




CA400

Technical Guide



Project Title: Synche

Student 1 Name (ID): Tara Collins (15416228)
Student 2 Name (ID): Theo Coyne-Morgan (17338811)
Project Supervisor: Dr. Stephen Blott
Date Submitted: 7th May 2021



0. Table of Contents

0. Table of Contents	2
1. Introduction	5
1.1 Project Overview	5
1.2 Glossary	6
1.3 Motivation	7
2. System Architecture	8
2.1 Languages and Tools	8
Go	8
GNU Make	8
Docker	8
React	8
Node.js	8
Tailwind	8
2.2 Direct Dependencies	9
sqlmock	9
jwt-go	9
structs	9
fsnotify	9
Go-MySQL-Driver	10
imohash	10
go-homedir	10
go-cache	10
logrus	10
afero	11
cobra	11
pflag	11
viper	11
testify	11
go-swagger	12
gorm	12



2.3 System Architecture Diagram	13
3. High-Level Design	15
3.1 Context Diagram	15
3.2 User Data Flow Diagram	16
3.3 Server Data Flow Diagram	17
3.4 Client GUI	18
4. Implementation	22
4.1 Authentication	22
4.2 Configuration	22
4.3 Caching	24
4.4 Database	25
4.5 User Account Creation	25
4.6 User Account Deletion	26
4.7 Upload File	26
4.7.1 File Splitter	26
4.7.2 File Uploader	27
4.7.3 Handling Failed Uploads	27
4.7.4 Queueing Uploads	29
4.8 Reassemble File	29
4.9 Delete File	29
4.10 Move File	29
4.11 Download File	31
4.12 Retrieve File Information	32
4.13 Create Directory	32
4.14 Delete Directory	33
4.15 List Directory Contents	34
4.16 Move Directory	35
4.17 Docker	36
5. Development Workflow	37
5.1 Clubhouse Kanban Board	37
New	37
Icebox	37
Ready for Development	37
Priority	37



In Development	38
Completed	38
5.2 Git	40
5.3 Meetings	41
6. Testing	42
6.1 Unit Testing	42
6.2 Integration and Regression Testing	43
6.3 User Testing	44
7. Research, Challenges, and Resolutions	46
7.1 GoLang	46
7.2 Caching	46
7.3 Concurrency	47
7.4 Database Management	47
7.5 Time Management	48
7.6 Covid-19	48
8. Future Work	49
8.1 Data Deduplication	49
8.2 Metrics and Statistical Dashboard	49



1. Introduction

1.1 Project Overview

Synche is a file management system that allows for multiple concurrent users and features a multipart file transfer tool. It can be considered a faster and more reliable alternative to cloud storage solutions such as Google Drive or Dropbox. The primary objective of Synche is to provide a faster and more reliable method of transferring large files than traditional file transfer protocols. Synche also grants the user the ability to have full control of their data and file privacy by utilising the storage space that they already own or rent and eliminating the need for third-party storage software.

Currently, the only multipart file uploading tools available (such as AWS S3) are expensive, proprietary, and cater to large businesses rather than the average user or small company. Renting a server along with using Synche is significantly cheaper than paying to use a storage solution such as AWS S3. The bridge that allows the average user or small businesses to access fast multipart file transfer, data loss mitigation, and private self-owned storage does not yet exist. Synche aims to become that bridge.

The faster upload speeds are facilitated by the multipart file uploading tool. This tool increases upload speeds by splitting files into “chunks” and concurrently uploading these individual file chunks to the server. Using multiple connections that concurrently upload file chunks is monumentally faster than using a single connection to upload a large file. This file upload method also caters for better upload failure handling and mitigates data loss from connection failure. In the event of a connection failure midway through an upload, chunks that have successfully been uploaded remain on the server and the client only needs to resend the chunks that have failed to upload.

Synche’s services can be accessed through the CLI or through the clean user-friendly GUI for users that are not comfortable using the command line.



1.2 Glossary

API	An application programming interface (API) is a set of functions and procedures allowing the creation of applications that access the features or data of an operating system, application, or other service.
AWS	Amazon Web Services (AWS) is a subsidiary of Amazon that provides cloud computing platforms and APIs, on a metered pay-as-you-go basis.
Chunk	A segment of a file that has been split into small parts.
CLI	A command line interface (CLI) is a terminal program that handles input from a user via text based commands.
GUI	A graphical user interface (GUI) is a form of user interface that allows users to interact with an underlying computer system via visual indicator representations.
OS	An operating system (OS) is the software that manages and communicates with a computer's hardware
S3	Simple Storage Service (S3) is a service offered by Amazon Web Services that provides object storage through a web service interface.
Metadata	A set of data that describes and gives information about other data. In relation to this project, 'metadata' will be used to describe the information pertaining to a file such as its on-disk size, location, hash, modification date, upload date and permissions.



1.3 Motivation

Both developers have a personal interest in digital media, film and photography in particular. The quality of digital media is ever-increasing. Quality increases result in data sizes becoming larger. With this, the cost and difficulty of managing media storage has become complicated.

We both have experience with using Google Drive and Dropbox as a means of storage. While the services they provide are adequate, the recurring monthly subscription fees are incredibly off-putting for students with low income that mainly need the storage space for personal projects and hobbies. We both independently began researching alternative storage solutions and the costs of renting private servers.

We realised that self-owned storage servers were a solution to part of our problem. In the long run, buying or renting our own servers is more financially viable. We found cheap server space that we could pay for on a monthly or yearly basis. This gave us complete control over where we could store our data, and we could share the storage space in order to reduce costs. We could even repurpose old laptops or computers and create storage servers within our own household, on our own networks. This is cleaner and safer than using external hard drives, and cheaper than paying for file hosting services.

We solved *where* we could store our data, but we had not yet figured out *how*. Media files are more often than not, very large. They cannot be compressed any further than they already are. This makes uploading large files in whole, slow. We researched methods of transferring files and discovered that implementing our own version of multipart file transfer would greatly increase the speed of upload files. This solved *how* we could easily transport large volumes of data.

We figured out a solution to our problem, one that many people have, and we wanted that solution to become accessible to the average person or small business.



2. System Architecture

2.1 Languages and Tools

- **Go**
 - **Version:** v1.16.2
 - **Usage:** Synche client, Synche server, Synche SDK
- **GNU Make**
 - **Version:** v4.1
 - **Usage:** Building the Synche client and server
- **Docker**
 - **Version:** v19.03.14
 - **Usage:** Synche server
- **React**
 - **Version:** v17.0.2
 - **Usage:** Synche GUI
- **Node.js**
 - **Version:** v16.1.0
 - **Usage:** Synche GUI
- **Tailwind**
 - **Version:** v2.1.2
 - **Usage:** Synche GUI



2.2 Direct Dependencies

These are the direct dependencies of our Go application only. Some of these libraries may have other dependencies that are not listed below. Go Open API libraries have not been included.

- **sqlmock**
 - **Version:** v1.5.0
 - **Description:** a mock library implementing SQL/driver. Its purpose is to simulate SQL driver behaviour in tests, without needing a real database connection.
 - **Source Code:** <https://github.com/DATA-DOG/go-sqlmock>
 - **Usage:** Unit and integration testing.
- **jwt-go**
 - **Version:** v3.2.0
 - **Description:** A library that supports the parsing and verification, generation, and signing of JWTs.
 - **Source Code:** <https://github.com/dgrijalva/jwt-go>
 - **Usage:** Server-side authentication.
- **structs**
 - **Version:** v1.1.0
 - **Description:** A library that contains various utilities to work with Go structs.
 - **Source Code:** <https://github.com/fatih/structs>
 - **Usage:** Allows the user to input initialisation configuration.
- **fsnotify**
 - **Version:** v1.4.9
 - **Description:** A library that utilizes golang.org/x/sys rather than `syscall` from the standard library.



- **Source Code:** <https://github.com/fsnotify/fsnotify>
- **Usage:** Allows the user to input initialisation configuration.

- **Go-MySQL-Driver**
 - **Version:** v1.6.0
 - **Description:** A MySQL-Driver for Go's database/sql package
 - **Source Code:** <https://github.com/go-sql-driver/mysql>
 - **Usage:** Database management.

- **imohash**
 - **Version:** v1.0.0
 - **Description:** A library for fast, constant-time hashing.
 - **Source Code:** <https://github.com/kalafut/imohash>
 - **Usage:** Validating file integrity.

- **go-homedir**
 - **Version:** v1.0.0
 - **Description:** A library for detecting the user's home directory so that the library can be used in cross-compilation environments.
 - **Source Code:** <https://github.com/mitchellh/go-homedir>
 - **Usage:** Configuration.

- **go-cache**
 - **Version:** v2.1.0
 - **Description:** A library for an in-memory key:value store/cache similar to memcached that is suitable for applications running on a single machine.
 - **Source Code:** <https://github.com/patrickmn/go-cache>
 - **Usage:** Caching.

- **logrus**
 - **Version:** v1.8.1
 - **Description:** A structured logger that is compatible with the standard library logger.



