up to date information about me and can see my accurate contact information.
5. As an administrator, I want registration to be a privileged task, so that we can ensure only the right people have access to the tool and the saved log data.
6. As an administrator, I want to be able to manage the application users, so that I can create accounts for new team members and remove accounts when people are no longer around.
7. As an administrator, I want to be able to manage the stored logs, so that I can edit log metadata or remove logs from the application.
8. As a registered user, I want to see application metadata, so that I can see usage trends for the app.
9. As a registered user, I want to see application metadata, so that I can see what the most common errors are and be proactive against them.

REST API

Saturday, March 14, 2020 8:56 AM

The DotNet Core backend is going to use a REST pattern for its API. This pattern will consist of the endpoints detailed below.

Identity

The identity portion of the API doesn't have a RESTful structure, but it is modeled after DotNet Core's default pattern.

Login

URL	identity/login
Method	POST
Parameters	LoginRequestModel (string: username, string: password)
Returns	LoginSuccess (string jwtToken, string username): 200 OK LoginFailure (string errorMessage): 401 Unauthorized

Register

URL	identity/register
Method	POST
Parameters	RegisterRequest(string username, string password)
Returns	RegisterSuccess: 200 OK RegisterFailure (string errorMessage): 403 Forbidden, 409 Conflict

Logout

I'm not entirely sure if a logout method will be necessary, since I'm using an API backend with a separate frontend application and not a closely-linked frontend. I'll have to play around with this to see what it would provide, but the endpoint will be structured as such:

URL	Identity/logout
Method	POST
Parameters	LogoutRequest(string jwtToken)
Returns	204 NO CONTENT

Update: 3/15/2020
After playing with the test API, I don't think there will be a need for a Logout endpoint.

Log

The log controller will use a RESTful API structure, exposing CRUD endpoints for users and administrators.

Create

To create a new log file object in the database.

URL	log
Method	POST
Parameters	LogUploadRequest(LogUploadModel upload)
Returns	LogCreationSuccess(string logId): 201 Created LogCreationFailure(string errorMessage): 400 Bad Request, 403 Conflict

Analyze

Since the application will allow logs to be analyzed without being saved, a separate endpoint must be open for analysis.

URL	log/analyze
Method	POST
Parameters	LogAnalysisRequest(LogUploadModel upload)
Returns	LogAnalysisResults(LogViewModel log): 200 OK LogAnalysisFailure(string errorMessage): 400 Bad Request

Get (by ID)

Searches for a log by its ID, returning the log, if found. This endpoint will be protected, meaning only logged-in users can retrieve data.

URL	log/{id}
Method	GET
Parameters	LogRetrievalRequest(string logId)
Returns	LogRetrievalSuccess(LogViewModel log): 200 OK LogRetrievalFailure(string errorMessage): 403 Forbidden, 404 Not Found

Get (for user)

This endpoint will return a list of log ids for all logs owned by a particular user.
A standard user can only retrieve their own logs; Admins can retrieve logs for any user.

URL	log/user/{id}
Method	GET
Parameters	UserLogsRequest(string userId)
Returns	UserLogsSuccess(List<string> logIds): 200 OK UserLogsFailure(string errorMessage): 404 Not Found, 403 Forbidden

Update

URL	log/{id}
Method	PUT
Parameters	LogUpdateRequest(string logId, LogUpdateModel update)
Returns	LogUpdateSuccess: 200 OK LogUpdateFailure(string errorMessage): 403 Forbidden, 404 Not Found

Delete

URL	log/{id}
Method	DELETE
Parameters	LogDeleteRequest(string logId)
Returns	LogDeleteSuccess: 204 No Content LogDeleteFailure(string errorMessage): 403 Forbidden, 404 Not Found

User

Administrators will have the ability to create and remove users from the application.
Users will also have the ability to modify their user data.

Create

URL	user
Method	POST
Parameters	UserCreationRequest(string username, string password)
Returns	UserCreationSuccess(string userId): 201 Created UserCreationFailure(string errorMessage): 403 Forbidden, 409 Conflict

Get (by ID)

URL	user/{id}
Method	GET
Parameters	UserRetrievalRequest(string userId)
Returns	UserRetrievalSuccess(UserViewModel user): 200 OK UserRetrievalFailure(string errorMessage): 403 Forbidden, 404 Not Found

Update

URL	user/{id}
Method	PUT
Parameters	UserUpdateRequest(string userId, UserUpdateModel update)
Returns	UserUpdateSuccess: 200 OK UserUpdateFailure(string errorMessage): 400 Bad Request, 403 Forbidden, 404 Not Found

Update Password

URL	user/{id}/password
Method	PUT
Parameters	UserPasswordUpdateRequest(string userId, string newPassword)
Returns	PasswordUpdateSuccess: 200 OK PasswordUpdateFailure(List<string> errorMessages): 400 Bad Request, 403 Forbidden, 404 Not Found

