

# Development Book

4/3/2019

|                                   |           |
|-----------------------------------|-----------|
| <b>Team Members and Roles</b>     | <b>2</b>  |
| <b>Background</b>                 | <b>2</b>  |
| <b>Project Description</b>        | <b>3</b>  |
| <b>Project Requirements</b>       | <b>3</b>  |
| <b>Business Rules</b>             | <b>4</b>  |
| <b>Technologies Used</b>          | <b>5</b>  |
| <b>Design Patterns</b>            | <b>5</b>  |
| <b>Timeline</b>                   | <b>5</b>  |
| <b>Layering</b>                   | <b>7</b>  |
| Presentation Layer                | 8         |
| Data Layer                        | 8         |
| Database Layer                    | 8         |
| <b>Exception Handling</b>         | <b>9</b>  |
| <b>Performance and Refactor</b>   | <b>20</b> |
| <b>Testing</b>                    | <b>22</b> |
| <b>Packaging &amp; Deployment</b> | <b>25</b> |

## Team Members and Roles

1. Ryan Garcia -
    - a. Project Manager: In charge of coordinating with team members to assign responsibilities, manage documentation, and monitor team's progress on meeting deliverable deadlines.
    - b. Front-End Developer: Responsible for designing and developing the part of the website users can see and interact. This includes designing the UI and UX experience.
  2. Brandon Connors -
    - a. Back-End Developer: Responsible for designing and developing the part of the website users cannot see and interact with. This includes the database and data layer code.
  3. Aaron Kelly
    - a. Back-End Developer: Responsible for designing and developing the part of the website users cannot see and interact with. This includes the database and data layer code.
    - b. Assistant Project Manager: Responsible to inform everyone based on timeline and update record for Gantt chart.
  4. Shawn Xu -
    - a. Front-End Developer: Responsible for designing and developing the part of the website users can see and interact. This includes designing the UI and UX experience.
- 

## Background

Magic: The Gathering is a collectible and digitable card game where players build their respective decks with unique cards and battle other players. Because of all the different types of cards, a huge part of the Magic: The Gathering strategy is actually building your deck. Carefully crafting your deck is a pivotal function of the game and as players spend a significant amount of time swapping out cards and creating different decks for different occasions. It would be useful for these players to have some sort of way to efficiently build decks and easily view the decks they have created.

---

## Project Description

This project will be a web-based application that will allow Magic: The Gathering players to view the stats of individual cards and build custom decks based on these cards. This application will allow users to build a Magic: The Gathering from scratch. It will also allow users to be able to quickly look through current decks in order to add or remove cards from it. The stats of each card should be easily visible so the user can make quick decisions on which cards they want to keep in their deck.

---

## Project Requirements

- A user should be able to search for any cards that are relevant to the current version of the game
- A user should be able to sort/filter the the search results
  - Mana cost
    - [0 - 15]
  - Color
    - Red
    - White
    - Blue
    - Black
    - Green
  - Alphabetical
  - Type
    - Creature
    - Land
    - Instant
    - Sorcery
    - Enchant
    - Artifact
- A user should be able to create an account
  - Username
  - Fname, lname
  - Password
- A user should be able to create different decks

- A user should be able to add cards to their decks
  - The application should allow users to create no more than 10 decks in their bag
  - The application should
- 

## Business Rules

### **Users:**

- Users will be able to search through and view all standard legal cards WITHOUT creating an account
- User must sign up for an account before decks can be created and saved
- Users must sign into their account before they can perform transactions on the database (create decks/add cards)
- User must create a new deck with a name before they can begin adding cards and viewing deck statistics
- Users are limited to creating 10 total decks
- Users can have many decks but a deck belongs to only one user
- User accounts are stored in database table

### **Decks:**

- A Deck cannot contain more than 4 copies of a single card
- A Deck cannot contain more than 75 total cards
- A Deck can only be edited or deleted after it has been created and saved
- A Deck belongs to one user but a user can many decks
- A Deck is stored in database table and associated with the User table through a join table User\_Deck