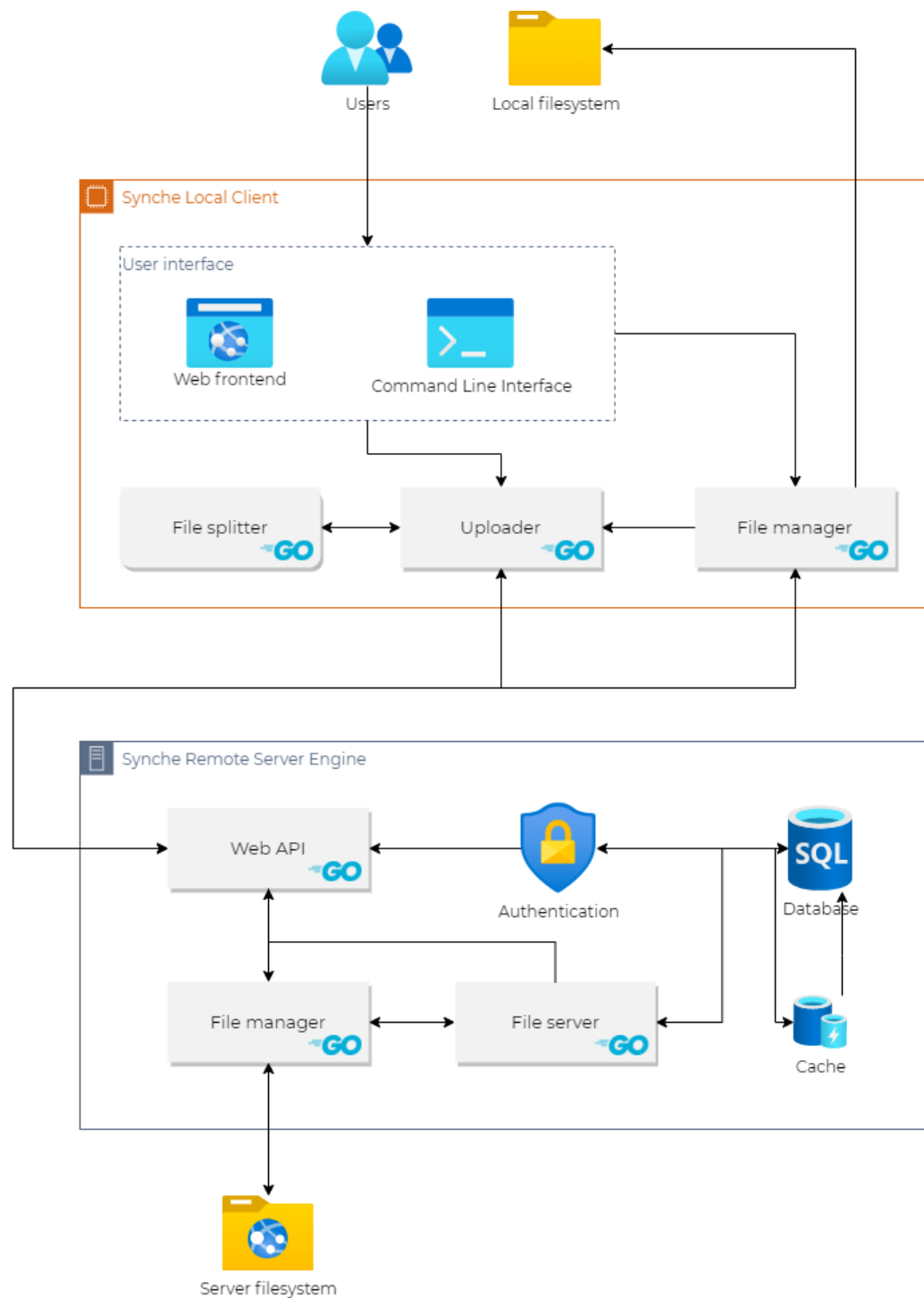
- **Source Code:** <https://github.com/sirupsen/logrus>
- **Usage:** Clear and legible logging.
- **afero**
 - **Version:** v1.6.0
 - **Description:** A filesystem framework providing a simple, uniform and universal API interacting with any filesystem.
 - **Source Code:** <https://github.com/spf13/afero>
 - **Usage:** File management.
- **cobra**
 - **Version:** v1.1.3
 - **Description:** A library for creating powerful modern CLI applications.
 - **Source Code:** <https://github.com/spf13/cobra>
 - **Usage:** CLI.
- **pflag**
 - **Version:** v1.5.0
 - **Description:** A flag package for implementing POSIX/GNU-style --flags.
 - **Source Code:** <https://github.com/spf13/pflag>
 - **Usage:** CLI and binding server ports.
- **viper**
 - **Version:** v1.7.1
 - **Description:** A configuration solution designed to handle all types of configuration formats.
 - **Source Code:** <https://github.com/spf13/viper>
 - **Usage:** Configuration.
- **testify**
 - **Version:** v1.7.0
 - **Description:** A library that provides testing packages.
 - **Source Code:** <https://github.com/stretchr/testify>



- **Usage:** Testing.
- **go-swagger**
 - **Version:** v0.27.0
 - **Description:** A tooling ecosystem for developing APIs with the OpenAPI Specification (OAS).
 - **Source Code:** <https://github.com/go-swagger/go-swagger>
 - **Usage:** Validating user passwords.
- **gorm**
 - **Version:** v1.21.8
 - **Description:** An ORM library.
 - **Source Code:** <https://gorm.io/>
 - **Usage:** Database management.



2.3 System Architecture Diagram

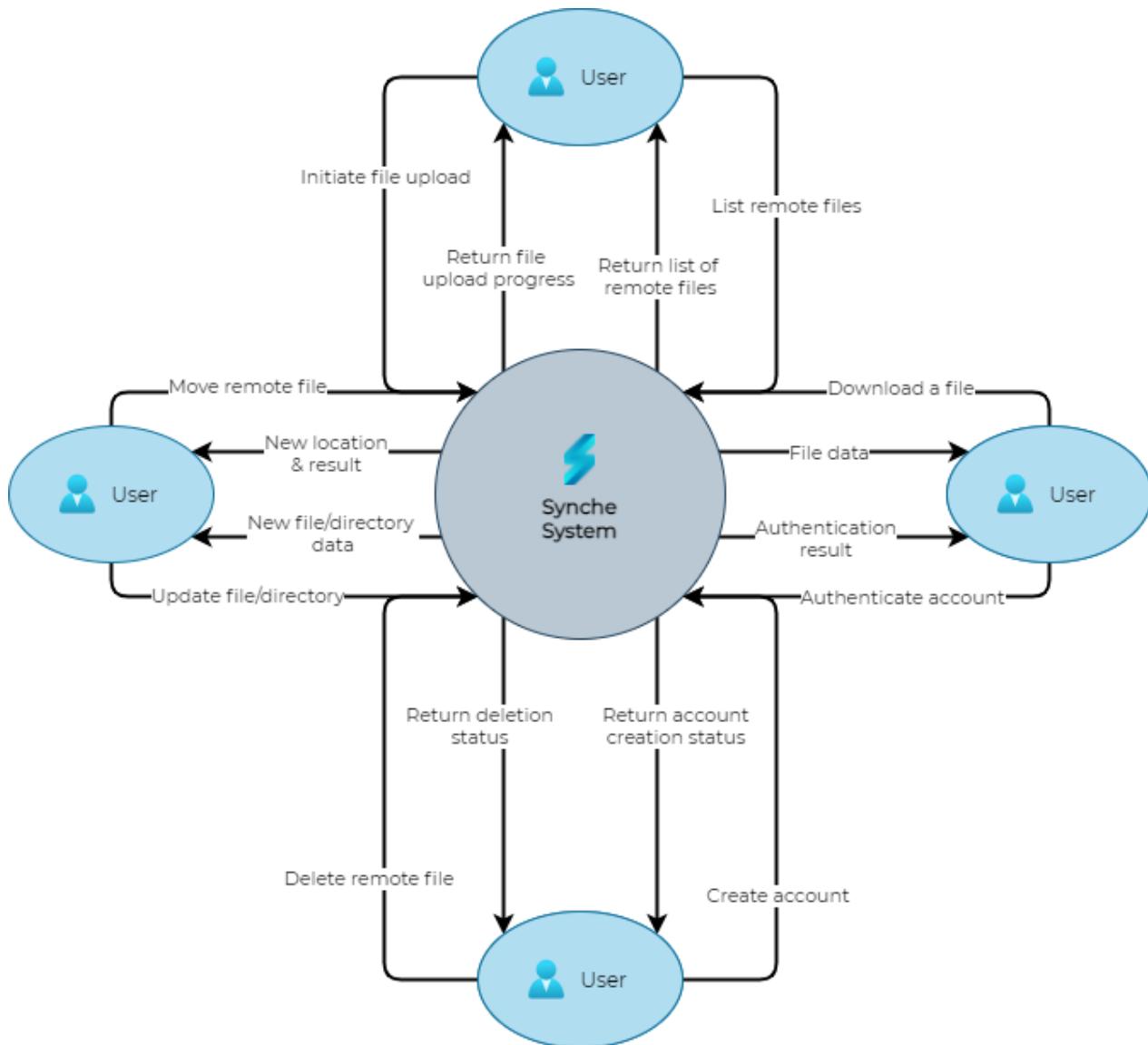


The figure above illustrates a simplified overview of the Synche system. Synche consists of two main components; a user client and a server storage system. The client's core purpose is to provide an interface for the end-user which will communicate with the server API, upload, list and delete files on the remote server. The client can be accessed through either the CLI or the Synche GUI. The server can run locally through the GUI or within a Docker container.