Delete

URL	user/{id}
Method	DELETE
Parameters	UserDeleteRequest(string userId)
Returns	UserDeleteSuccess: 204 No Content UserDeleteFailure(string errorMessage): 403 Forbidden, 404 Not Found

Planning

Wednesday, April 8, 2020 7:44 AM

4/6/2020 - 4/19/2020

After taking a break from planning and design, the new school term has begun.
This sprint will lay the foundations for the rest of the project:

- Set up project in DevOps
- Create the Docker containers for the DotNet Core backend, Vue.js frontend, and a SQL Server database

Breakdown

I don't expect these tasks to take up the full sprint, so I will likely begin pulling in tasks and user stories beyond the tasks initially assigned to this sprint.

DevOps

The project will be tracked by using Microsoft's DevOps service. For this sprint, I will need to create the project in DevOps and begin fleshing out the epics, user stories, and tasks. This will, itself, be a task.

I had previously investigated using DevOps as the primary code repository, but I would like to keep all my projects in my personal GitHub. By the end of this sprint, I will need to have made a final decision about where I'm storing my code.

Docker Infrastructure

This sprint will be primarily focused on getting the infrastructure set up for the rest of development and into production. The main needs include setting up the Docker environment so that the DotNet Core backend is containerized, the Vue.js frontend is containerized, and a SQL Server/PostgreSQL server database is containerized. This will allow me to easily swap the database or move to a hosted database in the future, while only needing to change minimal configuration.

Tasks

- [x] Set up Red Giant Diagnostics project in DevOps
- [x] Create epics and main user stories in project board
- [x] Create DotNet Core container and Dockerfile
- [x] Create Vue.js container and Dockerfile
- [ ] ~~Create SQL/PostgreSQL container and Dockerfile~~
- [x] Create MongoDB container in Docker Compose

Edit: 4/21/2020

April 8, 2020

Wednesday, April 8, 2020 10:05 AM

I have created the project in DevOps (dev.azure.com) and have begun fleshing out the epics, features, backlog items, and some of the initial tasks.

I think all the epics are present in the DevOps board now along with all of the features that I can think of ahead of time.

I have also added backlog items and tasks for the foundational steps:

- Create the backend application
- Containerize the backend application
- Create the frontend application
- Containerize the frontend application
- Containerize a SQL database
- Create a base Docker Compose file with configuration common across development and production
- Create a development Docker Compose file with configuration specific to development of the application
- Create a production Docker Compose file with configuration specific to production of the application

I don't expect these tasks to take long to complete, so each one has been given an effort of 1

The Sprint 1 tasks have been assigned and the burndown chart is live.
I currently have set myself a capacity of 4 per day, giving myself a total of 52 effort points per sprint (Monday through two Sundays). This sprint currently has a total of 8 effort points, so I expect to begin pulling tasks from the backlog after a couple days of work.

April 10, 2020

Friday, April 10, 2020 3:36 PM

I have started building the infrastructure for the application and have so far completed setting up the api and web client projects, as well as completed the Dockerfiles for each image.

I am now setting up the database project, ~~which should really only consist of a Dockerfile for a PostgreSQL container.~~
Actually, I don't think I will need a Dockerfile for the PostgreSQL container; at least not at this time. I should be able to configure and launch it directly from the docker-compose file.

PostgreSQL - Entity Framework Core
https://entityframeworkcore.com/providers-postgresql

Dockerize PostgreSQL
https://docs.docker.com/engine/examples/postgresql_service/

PostgreSQL - Docker Hub
https://hub.docker.com/_/postgres/

Default PostgreSQL Data Location
`/var/lib/postgresql/data`

April 12, 2020

Sunday, April 12, 2020 10:40 AM

Easter Sunday

10:45am
Some severe storms are rolling into the South today and will be lasting until tonight/early tomorrow morning.
I am going to try and get some work done, but if I lose power that may be the end of it for the day.

I have made significant progress on my initial tasks and have the projects created and have been working on the Docker Compose configurations. Luckily, I have been developing other applications over the last few months that have given me time to practice and work out many of the issues that I will be facing in this project; This will allow me to breeze through some of the obstacles that would have been major roadblocks.

Today, I would like to finish the Docker Compose configurations and get both projects up and running in Development and Production modes. Once I get that complete, I will examine the backlog and add a few more items to my task list for this first sprint.

I am starting to realize that re-writing the entire application from the ground up in one term could be difficult to accomplish, but I have confidence that the Red Giant RLM Manager, which consists of a Node.js backend API and a Vue.js frontend web app, and the TestAPI project, which consists of a DotNet Core Web API backend, has given me the experience to quickly ramp this project up. I have been able to use these projects as templates for much of the configuration and set up already.