### **Cards:**

- Card information is viewable to all users whether they have an account or not
  - Cards have unique IDs and names
  - Cards are not a limited resource, when a card is added to a users deck, it does not belong solely to that user or that deck, it is just referenced by the deck
  - Card IDs are associated with user decks and ONLY card IDs, we will be using an API to display card information based on the card ID that is given to the API call
  - Cards can belong to many users and belong to many decks
  - Cards that are removed from a users deck are only removed that deck, the cards information is still globally accessible through the search/filtering tool
  - Card information is not stored in our database, the only card information stored in the database are the card IDs associated with a users deck
  - Cards IDs are stored in a joined table Deck\_Card
-

## Technologies Used

MySQL - Database

Node.js - Back end

- Using Express for server set up

HTML/CSS/JavaScript

- JQuery
- 

## Design Patterns

**Repository pattern** - for a layer of abstraction over the database and caching of data

**Factory Pattern** - to create models without coupling them when they are used

**Builder Pattern** - used in conjunction with the factory pattern to create complex models from our base models.

**Singleton Pattern** - to reduce coupling and increase cohesion, by only using one instance of an object.

**Model View Controller(principle)** - for rendering our web pages, accepting http requests and manipulation of models

**Dependency Injection(principle)** - to reduce coupling and increase cohesion, by injecting dependencies into the constructors of objects.

---

## Timeline

This is the description of the application for major deliverables and milestones to be completed before the deadline according to their deadline. For more support in reading the timeline, a gannet chart may be used in addition to this timeline.

*All of the following are due at midnight according to their date, unless a different time is said.*

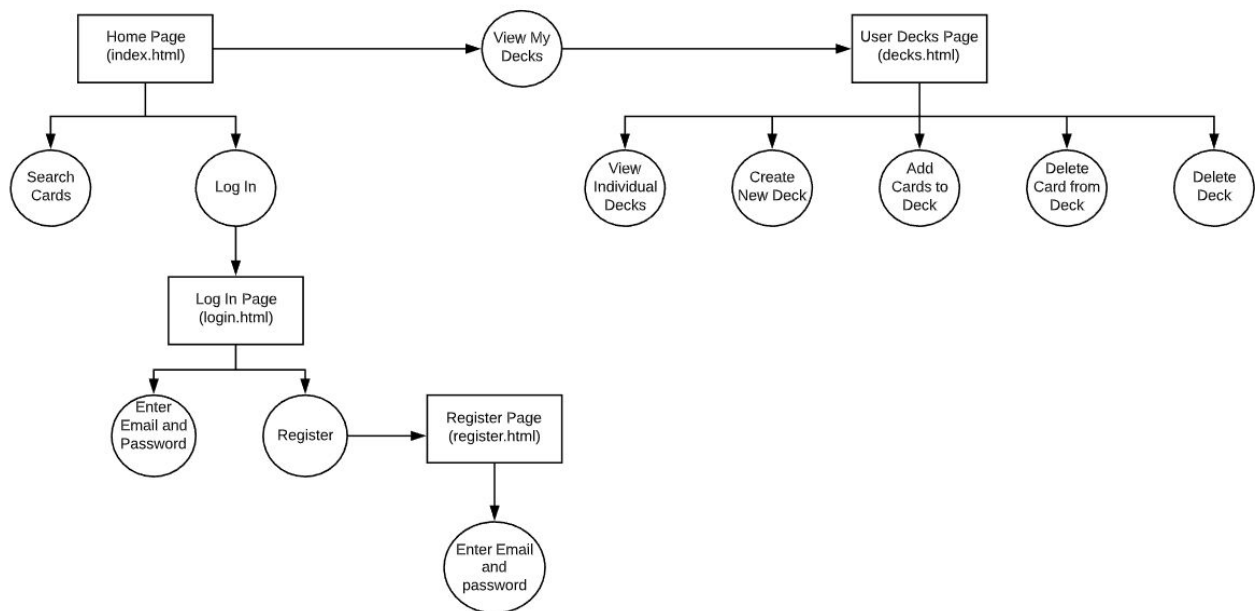
- Milestone 2 (this design documentation and design patterns): Feb 21, 2020
  - Describes the application project about the card game Magic the Gathering and to feature the whole as a documentation. This includes diagrams of the design diagram, UML, and other sources in assisting to further the understanding of the purpose.