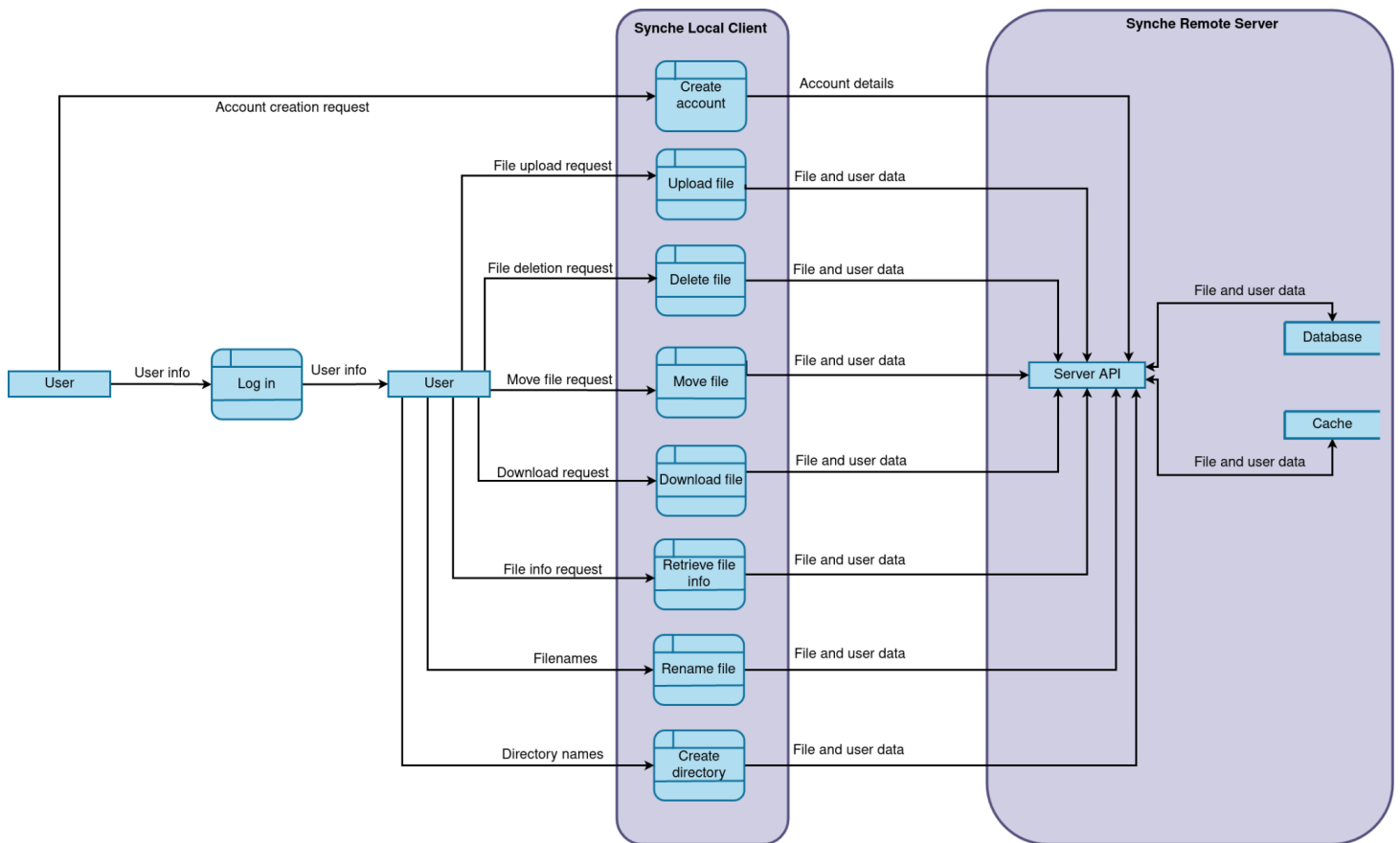
3. High-Level Design

3.1 Context Diagram



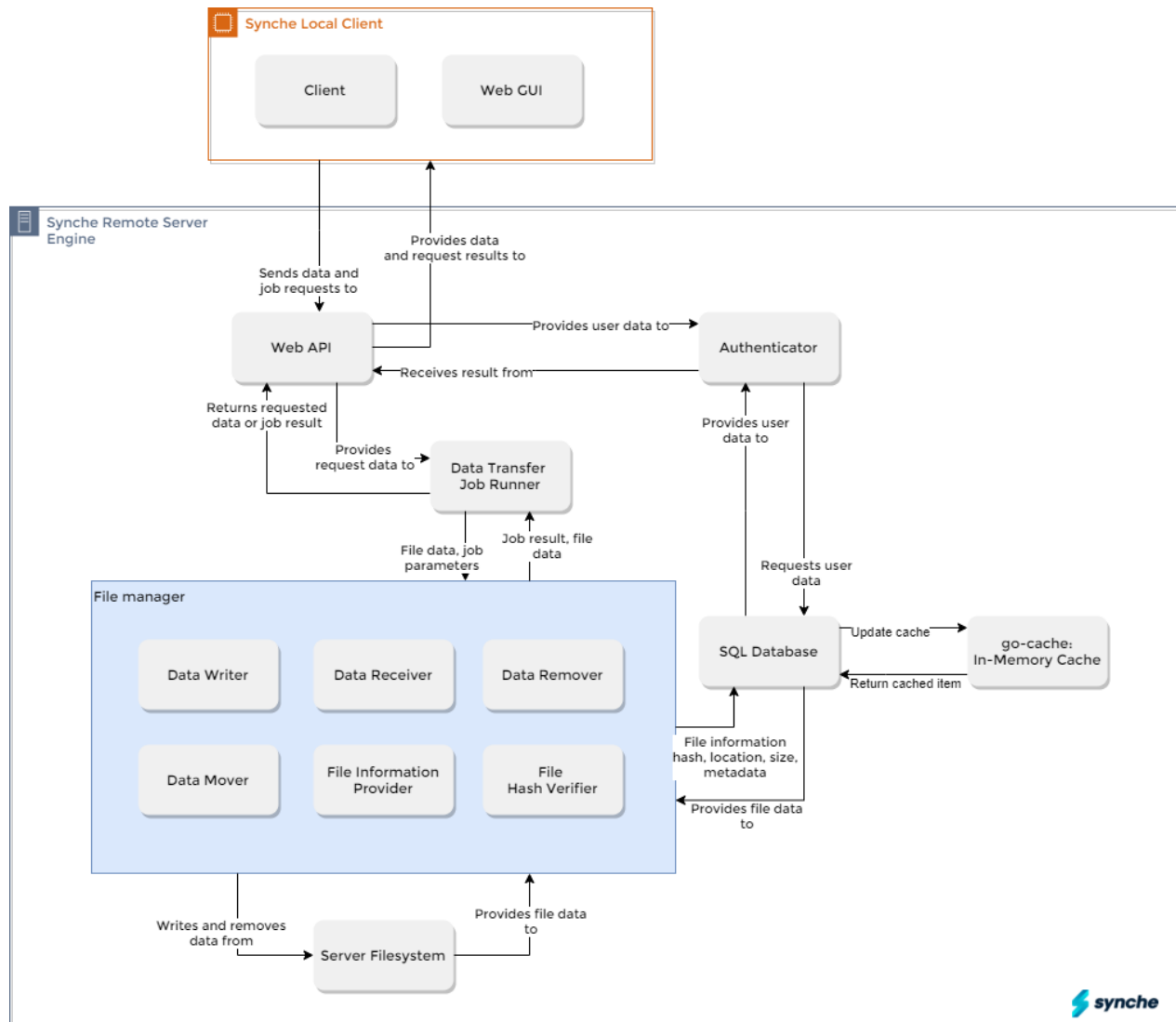
The diagram above illustrates the actions that users can perform with the Synche system.

3.2 User Data Flow Diagram

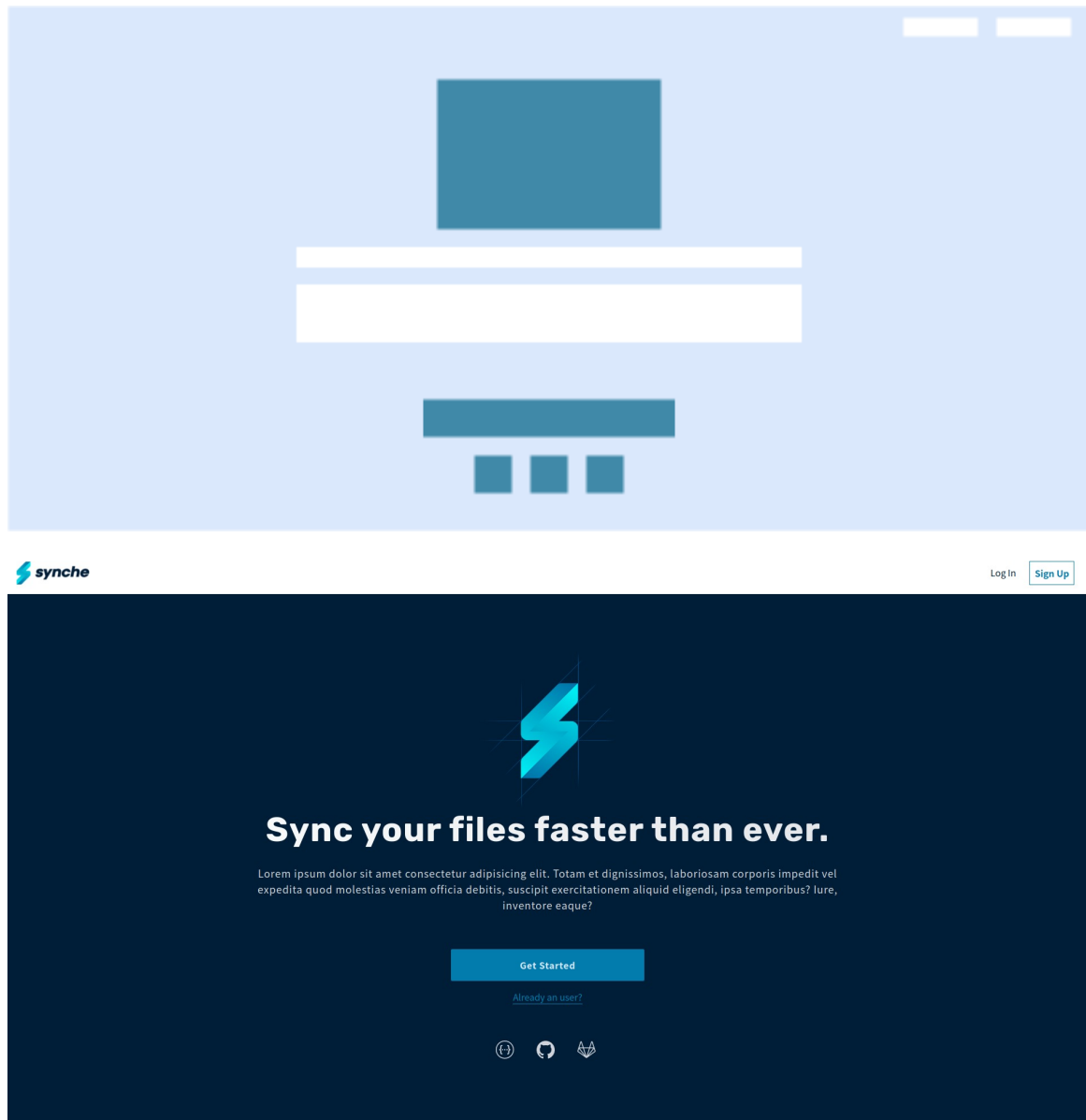


The diagram above illustrates the transfer of data from a user's perspective. Users have one primary connection to the Synche storage system through the local client. Once the user has created an account and logged in, they have access to Synche's services. The user is sending primarily file data to the server. The server API caches these requests and manages the user's data through utilising both a database and a cache.