1pm
I have already made significant progress and have begun adding more items to the backlog to begin implementing the application.

I would like to make a serious effort to use a Test-Driven Design (TDD) methodology, so maybe my next tasks should be to add a Test project and begin building tests for the API.

7pm
I think the log file data is too complex to use EF Core for a code-first approach to persistence, so I have begun researching the MongoDB .Net package. I used MongoDB for the original application, so it may be possible to use this framework and not lose any data.

MongoDB URLs

MongoDB.Driver	https://mongodb.github.io/mongo-csharp-driver/2.10/getting_started/quick_tour/
MongoDB DotNet	https://docs.mongodb.com/drivers/csharp
MongoDB Nuget	https://www.nuget.org/packages/MongoDB.Driver
Microsoft MongoDB Tutorial	https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mongo-app?view=aspnetcore-3.1&tabs=visual-studio

April 13, 2020

Tuesday, April 14, 2020 7:57 AM

I spent most of the day figuring out how to pattern the repository/data context/services for the Logs in a way that would apply Dependency Injection (DI) and allow me to unit test each class.

I ended up with an IDataContext interface with a single property, **Logs**, that returns an IMongoCollection of ILogFile objects. In the implementation of the concrete DataContext class, this property instantiates a MongoClient, gets the Database, and then gets the collection of Logs from the MongoDB.

The DataContext constructor takes an ILogsDatabaseSettings interface through DI; This interface exposes three properties: LogsCollectionName, ConnectionString, and DatabaseName. In the concrete implementation of this interface, the LogsCollectionName is set to "Logs"; The DatabaseName is pulled from the environment variable **DB_DATABASE**; The ConnectionString is pieced together using:

- **Host:** DB_HOST
- **User:** DB_USER
- **Password:** DB_PASSWORD
- **Port:** DB_PORT

This ends with a string like "mongodb://{user}:{password}@{host}:{port}"

April 17, 2020

Friday, April 17, 2020 8:17 AM

I have made significant progress over the last few days, including finishing the LogsController, LogsService, LogsRepository, and all their unit tests. However, once I started testing the actual API performance with Postman I discovered that the Upload method was not set up correctly.

I have been refactoring this method since yesterday and have had success getting it to take a file upload, parse it into string data, and pass it to the LogFactory and LogParser classes. I have made an effort to use multi-threading techniques so that the file upload and the log parsing can both happen on background threads.

I will have to refactor/modify the unit tests (and likely add more tests) now that I have made these breaking changes.

April 21, 2020

Tuesday, April 21, 2020 8:17 AM

It's been awhile since I updated my progress.

Over the last few days, I implemented the log parsing algorithm and can now upload a file to the LogsController and return an ILogFile object (in the form of a JSON object) via the same endpoint.

This was likely the most difficult and time-consuming portion of the application and will probably require some creative testing, but with it done and out of the way I can move on to other areas of the API; Everything else should be relatively easy - With the parsing done, now I will only have to worry about working with ILogFile objects.

I didn't do anything related to this project yesterday, so I will have to treat today as the Sprint Review and Sprint Planning day. At least now I'll have a better idea of how much work I can handle in one sprint, as last time I was just guessing and drastically underestimated my capacity (by somewhere close to 1/3).

I also have a meeting with Sherry today and I'm excited to show all the progress I have made in just the first two weeks of the project.

Sprint Review

Tuesday, April 21, 2020 8:38 AM

It looks like I completed a workload of 50 points over this first sprint. I ended up assigning myself a total of 52, but there were 2 points left over - These points came from a task to update the LogsController.Upload tests after working on the upload and parsing algorithms.

I greatly underestimated my capacity this sprint; Looking at my Sprint Planning notes for Sprint 1, I only assigned myself some of the basic infrastructure and setup tasks, like making the projects and Dockerizing them along with a Docker-DB. I finished those quickly and began creating backlog items for the LogsController and LogsController tests, completed those, and then began working on the file upload and parsing algorithms.

Once I started working on the file upload, I had to modify the LogsController.Upload method and I still need to go back and re-do some of the LogsController unit tests. I'm not entirely sure how to test the parsing algorithms other than parsing a file and making sure it comes out as expected, but I will have to spend some time in Sprint 2 researching and practicing unit testing the various Factories and the static Parse methods - This could actually take up a lot of time.

For this next sprint, I should probably assign somewhere between 40 and 50 points, as I will still be working for Red Giant and working from home for this next sprint so I should be able to get work done on this project during the day. Once I enter Sprint 3 and beyond, I may have to cut back on my capacity since I will be working at Dynetics and won't have any chance to work on this project during the day.