- Milestone 3 (Layering) : Feb 28
  - Required to provide at least 3 layering, may use a database layering concept (Data layer, Presentation Layer, and Business Layer). Each layer consisting of roles functions describe in a logic to reduce and control the authority of other layering and to provide a relationship among classes.
- Milestone 4 (Exception Handling): March 20
  - Build an exception class or layer to provide a customized exception for location in Data Layer, Business Layer, and or database application. Cannot be used on the presentation layer. Dirty data, login error and or failure, as well as attempt to change the authority when not in the right role are expected to be account for the exception.
- Milestone 5 (Refactoring): April 3
  - Performance and Refactoring
  - Improving the performance by refactoring part of the codes such as reducing the number of lines and conveniently improvement in naming variables.
- Milestone 6 (Testing): April 17
  - A reference as a "appendices" ' added to the project in either a separate file or a build file and provide an instruction for testing. The file included an instruction and build utility for a testing framework. The testing framework designed to test each testing fragment of the codes.
- Milestone 7 (Packaging): April 24
  - Provide basic names of namespace, classes, package, and or other variable names used to group multiple sources files and folders. Used to provide for a package deployment.
- Final Project: April 27
  - Publishing the final change for the application project.
- Peer Evaluation : April 28
  - Evaluating each of your team members' performances and well participation include their effort after the final project.

## Layering

### Presentation Layer

This layer acts as the user's only interface to the system as it contains the front-end UI. This layer communicates directly with the data layer and only receives objects from there, it has no direct connection to the database. This means that no SQL should exist at this layer and the only validation that occurs is form validation. The presentation layer will consist of code made using HTML, CSS, and Javascript/JQuery.



### Service Layer -

Our service layer will consist of node.js express controllers to accept http requests, all CRUD methods involved with creating, retrieving, updating and deleting a User and a Users decks.



## Business Layer

One of the layers contains all methods needed to validate, confirm, and authorize the use of the actions received by the business layer before accessing the data layer. To be specific, one function consists of methods to validate a given data using

```
validate(var param) {...}
```

Which is responsible to clean data using sanitization and configure the value to harmless type. This will target all data specifically received from the service layer for actions such as base sql statement, advanced generic statement, prepared sql, and lastly an opening sql statement. They are operated at a data layer which would be coordinated by this layer for other layers to communicate to set between.

## Data Layer

The data layer will consist of methods to do the necessary CRUD operations to the database. E.g. Creates a corresponding record for a registered user, storing deck information for that user in the corresponding tables.

The data layer functions will be called by the layers above depending on different kinds of situations. This will provide all specific actions needed to meet the user's want: basic sql statements, advanced statements using prepared and multi-join, and an opening sql statement which let the user type sql statements himself/herself. The opening sql statement is useful in a case when none of the two other choices meet user's satisfaction and can provide a choice for a complicated statement involving subqueries, multi-join queries, and special commands like "case when".

## Exception Handling

With exception handling added, the software and code within the application are expected to account for all types of errors available or found appear during runtime and running. The following of the JavaScript files contains at least a method to initiate an error exception under different locations based on the layer of the software.

1. An exception will occur if a signed user doesn't have an existing deck under this person's account when attempting to open a deck. The error was placed under the one level before the presentation layer and no pass argument from the exception. To ensure any new signed user is coming prepared with a new deck to hold, this method is used to ensure the availability for them. Otherwise, they are placed in a console log where the developer is allowed to see for a future fix.

```
async newDeck(req,res){  
  try{  
    const result = await this.service.newDeck(req.body.id,req.body.name);  
    res.send(result);  
  }catch (e) {  
    res.send(e);  
  }  
}
```

a.

```
async newDeck(user,name){  
  try {  
    const result = await this.repo.create({user: user, name: name});  
    console.log(result);  
    return result;  
  }catch (e) {  
    console.log(e);  
  }  
}
```

b.

Under the DeckService.js where a controller, DeckController can initiate a super constructor to here.

2. Repositories under repositories folder at - create(), update(), delete()

```

async create(values){
  try {
    const query = queryBuilder.storedProcedure( method: 'CREATE',this.table, values);
    const params = queryBuilder.getParams(values);
    values.id = await this.database.execute(query, params)[0][0];
    const entity = this.make(values);
    this.entities.push(entity);
    return entity;
  }catch (e) {
    console.log(e);
  }
}

retrieve(id){
  return this.entities.find( predicate: (entity :T )=>{return entity.id === id});
}

async update(id,values){
  try{
    const query = queryBuilder.updateOne(this.table,id, condition: 'id',values);
    const params = queryBuilder.getParams(values);
    await this.database.execute(query,params);
    const entity = this.retrieve(id);
    const keys = Object.keys(values);
    keys.forEach((key :string )=>{
      entity[key] = values[key];
    });
    return entity
  }catch (e) {
    console.log(e);
  }
}

async delete(id){
  try {
    const query = queryBuilder.delete(this.table, condition: 'id');
    await this.database.execute(query, [id]);
    for (let i = 0; i < this.entities.length; i++) {
      if (this.entities[i] === id) {
        return this.entities.splice(i, deleteCount: 1);
      }
    }
  }catch (e) {
    console.log(e);
  }
}
}