3.3 Server Data Flow Diagram

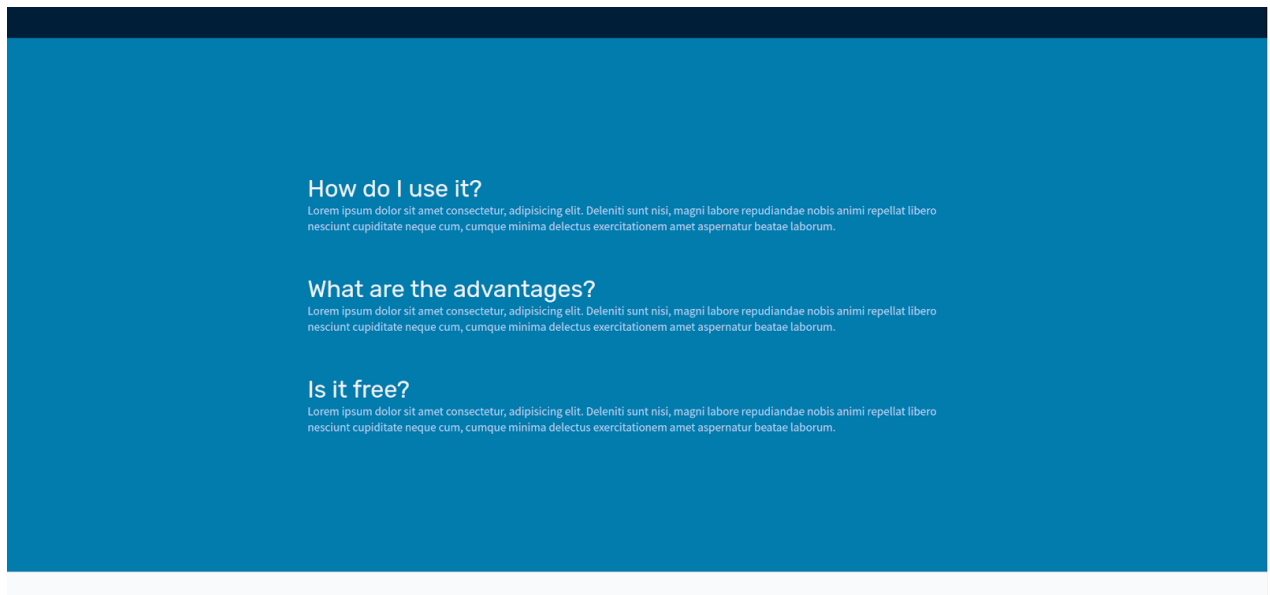


3.4 Client GUI



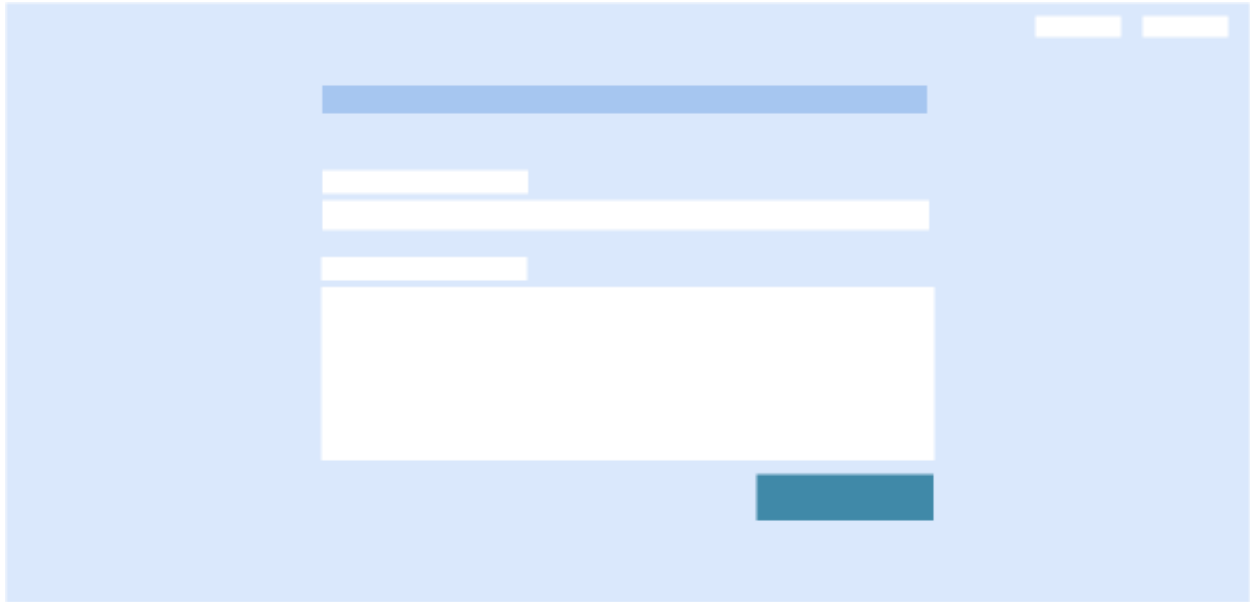
The figures above illustrate the home page that the user sees. It features a central logo at the top with some basic information regarding the Synche project underneath it. At the bottom, the user may get started with creating their account or click the links to visit Synche social media sites where they can find Synche updates.





The figures above illustrate the simple info page that answers some questions that we were frequently asked when developing Synche.






Questions? We got the answers!

Email *

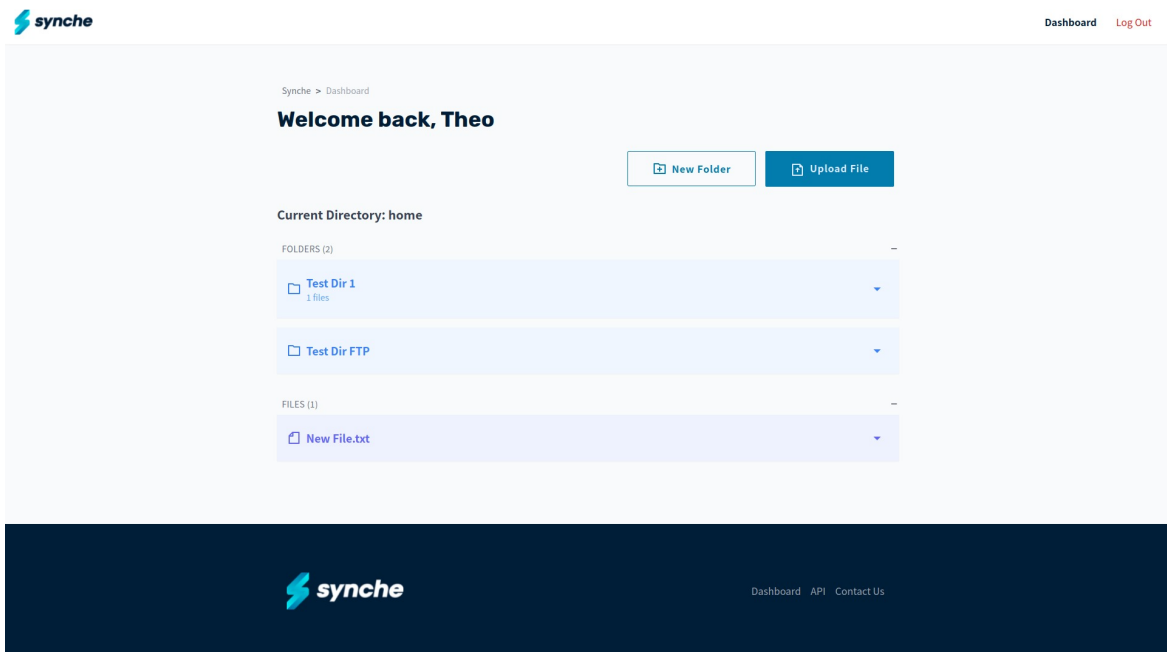
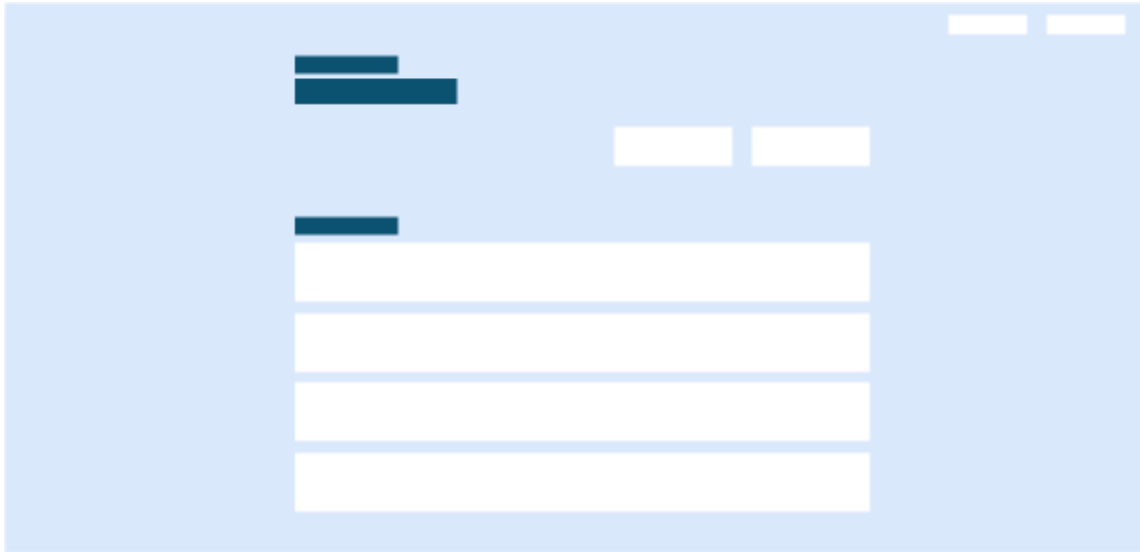
Message *

Send

[Dashboard](#) [API](#) [Contact Us](#)

The figures above illustrate the contact page so that the user may directly contact the developers with regard to any questions that may have. This includes tech support and suggestions for future versions of the application.