I don't expect to do as much unit testing this next sprint, because the majority of the LogsController tests are already complete - I will likely add more as needed - and the IdentityController will use the .Net Identity Framework, so I won't have to write many in-depth tests for this endpoint.

Once I begin working on the frontend, I will need to learn how to test Vue.js applications, which will likely take up a lot of time, but I don't expect to begin working on the frontend until Sprint 3.

April 21, 2020

Tuesday, April 21, 2020 8:28 AM

Today is the second day of Sprint 2 - I didn't do any work on this project yesterday, so today is a Sprint Review/Sprint Planning day.

I haven't done the review yet, but based on how much I remember accomplishing this last sprint (and taking into account my GUI course, which has taken more time than I expected), I still expect to be able to complete the LogsController and its related classes. I would like to finish this and the IdentityController by the end of the sprint so that I can spend the last two sprints on the frontend app and the project report.

I will have to provide an update once I do my Sprint Review and Planning.

Planning

Tuesday, April 21, 2020 8:46 AM

For Sprint 2, my primary task will be to complete the LogsController and all its unit tests; I don't have a good prediction for how much work this will take, but if I do finish it before the end of the sprint I will either:

a) Begin working on the IdentityController
b) Refactor the LogsController/Tests

How much time is left in the sprint will determine which of these paths I choose. If I don't have much time left in the sprint, I will likely spend the remaining time improving the LogsController and its tests; if I still have quite a bit of time left, I will begin working on the IdentityController.

What I still have left for the LogsController is really just implementation details and doing some manual API testing to make sure I have the API endpoints configured correctly - This was a problem with the Upload method, since I didn't know how to handle a raw file upload in a DotNet controller method. I don't anticipate having many problems with the rest of the LogsController methods, since everything from this point on will be handling the already-parsed ILogFile objects.

LogsController Tasks

There are some manual tests that I would like to complete to find any bugs that the unit tests might not catch; These will all consist of using Postman to send HTTP requests with the log data.

Postman Tests

- [x] Get
- [x] GetById
- [x] GetForUser
- [x] Update
- [x] Delete
- [x] Save

I will also need to update (and likely add more to) the unit tests for the LogsController to make sure I am testing every scenario possible. Luckily, once the log file has been uploaded and parsed, all the other endpoints should be using non-user data - meaning that the users won't be entering any information into the system, it will only be the log data being passed around.

Now that I say that, there will be some additional log data that will be user-generated. In the original application, the users have the ability to save the log data with a Title and some Descriptive text - I would like to include that in this application, as well. This means there is a need for sanitization, but since it's only two text fields, I don't anticipate it being too difficult to clean.

I will have to add a couple fields to the ILogFile object to accommodate a Name or Title field and a Description field. Luckily, adding two string properties to the interface won't require many (or any) additional unit tests - It's just more data, not methods.

April 25, 2020

Saturday, April 25, 2020 2:45 PM

I haven't been keeping notes these last few days, but I had to refactor all of the factories and parsing algorithms to make them testable. They were originally all static classes, but this left me unable to test them as no mocking frameworks can mock static methods or classes.

To make them testable, I refactored all of the factories into interfaces and concrete implementations. I have since been writing unit tests for all of the factories and am almost complete; I hope to be finished with the factory unit tests by the end of day today. It would be a tremendous milestone to finish all the tests for the parsing algorithms, so that I can make sure I don't break the parsing anywhere down the line. These algorithms are also the most intensive algorithms in the application, so to be assured that they are still working correctly is a huge benefit.

I also had to refactor the data models to not use interfaces and rely on concrete implementations; I don't like this design, but it was the only way I could find to reliably serialize and deserialize between the complex models and JSON strings.

April 27, 2020

Monday, April 27, 2020 10:23 AM

I finished the unit tests for the parsing algorithms and factory classes this morning. I am testing the functionality using Postman and can confirm that the logs API is working correctly!

I can upload a log file through Postman to the Logs Upload endpoint and it returns a JSON LogFile object; I can then upload this JSON data to the Save endpoint and see the log saved to the database, which is then returned as a JSON object with a legitimate ID string.

My next steps are to develop and test the Identity and Authentication features of the API, this will assign an "Owner ID" to the logs and allow users to retrieve all the logs they have created.

This will closely mirror the functionality of the original application, in that only users in the Admin role will be able to see all logs generated by any specific user, but individual logs can be shared and viewed by any user. Only Admins and the log owner can edit log details.

My tasks for the rest of the sprint are:

- [x] Create IdentityController
- [x] Incorporate DotNet's Identity framework
- [x] Create IdentityService
- [ ] Add "Admin" role
- [x] Test IdentityController
- [x] Test IdentityService
- [ ] Implement Authentication
- [ ] Implement Authorization

I have a feeling I may need to set up a SQL server for the Identity framework, as I don't know if it can use a Mongo database; I will need to research this.