```

- a.
- b. The code file is located in a business layer. As the database is expecting to have an exception for non-existent data, all of the following methods placed errors to console log for a future fix.

### 3. Authentication Service under service folder - authenticate()

Throws an exception when the user provides invalid email/username and password, the authenticate() method calls the validate() method to check if the email exists in the database, if it does then it checks if the hashes match.

```
class AuthenticationService{
  constructor(repo){
    this.repo = repo;
  }
  async authenticate(email,password){
    const valid = this.validate(email,password);
    if(valid){
      const user = this.repo.getByEmail(email);
      return await jwt.sign(user);
    }else{
      return 'bad login';
    }
  }
  async validate(email,password){
    let validEmail = this.repo.exists( prop: 'email',email);
    let validPass;
    if(validEmail){
      const user = this.repo.getByEmail(email);
      const givenHash = await encryption.hash(password);
      const correctHash = user.getPassword();
      validPass = givenHash === correctHash;
    }
    return validEmail && validPass;
  }
}
```

The validate method will then return a boolean value depending on the results of the two boolean values for email and password.

If the value returned is false it returns “bad login” -- considering returning this to the user and returns something else to the developers so that the information can be kept track of.

### 4. Encryption under util folder which is accessible by all classes - module.exports.hash

```
const bcrypt = require('bcrypt');
module.exports.hash = (data) => {
  return new Promise( executor: (res,rej)=>{
    const saltRounds = CONFIG.ENCRIPTION.SALT;
    bcrypt.hash(data,saltRounds, cb: (err,hash)=>{
      if(err){rej(err)}
      res(hash);
    });
  });
};
```

- a.
- b. Instead of a class for a layer structure, it is treated as an extensive tool for all classes to access. Whenever a login is attempted, the password detail is passed

to the encryption file for the purpose of hash the password to prevent the data from visually real passwords. However, any exception through the hash password is failed would be rejected instead. The failed attempt is rejected because neither of the password nor the result after been hash would need to be returned.

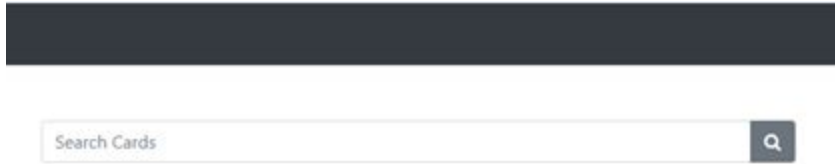
5. Database under src folder - execute()

```
async execute(statement,params){  
  return new Promise( executor: (resolve,reject)=>{  
    this.conn.execute(statement,params,(err,res,fields)=>{  
      if(err){reject(err);}   
      resolve(res);  
    });  
  });  
}
```

- a.
- b. Every database layer always comes with an exception prevention method to ensure the database source doesn't break or errors with transactions for many purposes such as account lock and writing at the same time. The method above executed a Promise which is expected to return a value after a given SQL statement is executed along with parameters. Once the return value is expected, the method determines it will return its value as a default and will be rejected if an error occurs. This provides an error that is rejected and continues to return the values.

## Card Exceptions:

Cards can be searched by name by the public to view without an account.