The figures above show the user's home page once they are logged in Synche. They are greeted based on the username that they provided. The buttons to upload a file or create a new folder are clearly visible. Below that, the user's saved files and the folder that the user is currently in are displayed. Users may view each file's details and request to delete them on this page. The user may also navigate to another directory on this page and where they may also perform file management operations.



4. Implementation

4.1 Authentication

User authentication is achieved using JWT (JSON Web Tokens). JSON Web Tokens are an open industry standard method for representing claims securely between two parties. JSON Web Tokens can be generated from user data and can be decoded to authenticate users on the system. The Synche server handles generating and storing the relevant data for these tokens in the database. Once a client authenticates a user using the login endpoint, the user credentials such as the access token and refresh token are stored in a file on the disk. This allows the user to maintain an authenticated state, without having to re-login every time a user performs an action using the client. By storing the refresh token a user can continually refresh an access token to provide access to the server. This is a secure way to persist user access data without actually storing the username and password.

4.2 Configuration

Both the server and client are fully configurable through the use of simple configuration files. This is implemented through using the Viper library. Viper facilitates different configuration file types by providing un-marshalling functionality to allow a consistent configuration structure. The configuration file structure is defined for the server and client in their respective folders. If either the server or the client configuration files are not present upon execution, Synche will prompt the user to input the respective configuration values through the CLI.

The user can edit these configuration files at any time and relaunch the server or client. Configuration files are expected to be in `/home/user/synche` by default.



Below is a snippet from `/src/config/read.go` that illustrates how this is implemented.

```
// ReadOrCreate Reads in a config if it exists, if it does not exist it will call  
// Setup() to prompt the user to set up their configuration
```

```
func ReadOrCreate(name, path string, defaultCfg, configStruct interface{})  
(created bool, err error) {  
    err = Read(name, path)  
    if _, ok := err.(viper.ConfigFileNotFoundError); ok {  
        log.Warn("No config file found")  
  
        newConfig, err := Setup(defaultCfg)  
        if err != nil {  
            return false, err  
        }  
  
        if path == "" {  
            path = filepath.Join(SynchedDir, name+".yaml")  
        }  
  
        viper.Set("config", newConfig)  
        if err := viper.UnmarshalKey("config", &configStruct); err != nil {  
            return false, err  
        }  
  
        if err = Write(path, newConfig); err != nil {  
            return false, err  
        }  
  
        return true, nil  
    }  
  
    return false, err  
}
```

4.3 Caching

A critical element of Synche is the speed and performance in which uploads can be completed. Caching all frequently accessed information is an effective way of increasing performance speeds. The most important things to cache are user tokens and chunk data because this information is accessed regularly.

Caching the chunk data allows Synche to facilitate fast uploads. The cache stores how many chunks have been received for each file that's being uploaded. Once the number of chunks received reflects how many chunks are expected for a specific file, the cache triggers the file re-assembler. Any data relating to that file will then be removed from the cache. To ensure that no redundant data is left sitting in the cache, each cache is created with a default expiry time of 5 minutes and a purge time of 10 minutes.

Below is a snippet from `/src/server/handlers/upload_chunk.go` that illustrates how the number of chunks is added to the cache. If a file has not yet been added to the cache, an entry is initialised. The cache relating to a given file is then incremented in order to track how many chunks have been received.

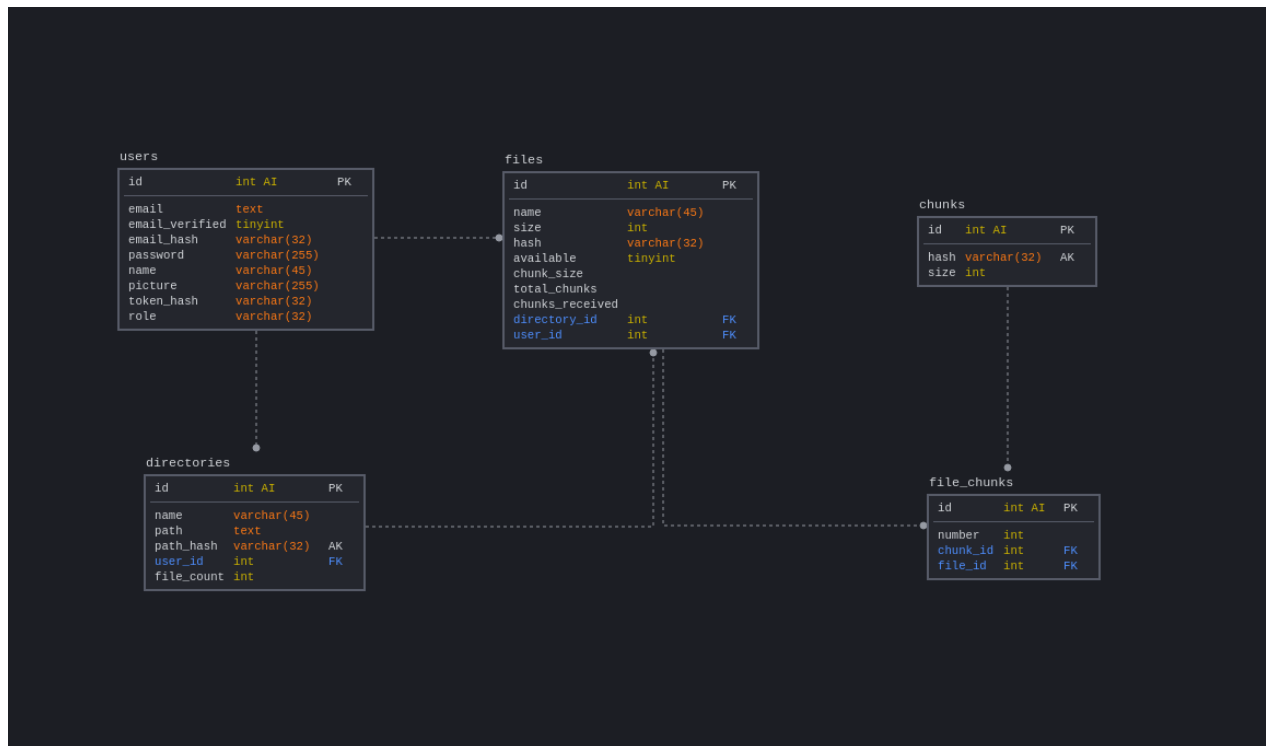
```
strFileID := strconv.Itoa(int(file.ID))
var chunksReceived int64
if item, ok := repo.FileIDChunkCountCache.Get(strFileID); ok {
    chunksReceived, ok = item.(int64)
    if !ok {
        log.Error("invalid cache entry for chunks received")
    }
} else {
    repo.FileIDChunkCountCache.Set(strFileID, int64(0), cache.DefaultExpiration)
}

err := repo.FileIDChunkCountCache.Increment(strFileID, 1)
if err != nil {
    return serverErr.WithPayload("failed increment the chunks received count")
}
```



4.4 Database

The schema diagram below illustrates the format of our database and how it has been normalised to 3NF. It highlights that there are no transitive functional dependencies.



4.5 User Account Creation

When a user requests to create a new account, they are prompted to input the new account details. This includes their email address, name, and password. The password will be screened to ensure that it is a strong password. If it is not, the request will be rejected and the user will be prompted to input a valid password. This is to ensure the highest level of security.

These details will be sent to the server which will validate that a user does not already exist with the email address provided. Email addresses must be of a valid email address format. As mentioned in section 4.1, the server manages the generation of tokens.

4.6 User Account Deletion

When a user wishes to delete their account, they will be prompted to enter their email and password to confirm and authenticate the deletion. If authenticated, everything relating to the user will be deleted from the server. This includes user data in the database, user data in the cache, and any files or directories owned by the user. This means that the user does not need to worry about the possibility that any of their data remains on the server.

4.7 Upload File

When the user requests to upload a file, the client will first send a request to the server containing the file details. This includes the file name, hash, size, and the number of chunks that the file will be split into. The server will respond appropriately. If the file already exists on the server, the server will respond to the client so that it can inform the user and the user does not upload a duplicate file. If the server accepts the upload request, the client will begin the protocol for uploading a file.

4.7.1 File Splitter

File chunks are uploaded concurrently. Multiple concurrent connections allow for faster uploads speeds for large files. The file splitter facilitates this by splitting a file into smaller file chunks. It does this by reading in an amount of bytes specified in the client configuration file. This value can also be overwritten by the user for an individual file with the `-s` flag. This “chunk” is then sent to the asynchronous uploader.

The below snippet from `/src/client/upload/file.go` highlights how this is implemented.



```

err = splitter.Split(
    func(chunk *data.Chunk, index int64) error {
        log.WithFields(log.Fields{"chunk": *chunk, "index": index}).Info("")
        if index%int64(c.Config.Chunks.Workers) == 0 {
            log.Infof("%d - Waiting for %d workers...", index, c.Config.Chunks.Workers)
            wg.Wait()
        }
        params := NewChunkUploadParams(*chunk, upload.ID)
        wg.Add(1)
        go asyncChunkUploader(&wg, params, uploadErrors)
        return nil
    },
)

```

4.7.2 File Uploader

The uploader manages the concurrent uploads. Concurrency in Go is implemented through goroutines. A channel is created with a number of workers that is taken from the client configuration file. If the user wishes to override that number for a specific upload, they can do so by using the `-w` flag. A WaitGroup is created to manage the workers. A WaitGroup waits for a collection of goroutines to finish. The main goroutine calls Add to set the number of goroutines to wait for. Then each of the goroutines runs and calls Done when finished. At the same time, Wait can be used to block until all goroutines have finished. Wait is used to block the client from sending a finish upload request. The details of this are explained below.