UPDATE: It looks like I can use a MongoDB database with the Identity framework, but it will take some setup that I haven't tried before.

I think I will just keep the User data in a separate database (Postgres, probably) and the Log data in the Mongo database.

April 28, 2020

Tuesday, April 28, 2020 4:21 PM

I have made substantial progress on the identity portion of the application. I have created and fully tested the Identity setup services, the IdentityService, the IdentityController, and the TokenService for handling JWT tokens and Authentication. User data is stored in a PostgreSQL database while the logs are stored in a MongoDB database; This makes more sense, given that user data is tabular and the log data is naturally more of a document.

I have been doing good about sticking to a TDD methodology, designing the interfaces, stubbing out a concrete class, writing all the unit tests and figuring out how the application should behave before writing the actual implementation. I occasionally forget something or need to change something later down the line, but I have been keeping up with keeping the tests up-to-date.

I have been having a strange issue with VSCode and the unit test framework where clicking the "Run Test" or "Debug Test" links will suddenly break - I have to close and restart VSCode to get that functionality back; It's been happening quite often as my test bank grows. I found one Stack Overflow article that said something about editing and saving rapidly breaking this feature in a far previous version, so I turned off Autosave and it has helped tremendously, but it does still happen.

April 29, 2020

Wednesday, April 29, 2020 10:09 AM

I have basically completed the backend API by the point - The Logs and Identity controllers are complete and fully tested, along with all the services they depend on. I think I might need to add a simple endpoint for the front-end app to validate and/or refresh its current token.

I also think I should change the LogsController so that it returns a list of log IDs for the GetAllLogs and the GetLogsForUser endpoints instead of returning all of the log data. This will help speed the load time, sending a list of small ID strings instead of potentially dozens of full log files; Once the front-end has the log IDs (and perhaps the owner IDs), the logs can be individually fetched.

UPDATE: 2:20PM
I have made these changes and will be updating the tests accordingly.
I also made a similar change for when a user is registered - The controller now sends a UserRegistrationResponse containing the new user ID and a login token.

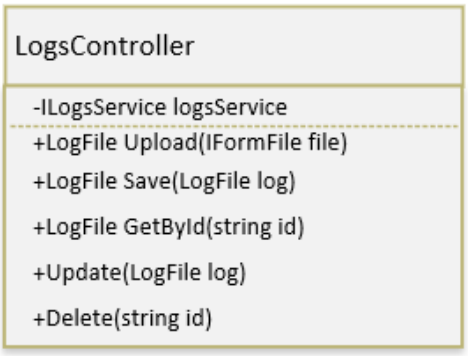
April 30, 2020

Thursday, April 30, 2020 10:19 AM

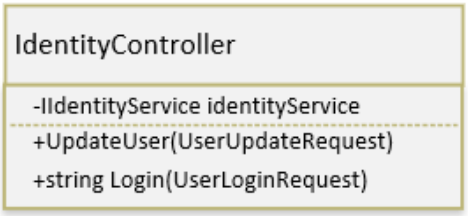
In trying to implement authentication and authorization I have made the realization that I will need to split off the Administrative tasks into their own controller. I should be able to utilize the identity service in this controller, so it won't be too much work to break these tasks out.

The reason I need to do this is that I'm trying to protect the identity actions and restrict access to two sets of users: The current user and Administrators. Users should be able to update their own information, but no one else's, and Administrators should have the rights to modify any user's information.

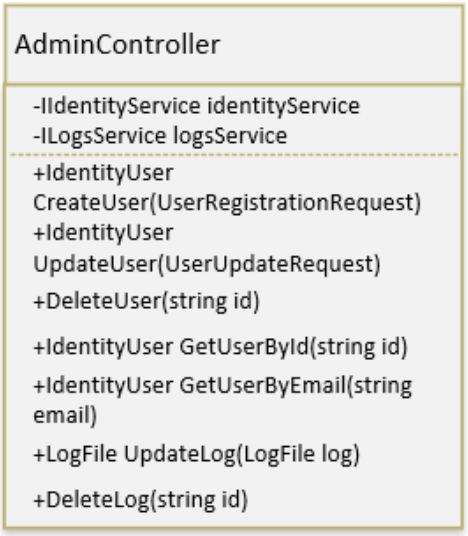
I believe I will have to refactor the controllers to match the diagram below:



The LogsController will retain most of its current functionality, as users should be allowed to upload, save, edit, and delete their owned logs.



The IdentityController will be significantly reduced, only allowing users to edit their own information and anonymous access to the Login endpoint. I may need to include an endpoint to retrieve user data from a token, so that the front-end can refresh its info from a token saved in a cookie.