The name that is provided in the search bar is sent as a GET request parameter to the CardController's search method. The controller then calls the CardService's search method, providing the query parameters of the request. The results are rendered to the page in a list showing images of each card, if an error occurs or no cards are found, a "No Cards Found" message is displayed to the user.

```
async search(req,res) {  
  const results = await this.service.search(req.query);  
  res.render('index', {results: results});  
}
```

The CardService is injected with an ApiProxy class, which makes requests to a 3rd party API for card information. The proxy's query method is called and the results are returned to the controller.

```
async search(values){  
  return await api.query(values);  
}
```

The ApiProxy's query method queries the 3<sup>rd</sup> party API using the provided parameters which if any errors occur in the API it returns an empty array. The results are then processed and formatted to fit our more simplified model. If any errors occur here, an empty array is returned and the event is written to the server log(not currently implemented), otherwise, an array of Card Models are returned to the service.

```
const query = async(query)=>{
  console.log(query);
  query.gameFormat = CONFIG.GAME_FORMAT;
  const totalResults = await mtg.card.where(query);
  console.log(totalResults);
  const currentResults = [];
  totalResults.forEach((queryResult)=>{
    if(!alreadyExists(queryResult.name,currentResults)) {
      if (hasMultipleVersions(queryResult.name, totalResults)) {
        const imageVersion = getVersionWithImage(queryResult.name, totalResults);
        currentResults.push(imageVersion);
      } else {
        currentResults.push(queryResult);
      }
    }
  });
  return process(currentResults);
};
```

## Viewing a Card profile:

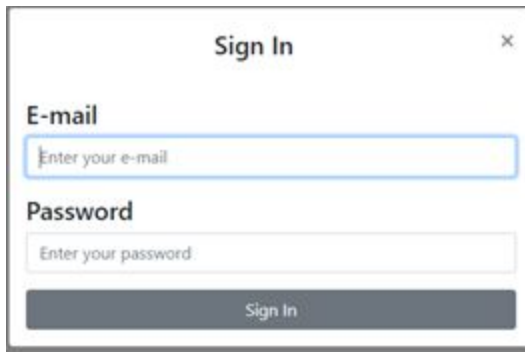
If an image of a card is clicked, the user is taken to /cards/:id endpoint, which calls the CardController's cardProfile method. The 3<sup>rd</sup> party API is queried for the card associated with the ID. If nothing is returned a "Card not found" message is displayed to the user and the event is written to the server log. (not currently implemented)

```
async cardProfile(req,res){
  const id = req.params.id;
  const card = await this.service.getCard(id);
  res.render('card',{card:card});
}
```

## Account Exceptions:

### Login:

Login attempts are first verified client side, an ajax POST request is made to the authentication controller's verify endpoint, providing the email and password in the body.



The image shows a 'Sign In' modal window. It has a title bar with 'Sign In' and a close button. Below the title bar, there are two input fields: 'E-mail' and 'Password'. The 'E-mail' field has a placeholder text 'Enter your e-mail'. The 'Password' field has a placeholder text 'Enter your password'. Below these fields is a 'Sign In' button.

```
async verify(req, res) {  
  const email = req.body.email;  
  const password = req.body.password;  
  const authenticated = await this.service.authenticate(email, password);  
  res.send(authenticated);  
}
```

The verify endpoint calls the authentication service's authenticate method. The authentication service will sanitize and validate the provided email and password using regular expressions (currently not implemented). If the provided values contain any malicious strings, or are an invalid format, false will be returned to the controller.

If the sanitization and validation pass, the service will then check to see if there is an account associated with the email and that the provided password matches. If either condition fails, false is returned to the controller.



```
async authenticate(email,password){
  email = email.toLowerCase();
  return await this.validate(email,password);
}
async validate(email,password){
  let valid;
  valid = this.validateEmail(email);
  if(valid){
    valid = await this.validatePassword(email,password);
  }
  return valid;
}
async validatePassword(email,password){
  const user = this.getUser(email);
  const hash = user.getPassword();
  return await encryption.compare(password,hash);
}
validateEmail(email){
  return this.repo.exists( props: 'email',email);
}
```

If the response is false, an “Incorrect email or password message” is displayed on the form.

A screenshot of a web form titled "Sign In" with a close button (X) in the top right corner. Below the title, a red error message reads "Incorrect e-mail or password". The form contains two input fields: "E-mail" with the text "41221" and "Password" with masked characters "\*\*\*\*\*". At the bottom is a dark grey button labeled "Sign In".