4.7.3 Handling Failed Uploads

When the asynchronous uploader finishes sending all file chunks to the server. The client will send one final request to the server. This request indicates to the server that it should have received all chunks. The server will first check if the amount of chunks that it has received is the amount of chunks that it was told it would receive when the file upload was initialised.

If the number of chunks does not match, the server will check the chunk cache to gather the IDs of the chunks that it did not receive. It will respond to the client with these



chunks IDs. The client will resend the chunks IDs that it receives from the server and, again, send a final request to indicate that it is finished uploading all chunks.

This process repeats until either the server receives all the chunks or the chunk error timeout is hit.

The snippet below is taken from `/src/server/handlers/finish_upload.go`. It demonstrates how the requests to finish an upload are handled on the server side of the application. It highlights that it will simply respond with an empty array to the client if there are no chunks missing. This is so that the client knows it does not need to resend any chunks.

```
func FinishUpload(params transfer.FinishUploadParams, user *schema.User)
middleware.Responder {
    fileID := params.FileID
    chunksReceived := repo.GetCachedChunksReceived(fileID)
    expectedNumOfChunks, err := repo.GetTotalFileChunks(fileID)
    if err != nil || chunksReceived == 0 {
        return transfer.NewFinishUploadDefault(500).WithPayload("failed to access
cache")
    }

    // If the amount of chunks received != the expected, find what chunks are missing
    if chunksReceived != expectedNumOfChunks {
        missingChunks, err := getMissingChunks(fileID, int64(expectedNumOfChunks))
        if err != nil {
            return transfer.NewFinishUploadDefault(500).WithPayload("failed to access
database")
        }
        return
    }
    return transfer.NewFinishUploadOK().WithPayload(&models.MissingChunks{ChunkNumbers:
missingChunks})
}

// No missing chunks
return
transfer.NewFinishUploadOK().WithPayload(&models.MissingChunks{ChunkNumbers:
[]int64{}})
}
```

4.7.4 Queueing Uploads

The user inputs the path to each file that they want to upload when they call the upload command. These arguments are iterated through one by one. The client uploads each file one by one until there are no file paths left in the argument array. Each upload only begins once the previous upload has finished.

4.8 Reassemble File

Reassembling a file is simple and straight-forward. Once the cache triggers the reassembler, it retrieves the directory ID of the given file from the database. With this, it can begin reassembling the file. It creates a file with the given file name. If a file with that file name already exists in that location on the database, the reassembler will rename the file so that it has a unique file name. For example, if test.mp4 already exists on the server when the reassembler is triggered (but the file hash is different and therefore the contents differ), the reassembler will rename the file to test(1).mp4 and so forth.

Once the file is created, the reassembler appends the bytes from each chunk file. The order is given by the chunk number to ensure that files are reassembled correctly. Upon completion, it sets the file as “available” in the database. This “available” field ensures that only files that are fully reassembled are downloaded to a client.

4.9 Delete File

Deleting a file is very intuitive. The user specifies either a file path relating to a file they want to delete or a file ID of a file that they want to delete. If the file exists on the server, it will be deleted from the disk, and its information will be deleted from the database.



4.10 Move File

Files can be moved by specifying file paths or a file ID and directory ID. This allows users to specify IDs optionally caters for those that may frequently be accessing a particular file or directory and smooths their workflow. The move command can also be used to rename a file. This allows the user to have full control over the files that they upload.

Below is a snippet that illustrates the endpoint handler for updating files that is found at `/src/server/handlers/update_file.go`

```
func UpdateFileByPath(params files.UpdateFileByPathParams, user *schema.User)
middleware.Responder {
    var (
        file      *schema.File
        newFile    *schema.File
        err404     = files.NewUpdateFileByPathDefault(404)
        err500     = files.NewUpdateFileByPathDefault(500)
    )

    fullPath, err := repo.BuildFullPath(params.FilePath, user, database.DB)
    if err != nil {
        return err500.WithPayload(models.Error(err.Error()))
    }

    file, err = repo.FindFileByFullPath(fullPath, database.DB)
    if err != nil {
        return err404.WithPayload("file not found")
    }

    if file.UserID != user.ID {
        return files.NewUpdateFileByPathUnauthorized()
    }

    newFile, err = updateFile(file, user, params.FileUpdate, database.DB)
    if err != nil {
        return err500.WithPayload(models.Error("failed to update the file: " +
err.Error()))
    }
}
```



```
    return files.NewUpdateFileByPathOK().WithPayload(ConvertToFileModel(newFile))
}
```

4.11 Download File

The user can request to download a file specified by a file path or a file ID. This is implemented using a named reader to essentially upload the file to the client. If the file does not exist on the server, it will respond to the client appropriately. Below shows the delete file handler found in `/src/server/handlers/download.go`

```
// DownloadFile Responds to the client with the file specified in the client's request
func DownloadFile(params transfer.DownloadFileParams, user *schema.User)
middleware.Responder {
    var file schema.File
    tx := database.DB.Joins("Directory").First(&file, params.FileID)

    if tx.Error != nil {
        return transfer.NewDownloadFileNotFound()
    }

    if file.UserID != user.ID {
        return transfer.NewDownloadFileForbidden()
    }

    filePath := filepath.Join(file.Directory.Path, file.Name)
    fileReader, err := files.Afs.Open(filePath)
    log.Debugf("Reading file: %s", filePath)
    if err != nil {
        return transfer.NewDownloadFileDefault(500).WithPayload("failed to read the
file")
    }

    namedReader := runtime.NamedReader(file.Name, fileReader)

    stat, err := fileReader.Stat()
    if err != nil {
        return transfer.NewDownloadFileNotFound()
    }
}
```

```

    return transfer.NewDownloadFileOK().WithPayload(namedReader).
        WithContentDisposition(fmt.Sprintf("attachment; filename=\"%s\"",
file.Name)).
        WithContentLength(uint64(stat.Size()))
}

```

4.12 Retrieve File Information

File information can be retried using either a file's ID or the path to the file. This is implemented via a simple query to the database to retrieve the information and an appropriate response is sent to the client.

The below snippet taken from `/src/server/handlers/file_info.go` shows how concise it is to do this with the libraries implemented.

```

var file models.File
tx := database.DB.Model(&schema.File{}).Where(&schema.File{UserID:
user.ID}).Find(&file, params.FileID)
if tx.Error != nil {
    return files.NewGetFileInfoNotFound()
}
return files.NewGetFileInfoOK().WithPayload(&file)

```

4.13 Create Directory

The user can create a directory within any directory within the storage directory specified in their configuration file. The user can specify the path to the new directory that they wish to create or specify the ID of the parent directory that they wish to create the folder in. Allowing the user to specify a parent directory ID here facilitates an easy workflow. They simply have to specify a flag and a directory ID instead of a long path to a deeply nested directory. All directories have unique IDs and thus locating directories in the database is straight-forward.



Below is a snippet from the directory handler located at `/src/server/handlers/directory.go`. It illustrates how directories are created in a specified directory or by default in the user configured home directory.

```
var parentDirID uint
if params.ParentDirectoryID != nil {
    parentDirID = uint(*params.ParentDirectoryID)
} else {
    homeDir, err = repo.GetHomeDir(user.ID, database.DB)
    if err != nil {
        return errNoParentDir
    }
    parentDirID = homeDir.ID
}

directory, err = findExistingDirByParentID(params.DirectoryName, parentDirID)

if err != nil {
    directory, err = repo.CreateDirectory(params.DirectoryName, parentDirID,
database.DB)
    if err != nil {
        return errCreateFailed
    }
}
```

4.14 Delete Directory

A user can delete a directory specified by the path to the directory or the directory's ID. The server will ensure that the user is the owner of this directory before deleting it. Once confirmed, all directory contents and the directory itself will be deleted from both the disk and the database.

The below snippet from `/src/database/schema/directory.go` illustrates how this is done.

```
// executeDelete removes the directory and everything contained within it from the
```



```

disk
func (dir *Directory) executeDelete() error {
    return files.Afs.RemoveAll(dir.Path)
}

func (dir *Directory) Delete(forceDelete bool, db *gorm.DB) (err error) {
    if err = dir.executeDelete(); err != nil {
        return err
    }

    if dir.Name == "home" {
        return ErrDeleteHomeDirNotAllowed
    }

    var filesInDir int64
    db.Model(File{}).Where(File{DirectoryID: dir.ID}).Count(&filesInDir)
    if filesInDir != 0 && !forceDelete {
        return ErrDirNotEmpty
    }

    if err = db.Where(File{DirectoryID: dir.ID}).Delete(&File{}).Error; err !=
nil {
        return err
    }

    if err = db.Unscoped().Delete(&Directory{}, dir).Error; err != nil {
        return err
    }

    return nil
}

```

4.15 List Directory Contents

The user can request to view the contents of a directory specified by the path to the directory or the directory ID. If the path is specified, the server retrieves the directory ID from the database and uses this to determine the directory contents.