The AdminController will be entirely new and will have quite a bit of functionality, as Administrators should be able to create, read, update, and delete both users and logs without regard to who the user is or who the log is owned by.

The reason I am splitting things apart like this is for **Separation of Concerns** and to ease unit testing.

Separation of Concerns

My original plan was to have the controllers allow actions based on who was submitting the request, but this required a lot of methods to determine who was making the request, if they were an Admin or if they were the owner of the resource. Splitting these responsibilities into different controllers allows the controllers to have specific, simple authorization rules instead of complicated and dynamic rules.

For example, the original plan would have had the IdentityController's Update method receive the request, validate the data, then check who the current user was based on the token, retrieve that user's roles, check if they were an Admin and, if not, check if they were the user affected by the update request. The new method will allow the IdentityController to only check if the request is coming from the affected user; The data validation and authentication will happen

automatically using an AttributeFilter.

If an Admin wants to update a user, they will make a request on the AdminController, which authenticates and authorizes their permissions using an AttributeFilter, then processes the request. Since the user is an Admin, there is no need to check who owns the resource.

Unit Testing

The original catalyst for this change came when I was trying to unit test the controllers after implementing authentication and authorization; The current method of determining if the request is coming from an admin or the owner of the resource added a layer of complexity that became too difficult to (reasonably) test.

The hope is that, by splitting the responsibility apart, I won't need to utilize an in-method filter and can more easily test the actual logic of the controllers, instead of setting up complicated tests that pre-fill the HttpContext with specific data to test the authorization methods.

May 1, 2020

Friday, May 1, 2020 1:19 PM

I have nearly completed the re-write of the Controllers and have started looking at how to protect these endpoints and resources. There seem to be two main ways of protecting endpoints and resources: One is implementing an IAuthorizationService and using it within the endpoint methods themselves, the other is to create authorization policies and then decorate the endpoints with an action filter that utilizes one or more of these policies.

IAuthorizationService

A basic overview of how to implement an IAuthorizationService to handle resource-based protection is available on Microsoft's website: https://docs.microsoft.com/en-us/aspnet/core/security/authorization/resourcebased?view=aspnetcore-3.1

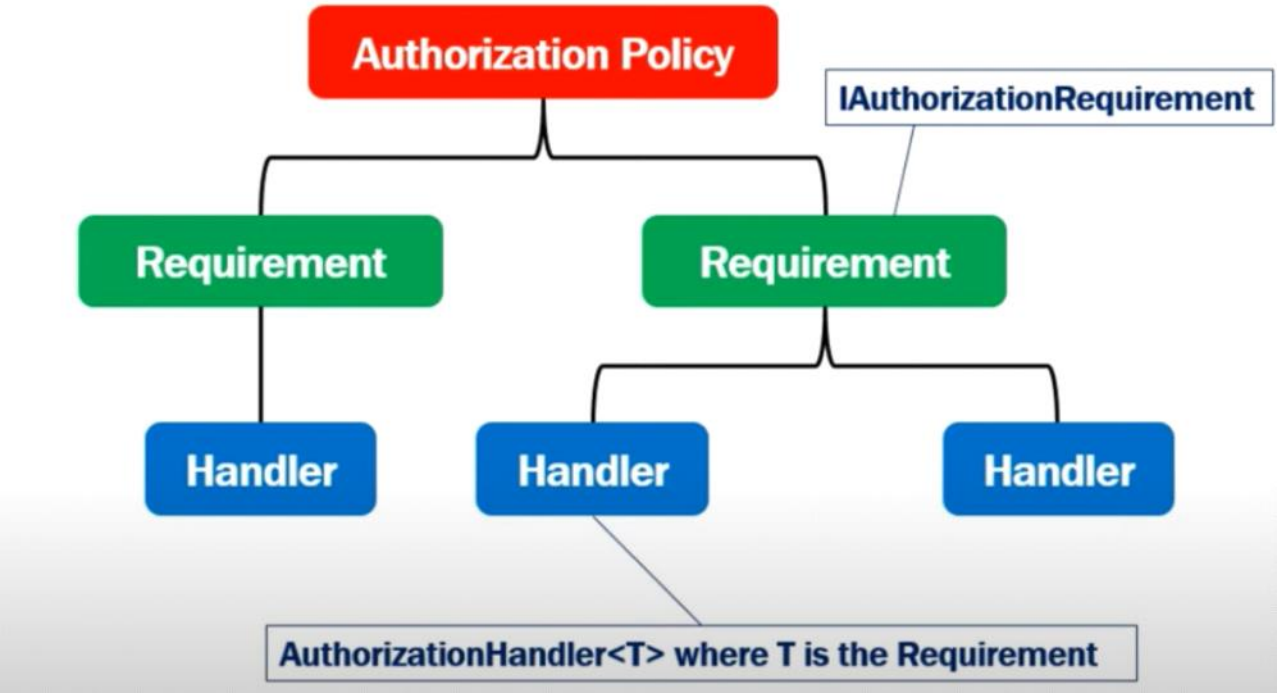
Authorization Filters

A good video tutorial on how to make authorization handlers that can be used as an action filter is available on YouTube at: Custom authorization requirement and handler example in asp net core