If the response is true, the form then posts to /login, which again calls the authentication services to authenticate method. If the authentication fails on the post to log in, the client is redirected to the index page, otherwise, a session is created for the user and they are redirected to their dashboard page(currently not implemented).

```
async login(req,res) {
  const email = req.body.email;
  const password = req.body.password;
  const authenticated = await this.service.authenticate(email,password);
  if(authenticated){
    req.session.user = this.service.getUser(email);
    res.send(req.session.user);
  }else{
    res.redirect('/');
  }
}
```

## Registration:

Registration information is first verified client-side, regular expressions are used to sanitize and validate the provided information. If any of the fields are invalid, an error will be displayed on the form. If all fields are valid, the form will POST to the UserController's register method, if the registration fails, the user will be informed that the account could not be processed at this time.



```
async register(req,res){
  const email = req.body.email;
  const password = req.body.password;
  const result = await this.service.register(email,password);
  res.send(result);
}
```

The UserController then calls the UserService's register method, the provided email and password, are then sanitized and validated. (not currently implemented) If the provided values contain any malicious strings, or are an invalid format, false will be returned to the controller.

The provided password is then hashed and the UserRepository's create method is called. If the repository fails to create the account, false will be returned to the controller.

```
async register(email,password) {  
  const hash = await encryption.hash(password);  
  const lowerCaseEmail = email.toLowerCase();  
  return await this.repo.create({email:lowerCaseEmail,password:hash});  
}
```

The UserRepository, create method attempts to execute a Stored Procedure in the database for account creation. If an error occurs during execution of the stored procedure, the error will be written to the server's log (not currently implemented) and a value of false will be returned to the service

```
async create(values) {  
  try {  
    const query = queryBuilder.storedProcedure( method: 'CREATE',this.table, values);  
    const params = queryBuilder.getParams(values);  
    values.id = await this.database.execute(query, params)[0];  
    const entity = this.make(values);  
    this.entities.push(entity);  
    return entity;  
  } catch (e) {  
    console.log(e);  
  }  
}
```

## Deck CRUD:

If a user is logged into their account, they can create/retrieve/update and delete a decklist of cards. When they perform a CRUD operation on their deck, an ajax POST request is made to the DeckController's associated endpoint. If the service fails, the CRUD operation executes a message that will be displayed "this operation cannot be completed as this time" and the event will be written to the server log. (not currently implemented)

```
async newDeck(req,res) {  
  try{  
    const result = await this.service.newDeck(req.body.id,req.body.name);  
    res.send(result);  
  } catch (e) {  
    res.send(e);  
  }  
}
```

The controller then calls a method in the DeckService that associated with the CRUD operation. Values

are sanitized and validated, If the provided values contain any malicious strings, or are in an invalid format, false will be returned to the controller. (not currently implemented).

If the repository fails to execute the operation in the database, false will be returned to the controller.

```
async newDeck(user, name) {  
  try {  
    return await this.repo.create({user: user, name: name});  
  } catch (e) {  
    console.log(e);  
  }  
}
```

The DeckService will then call a method in DeckRepository associated with the operation. It will attempt to execute a Stored Procedure in the database to perform the operation. If an error occurs during execution of the stored procedure, the error will be written to the server's log (not currently implemented) and a value of false will be returned to the service.

```
async create(values) {  
  try {  
    const query = queryBuilder.storedProcedure( method: 'CREATE', this.table, values);  
    const params = queryBuilder.getParams(values);  
    values.id = await this.database.execute(query, params)[0];  
    const entity = this.make(values);  
    this.entities.push(entity);  
    return entity;  
  } catch (e) {  
    console.log(e);  
  }  
}
```

## Performance and Refactoring

When refactoring the project's code, there are things to consider:

1. **New functionality should not be introduced during refactoring.** Refactoring is a separate process from development where it is making internal changes to improve readability and performance, but it should not change the external behavior of the code.
2. **All existing tests should pass after refactoring.** Do not modify existing tests in order to appease the refactoring work done, the existing functionality of the code must not be disturbed. A failed existing test gives an indication that the refactoring introduced a bug in the code that needs to be fixed. In some cases there could be an issue with the tests themselves, such as the existing tests being too low level, however, this should be considered last.