Below is a snippet from `/src/server/database/directory.go` that illustrates how the database is queried to retrieve the contents of a directory. It returns a directory contents object so that this data can be easily passed through to other functions.

```
func GetDirContentsByID(dirID uint) (*models.DirectoryContents, error) {
    contents := &models.DirectoryContents{}

    directory := &schema.Directory{}

    tx := database.DB.Where("id = ?", dirID).First(directory)
    if tx.Error != nil {
        return nil, tx.Error
    }

    contents.CurrentDir = &models.Directory{
        FileCount: uint64(directory.FileCount),
        ID:         uint64(directory.ID),
        Name:       directory.Name,
        Path:       directory.Path,
        PathHash:   directory.PathHash,
    }

    if directory.ParentID != nil {
        contents.CurrentDir.ParentDirectoryID = uint64(*directory.ParentID)
    }

    tx = database.DB.Where(&schema.Directory{ParentID:
&dirID}).Find(&contents.SubDirectories)
    if tx.Error != nil {
        return nil, tx.Error
    }

    tx = database.DB.Where(&schema.File{DirectoryID: dirID}).Find(&contents.Files)
    if tx.Error != nil {
        return nil, tx.Error
    }

    return contents, nil
}
```

4.16 Move Directory

Directories can be moved by specifying either file paths or directory IDs. Moving directories works similarly to moving files. A directory is created using functions that were created for making a directory. Once the directory is created, the function created to move a file is called for each file in that directory. The functions created for the list directory command are used to retrieve a list of what files are in the specified directory.

4.17 Docker

We recommend that the server is run within the Docker container we provide. This is to avoid database dependency conflicts. The Dockerfile is used to build the Docker image. This Docker image can then be deployed easily with a docker-compose file. The docker-compose file provided is a default file that should be edited by the user as instructed in the user manual.

The Dockerfile is as shown below

```
FROM golang:latest
COPY ./ /synche/src

WORKDIR /synche/src
RUN apt update && \
    apt install jq -y && \
    download_url=$(curl -s
https://api.github.com/repos/go-swagger/go-swagger/releases/latest | \
jq -r '.assets[] | select(.name | contains("'"$(uname | tr '[:upper:]'
'[:lower:]')"'_amd64")) | .browser_download_url') && \
    curl -o /usr/local/bin/swagger -L'#' "$download_url" && \
    chmod +x /usr/local/bin/swagger && \
    mkdir /synche/build && \
    make generate && \
    make build

EXPOSE 9449 2121
```



```
CMD ["/synche/build/server", "serve"]
```



5. Development Workflow

We followed an Agile development approach throughout the duration of our project so that we could fully utilise the time that we had to create our project. We used a Kanban board, had daily stand-up meetings, had bi-weekly scrum meetings, and followed a strict Git practice. Furthermore, we developed both independently and in a pair programming manner. Pair programming was time-consuming, so we only took this approach at critical points of the development process or when we hit blockers.

5.1 Clubhouse Kanban Board

We used app.clubhouse.io to manage our kanban board. We found this tool to be extremely useful as it was easily linked with GitLab. On Clubhouse, initially, we created epics. Each epic related to a big chunk of work that has one common objective. For example, we created an epic for the user client. For every feature, bug, or chore we created a story and linked it to the related epic. There were six states that each story could be in. They are:

- **New**
 - This is where each story began. We discussed each story and if we decided that it should be included, we moved it to the icebox state.
- **Icebox**
 - We put stories here when we had decided that they would be implemented, but they were blocked by another story.
- **Ready for Development**
 - Once a story had no blockers, we moved it to this state.
- **Priority**
 - This state was for stories that had no blockers and needed to be focused on when we completed whatever story we were working on.



- **In Development**

- Here, we tracked stories that we were actively working on. This allowed us to see how long we were spending on each story and ensure that our work wasn't conflicting.

- **Completed**

- This state contained an archive of all the stories that we had completed. The Clubhouse bot automatically moved stories to this state when a branch was merged that contained a link to the story.

When we added a story, we labelled any relationships between that story and other stories. This helped us prioritise stories that were blocking other work. It also ensured that we didn't do any redundant work that would need to be rewritten when a blocker was removed.

Request that missing chunks be resent

In the event of that file chunk(s) go missing or fail to send, the server should request that the client re-sends the chunks that it didn't receive.

Edit Description

Tasks

+ Add Task...

Add to Story

Relationships...

External Links...

Attach Files...

Story Relationships

Blocked by Support nested directories

Blocks Handle "please resend these chunks" requests

Show 1 archived related story

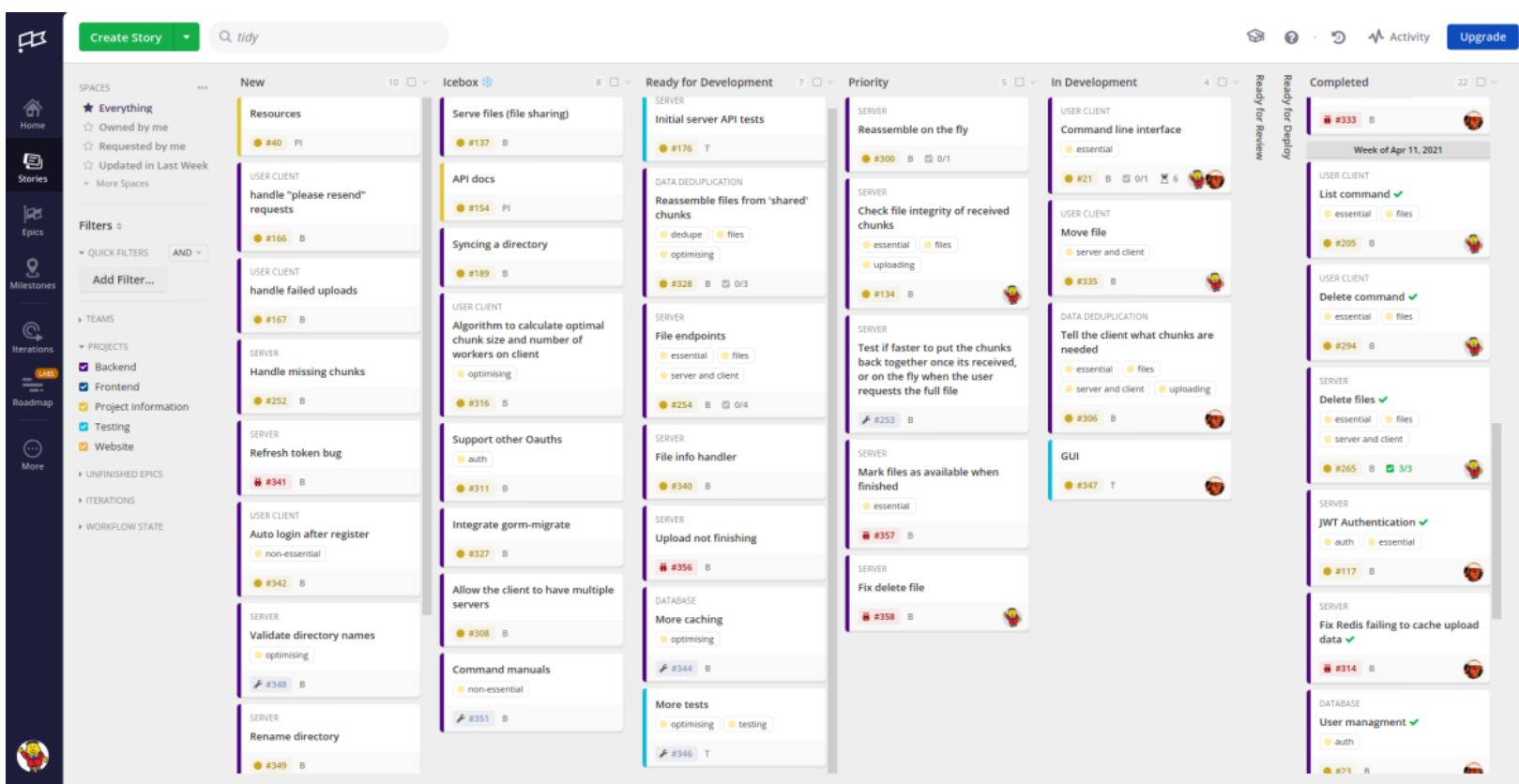
+ Add Story Relationship...

399

166

We also added labels to each story that indicated things such as if a story focused on optimising our application. This facilitated a smooth development process and allowed us to easily track the progress that we were making on both the project as a whole and individual features.





The screenshot above displays our kanban board from a point towards the middle of the development process. It shows how our stories were displayed and how they were organised into different states.

5.2 Git

We followed a strict Git commit practice and merge request practice. Each story was developed on its own branch. Each branch was named with the format:

```
[story-type] / [story-number] / [story-title]
```

The three story types were:

- Bug
- Feature
- Chore

Story numbers were taken from the automatically generated numbers on Clubhouse. The title also had to match the story title on Clubhouse. Each branch was developed locally. We would commit to our branches locally whenever we felt as though we had made an amount of progress that was worth bookmarking.

Commit messages followed [Chris Beam's Git commit guide](#). The main features of this guide are as follows:

- Separate subject from body with a blank line
- Limit the subject line to 50 characters
- Capitalize the subject line
- Do not end the subject line with a period
- Use the imperative mood in the subject line
- Wrap the body at 72 characters

When a story had been completed, the branch was rebased on the most recent version of master. This ensured that there were no merge conflicts. The branch was then pushed up to the remote branch on GitLab where merge requests were made, and we could review each other's code. Branches were labelled on GitLab so that we could easily search and find branches that had already been merged.



```
tara@Tazanator2000:~/go/src/gitlab.computing.dcu.ie/collint9/2021-ca400-collint9-coynemt2$ git branch
bug/ch155/create-directory-if-not-present
chore/ch215/tidy-server-files
chore/ch229/convert-to-use-appfs
feature/ch125/chunk-reassembler
feature/ch13/basic-server-api
feature/ch134/check-file-integrity-of-received-chunks
feature/ch175/add-database-management
feature/ch205/list-command
feature/ch206/set-default-config-values
feature/ch265/delete-file
feature/ch295/get-command
feature/ch32/file-uploader
feature/ch334/list-directory-contents-command-line-options
feature/ch335/move-file
feature/ch340/file-info-handler
```

The screenshot above displays some local branches and clearly illustrates the branch naming mechanism that we used.