Structure

The basic structure of this authorization handler is captured in the screenshot below:

Custom authorization requirement



We must implement the IAuthorizationRequirement interface for each requirement; For example, accessing the AdminController endpoints would have an AdministrativeRightsRequirement.

```
protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
    ManageAdminRolesAndClaimsRequirement requirement)
{
    var authFilterContext = context.Resource as AuthorizationFilterContext;
    if (authFilterContext == null)
    {
        return Task.CompletedTask;
    }

    string loggedInAdminId =
        context.User.Claims.FirstOrDefault(c => c.Type == ClaimTypes.NameIdentifier).Value;

    string adminIdBeingEdited = authFilterContext.HttpContext.Request.Query["userId"];

    if (context.User.IsInRole("Admin") &&
    context.User.HasClaim(claim => claim.Type == "Edit Role" && claim.Value == "true") &&
    adminIdBeingEdited.ToLower() != loggedInAdminId.ToLower())
    {
        context.Succeed(requirement);
    }

    return Task.CompletedTask;
```

In this example, we have access to the Context in which the AuthorizationFilter is acting, giving us access to the HttpContext of the controller. From this resource, we can access the current User, as well as parameters passed in the query string. From these, we can check whether the current user is an Administrator and if they are the same user as the one being access in the query string (which contains

the target user's ID).

Red Giant

I can use this example to help develop my own Authorization Filters for the Red Giant application. First, I will need a filter to only allow access to users with Administrative Rights (an AdministrativeRightsRequirement, implementing the IAuthorizationRequirement interface). From here, I can develop a handler (CanAccessAdministrativeFunctionsHandler, inheriting from AuthorizationHandler<T> where T is the AdministrativeRightsRequirement).

In the logic of this filter, I can verify that the current user is part of the "Admin" role.

A second filter will be to ensure that the current user is the owner of the resource being accessed. In this scenario, I would have a ResourceOwnerRequirement implementing the IAuthorizationRequirement interface and a IsResourceOwnerHandler inheriting from AuthorizationHandler<T> where T is the ResourceOwnerRequirement.

May 2, 2020

Saturday, May 2, 2020 9:44 AM

I was able to implement the authorization filters without too much hassle, completing it much quicker than I expected. With the authorization in place, I began testing the API last night and discovered that I hadn't set up the routing correctly - I was using a separate attribute on each endpoint for the routing URI.

For example, the LogsControllerV2 Update method was using this combination of attributes:

```
[HttpPut, Route(Contracts.Routes.Identity.V2.Update)]
```

During my testing, I was receiving 404 Not Found results for these endpoints and couldn't figure out why. I had set up the routing following RESTful best practices, as such:

- Create - POST /api/logs
- Update - PUT /api/logs/{id}
- Delete - DELETE /api/logs/{id}
- GetById - GET /api/logs/{id}

I wasn't able to access the Update or Delete endpoints, but could access the Get and Create endpoints. There wasn't a helpful error message, but it seemed that the API couldn't decide which endpoint I was trying to access, even with the HTTP method present.

I tried changing these to use unique descriptors, like:

- Create - POST /api/logs
- Update - PUT /api/logs/{id}/update
- Delete - DELETE /api/logs/{id}/delete
- Get - GET /api/logs/{id}

This set up worked, but then I read the best practices again and it specifically recommended **not** using CRUD verbs in the URI (see https://restfulapi.net/resource-naming/).

After going back to the original URI design, I finally came across this website which gave examples of how to set up the access-based URI routes: https://stormpath.com/blog/routing-in-asp-net-core

In example number 4, about building RESTful routes, they no longer used the "Route" attribute and, instead, included any routing information as part of the HTTP method attribute:

4. Building RESTful Routes

In order to declare a RESTful controller, we need to use the following route configuration:

```
[Route("api/[controller]")]
public class ValuesController : Controller
{
    // GET api/values
    [HttpGet]
    public IEnumerable<string> Get()
    {
        return new string[] {"hello", "world!"};
    }

    // POST api/values
    [HttpPost]
    public void PostCreate([FromBody] string value)
    {
    }
}
```

Here we are telling to our RESTful service to accept calls under the `/api/values` route. Note that we no longer use the `Route` attribute for actions. Instead we decorate it with `HttpGet`, `HttpPost`, `HttpPut`, `HttpDelete` attributes.

Or, we can take a look at a different scenario:

```
// POST api/values/5
[HttpPost("{id}")]
public void PostUpdate(int id, [FromBody] string value)
{
}
```

Here we have the following routes for the controller Values

- `HTTP Post` of `/values` route will invoke `Post()` action
- `HTTP Post` of `/values/PostName` route will invoke `Post([FromBody]string value)` action

This made me realize that the routing information in the HTTP method attribute would take into account the HTTP method of the request, while the route included in the "Route" attribute would not. I then updated the routing on my LogsController to only use the HTTP method attribute and it was able to discern which endpoint I was requesting based on the HTTP Method!

```
/// <returns>OK, BadRequest, or Conflict</returns>
[Authorize(Policy = Contracts.Poli
[HttpPut(Contracts.Routes.Logs.V2.Update)]
```

```
string Routes.Logs.V2.Update = "{id}"
```

Now I can use the correct, RESTful URIs for the routes and still be able to access the correct endpoints by using the HTTP method.

Request Models

Since the RESTful URIs require the resource's ID as part of the URI string, I will no longer need to include the ID as part of the request model. For example, the LogUpdateRequest model includes the LogId as part of the model properties, but this ID is already required as part of the URI:

UPDATE /api/logs/{id}

```
LogUpdateRequest
{
```

```
    string LogId { get; set; }
    string OwnerId { get; set; }
    string Title { get; set; }
    string Comments { get; set; }
}
```

This is a similar pattern in other request models, so I will need to ensure that I remove it to better follow the DRY principle.

I will also need to spend some time incorporating these kinds of Request and Response models into all the API endpoints, instead of sending the actual resource object back through the API (like the IdentityUser data should be stored into a UserResponse-type object). I will add this to the backlog and hopefully have it finished by the end of this sprint.

May 3, 2020

Sunday, May 3, 2020 3:07 PM

It's the last day of Sprint 2 and I feel like I am complete with the backend API; I have been working with the features through Postman and ensuring that everything works as expected and all the unit tests are passing. The scope of the controllers certainly grew as I developed them, but I made sure to follow the TDD paradigm when adding any new capabilities.

I only have a few more minor changes to make to the LogsService so that it operates similarly to the IdentityService and then I will conduct my sprint review and see how much I actually accomplished. If I have finished everything that is needed for the API, I will be able to focus the next sprint or two on the frontend application, utilizing the TestAPI project and the RedGiantRLM project as examples.

I would like to leave the last sprint or two to write up the project report so that I'm not rushed.

Sprint Review

Sunday, May 3, 2020 3:10 PM

I have completed nearly 100 work points this sprint and I feel like I more accurately allocated points than the previous sprint. I even created a new 0.5 point for some of the easier tasks.

I am satisfied with the progress I made over the last two weeks and have (for the most part) completed the backend API service. I have three controllers:

- LogsController - Manages basic user interaction with log data
- IdentityController - Allows basic user interaction with authentication and user data
- AdminController - Allows Admin users full access and control over all log and user data

I was able to implement fairly advanced authorization techniques and followed the TDD paradigm, which greatly influenced the overall design of the application. There were many instances where a simple solution within a class was prohibited by the complexity it would have created within its tests; This would often force me to research a better solution that would provide the functionality needed while allowing me to fully test everything.

Some of the biggest challenges I faced during this sprint had to do with authentication and authorization. In some of my previous projects, I had implemented authorization methods that *worked*, but were certainly not the best design. For this application, since I was focused on TDD and testability of the methods of all classes, I was forced to really think about the design and how to make everything testable. This often resulted in utilizing interfaces and DotNet Core's built-in DI engine, which was actually fun and satisfying to use, once I learned how and when it could be utilized.

I am quite proud of how much I have been able to accomplish since starting this project and how well I have followed some of the best practices and software development strategies. I'm sure there are areas I could improve in and it may be worth re-examining the API codebase with a more SOLID-focused lense, but I am in no way embarassed or ashamed of what I have built.

I know there will likely be changes to the backend once I start developing the front-end, but I tried to think through how the front-end would be used while developing the RESTful API. There are no endpoints that align perfectly with the front-end's needs, like an endpoint supplying an object specifically designed to carry everything the front-end page would need. This was something I often resorted to with previous projects and even the first version of this app (which was built with Node.js and was very easy to send custom objects).

This API follows RESTful practices as closely as possible so that the front-end application can get all the data it needs while also utilizing the RESTful design paradigm.

Project Submission

Wednesday, June 10, 2020 10:47 PM

I would like to acknowledge that I fell behind in my sprint notes and journaling. I started a new job as a full-time Software Engineer on May 4th and entirely stopped journaling. I have updated the design notes with the final UML designs, representing the current and final state of the application.