Each merge request had to be approved by the other developer before being merged. We felt that though this would result in the highest code quality as we would be under peer pressure to commit quality work and a second set of eyes on the code often spotted potential bugs or alternative implementations that took future features into consideration. Once approved, the branch was merged into the master branch.

5.3 Meetings

On a daily basis, we held stand-up meetings. We decided to do this because we feared that we could become disconnected from the project due to having to work remotely. We wanted to ensure that we had a high level of communication throughout the entire process. In these meetings, we quickly caught up on what we had done the previous day, what we were doing that day, and if there was anything blocking the work that we were to do that day. Every second week, we held scrum meetings in which we reviewed if we would need to deviate from our initial plan. Closer to the project deadline, we had this meeting on a weekly basis to ensure that we weren't getting side-tracked and losing sight of important features.



6. Testing

6.1 Unit Testing

Unit testing was critical for ensuring units of code were functioning as expected. In order to isolate various parts of the code from the rest of the codebase, we made use of multiple mocking tools. Especially helpful was [mockery](#), which allowed us to create mock versions of interfaces and functions. The mock interfaces could then be used to ensure certain functions were correctly being called by a function that was the subject of the unit test. With it, function calls could be validated whether the correct parameters or parameter types were being passed to the function.

This following example is a mock FileUploadFunc, which was used to test the upload command.

```
// Create a new mock FileUploadFunc and return no error for any
// string input
fileUploader := new(mockers.FileUploadFunc)
fileUploader.On("Execute", mock.AnythingOfType("string")).Return(nil)
uploadCmd := cmd.NewUploadCmd(fileUploader.Execute)
```

Database testing was trickier, but with the help of some useful libraries such as [sqlmock](#), we were able to test functions that required database access.

```
db, mockDB, err := sqlmock.New()
require.NoError(s.T(), err)

s.mockDB = mockDB
s.mockDB.ExpectQuery("SELECT
VERSION()").WillReturnRows(sqlmock.NewRows([]string{"VERSION()"}).Add
Row(""))
s.DB, err = gorm.Open(mysql.New(mysql.Config{Conn: db}))
```

We also required some method of interacting with files in unit tests since the project revolves around files and data transfer, so to do this, we used [afero](#). Afero is “a filesystem framework providing a simple, uniform and universal API”, which was perfect as we could then make use of its MemMapFs for unit tests. MemMapFs is in an in-memory filesystem meaning that files created in it would not need to be removed after they were written to. Since it is in memory, it also sped up unit tests.

```
func TestMain(m *testing.M) {  
    files.SetFileSystem(afero.NewMemMapFs())  
    os.Exit(m.Run())  
}
```

6.2 Integration and Regression Testing

We made use of GitLab’s continuous integration tool to automatically run integration tests each time either of us added a merge request to the repository. The integration tests operate by cloning the repository, generating and installing the necessary requirements before running `go vet`, `go fmt` and `go test`. If these commands passed, the `go build` command would be executed to ensure the software is compiling correctly. This CI/CD pipeline served as our integration and regression testing.

We used [SwaggerHub](#) to test our API endpoints. SwaggerHub is an integrated API development platform. It brings together the core capabilities of the Swagger framework, along with additional advanced capabilities to build, document, manage, and deploy APIs.

We also performed manual API tests using [Insomnia](#), which is an API design tool that worked with our Open API specification. This allowed us to thoroughly test our API endpoints.



6.3 User Testing

User testing was a huge priority when we were creating our test cases. We wanted to ensure that both the client and the server interfaces were accessible and usable by the user. We organized some user testing towards the middle of the development phase when we had a minimum working product. At this point, not all features had been implemented in full but

User testing was a bit of a challenge as everything had to be done remotely. We did two rounds of user testing in order to get as much valuable feedback as possible. We sent our volunteers a link to a Google Drive where they could download the user manual so that they could install the application. For convenience, we provided them with the client and server executables, default synche-server.yaml and synche-client.yaml files, and the Dockerfile and docker-compose.yaml files. The drive also contained instructions on how we wanted them to test the application.

Users were instructed to execute some Synche commands such as creating a new user, creating a directory, and uploading a file. We asked them to note roughly how long it took them to complete each task.

The consistent feedback that we got from the first round of user testing was that the volunteers did not like having to use flags on the command line. During the initialise design phase, we enforced that every argument following a command on the command line needed to be preceded by a flag.

For example:

```
synche delete -p path/to/a/file
```

The consensus that we got from this was that some users found it confusing to need to remember the flags for every command. We realised it would be more user-friendly to allow the user to specify file paths by default as this was the preferred method for the majority of the users. Instead of flags being required for all arguments, we changed it so that they are only required when specifying an optional argument, a file ID, or a directory ID.



We changed the command arguments so that by default the user could enter (note that the `-p` flag is no longer necessary):

```
synche delete [path/to/file]
```

If they wished to specify an ID, they could still enter:

```
synche delete -i [file-id]
```

After implementing these changes, we reached out to our volunteers and requested that they complete the testing tasks once again with the new command defaults. All the volunteers that responded reported that the application was easier to use without forcing flags. They also reported that it took them less time to complete each task. However, it is worth noting that this could also be because they were familiar with the application from when they previously tested it.



7. Research, Challenges, and Resolutions

7.1 GoLang

Both developers had no previous experience with GoLang. This presented a significant learning curve throughout the entire development process. There are several elements of Go that make it incredibly fast and ideal for our use case. A core feature of Go is its ability to manage and facilitate concurrency. Go is also compiled to machine code. This means that it outperforms languages that are interpreted or have virtual runtimes. Go is both fast and lightweight, making it the ideal language to write Synche in.

When we began the development process, we ran into some issues with managing GOPATH and GOROOT. Initially, we had moved our project out of the GOPATH which resulted in issues with our packages. This was easily fixed by correctly setting up our project in the correct directories.

Throughout the development process, we frequently encountered errors that were unfamiliar to us due to our inexperience with Go. The best solution we found to this was pair programming because it helped us challenge our understanding of the issues we faced.

7.2 Caching

As Synche is all about speed we wanted to implement caching as soon as possible to increase the upload speed and data transfer. The time it takes to query the database is time that could be saved using caching. We initially used Redis to implement caching. However, we found that we were spending too much time on implementing code to validate and store data in Redis and un-marshalling and marshalling the data to and from JSON. This was not ideal and caused many issues throughout the project. We soon realised that there are much more suitable alternatives that are written in Go.



We found go-cache to be perfect for our needs as it allowed us to store structs in the cache and retrieve them easily. We identified values that are either read or written to frequently in the database, and implemented code to store and retrieve them from cache memory rather than the database. An example of this, is storing non-privacy sensitive user data in the cache using their access token as the key. This meant that every endpoint that uses access tokens to authenticate users could avoid having to revalidate and repeatedly query the database for user data, and instead could quickly retrieve the required data from the cache.

7.3 Concurrency

We ran into issues with goroutines managing workers. We were consistently getting read/write errors on closed pipes. After a large amount of investigating, we realised that this was being caused by the server closing the connection to the upload request because it's idling for too long. We discovered that this could be fixed by running the server with:

```
--read-timeout 100s --write-timeout 100s --cleanup-timeout 100s
```

Specifically, the cleanup timeout needed to be higher, as that specifies how long the server should wait before closing idle connections.

7.4 Database Management

We initially created our database tables using the mysql driver for go. However, this required that we write long complex queries in an non-user-friendly format. We found this code difficult to read and ended up researching libraries that we could use alternatively. We refactored the database to be created and managed using the gorm library instead.

This refactoring took a considerable amount of work and time but in the long run it makes our application more maintainable. It is more maintainable because the code is



more readable and `gorm` easily integrates with `gorm-migrate` which allows us to migrate if it is ever needed during future development.

7.5 Time Management

Time management during the development of this project was very difficult. We had to manage all the work for our final year modules whilst ensuring that we followed the schedule for this project to the best of our abilities. We found this particularly challenging towards the end of the semester when continuous assessment deadlines for our other modules were of a high density, and we needed to study for our exams.

Likewise, we managed this by shifting planned work to weeks that we did not have deadlines for other modules so that we could afford to focus on other assignments and studying when we needed to.

7.6 Covid-19

The ongoing Covid-19 pandemic challenged us on a daily basis. We both felt as though we are more efficient working in a lab or office. This is due to many reasons, such as the social element and convenience of being within a close proximity of co-workers.

Again, the best solution that we found to this was pair programming on voice calls. This introduced the social element that we were lacking. It also meant that we were both immediately available if something needed to be discussed urgently and time was not wasted by design decision blockers.



8. Future Work

8.1 Data Deduplication

A key feature of splitting the files into chunks is that it allows for data deduplication on the server side. We ended up with insufficient time in this project to implement data deduplication on the server side, however it can be noted that most of the code to implement it is already there and the only changes that are needed would be to update the database so that files are not reassembled upon upload completion, but rather, on the fly - when they are requested. It would be a great feature to add to our project as data deduplication goes hand-in-hand with fast uploads and more reliable transfer as Synche offers a more comprehensive and reliable solution to data storage solutions such as Google Drive Dropbox.

Files would be stored only in 1 MB or 500kb chunks on the server and this would allow us to check the hashes of multiple chunks and ensure that the same chunk is only stored once on the server. This would be especially handy for large files that are similar in content, such as video clips. Large video clips especially, may contain large chunks of data that are the same amongst other video clips. Multiple users uploading different lengths of the same video might have a different overall hash, but a large percentage of the chunks that make up the data might have the same hash. The full video files may differ by only a few chunks, and in a traditional system it would be impossible to deduplicate the files without splitting up the data. If the same chunk is already stored on the system the composite file can then be made up by an already stored chunk rather than storing the chunk again.

8.2 Metrics and Statistical Dashboard

Displaying the user's file transfer statistics on a Grafana dashboard would be an interesting visual, and it would allow the user to perform some analysis on their data management. This is purely cosmetic and would enhance the user experience.