### Refactoring Techniques

We have identified refactoring techniques to be carried out throughout the development of the project:

1. **Extraction Method** - This is where longer pieces of code are broken down into smaller parts in order to find reusable pieces of code that can be turned into functions. For example, in the project sometimes it needs to be determined whether a card has multiple versions. So instead of writing the code for each time, it is needed, a separate function is made and called when needed.

```
const hasMultipleVersions = (name, results) =>{  
  let hasMultiple = false;  
  const versions = getAllVersions(name, results);  
  if(versions.length > 1){  
    hasMultiple = true;  
  }  
  return hasMultiple;  
};
```

2. **Make Variable/Function Names Descriptive** - Don't be vague with function/variable names, at a glance it should give the reader a good idea as to what it is. The vagueness makes the code harder to read and ultimately harder to maintain. An example of this project would be the function to add a new deck to the database, just looking at the name of the function you know what it's gonna go before even looking at its contents.

```
async newDeck(req,res){
  try{
    const result = await this.service.newDeck(req.body.id,req.body.name);
    res.send(result);
  }catch (e) {
    res.send(e);
  }
}
async addCard(req,res){
  const deckId = req.params.id;
  const cardId = req.body.cardId;
  const copies = req.body.copies;
  const result = await this.service.addToDeck(deckId,cardId,copies);
  res.send(result);
}
```

## Testing

This application uses unit testing in order to make sure that each portion of the codebase runs as intended. As the project is being tested, the framework we will initially be used with is Jest which is a Javascript test framework made by Facebook design to focus on simplicity. With Jest implemented in one of the proxy classes, the purpose is to search out for possible errors produced during and after the program execution.

### Test for Proxy:

```
const dotenv = require('dotenv');
dotenv.config();
const Proxy = require('../src/core').Proxy;
const proxy = new Proxy();

it('Gets card named Hero of Precinct One', async () => {
  const cardId = "87732718-1067-4e5f-a76d-409539c9ef3f";
  const card = await proxy.get(cardId);
  expect(card.name).toEqual("Hero of Precinct One");
});

it('Get first 175 cards', async () => {
  const cards = await proxy.getAll();
  expect(cards.length).toEqual(175);
});
```

This is to test the “proxy” which is responsible for connecting to the api, The first test is to run a card search using the proxy.get(cardId) method, the “card.name” variable should correspond to the ‘cardId’ entered. It makes sure that we have a connection with the api and information is being retrieved correctly from it. The second test is to test if the proxy.getAll() method correctly returns a list of cards that has a length of 175, since the program is designed to not retrieve thousands of cards at once, therefore it’s supposed to top at a limit.

### Test for a database:

```
const dotenv = require('dotenv');
dotenv.config();
const Database = require('../src/core').Database;
const database = new Database();

it('Connects to database without issues', async () => {
  expect(async () => {await database.connect()}).not.toThrow();
});

it('Gets admin user by ID', async () => {
  const adminId = 17;
  const result = await database.execute( method: "RETRIEVE", name: "USER", values: {id:adminId});
  expect(result[0][0].email).toEqual("admin@mtgdeckbuilder.com");
});
```



To test for database access, the most major part of the SQL is to check for connection and use a basic command. Using a basic command and the quality of the connection is considered worthy enough to intuitively know the result for the most part of the systems and as well as MySQL.

#### Test for server:

```
const Proxy = require('../src/core').Proxy;
const proxy = new Proxy();
const Database = require('../src/core').Database;
const database = new Database();
const Server = require('../src/core').Server;
const server = new Server(database, proxy);

it('Server starts without issues', async () => {
  expect(async () => {
    await database.connect();
    server.start();
  }).not.toThrow();
});

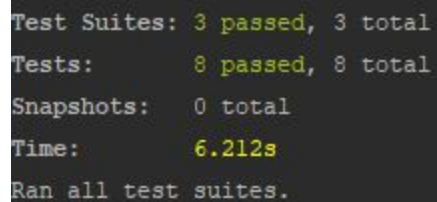
it('Server registers controllers', async () => {
  expect(async () => {
    const hasControllers = server.ctrl.length > 0;
    expect(hasControllers).toEqual(true);
  }).not.toThrow();
});

it('Server registers services', async () => {
  expect(async () => {
    const hasServices = server.svc.length > 0;
    expect(hasServices).toEqual(true);
  }).not.toThrow();
});

it('Server registers repositories', async () => {
  expect(async () => {
    const hasRepositories = server.repo.length > 0;
    expect(hasRepositories).toEqual(true);
  }).not.toThrow();
});
```

The image above shows the different checks applied to the server. The first check sees if the server starts without issue, which involves connecting to the database and then starting the server. The second check involves registering the controllers which will be used to manipulate the models. The third check makes sure that the server has registered the services, by checking the amount of services found in the server by an array. If the length is 0 then the server has not successfully registered the services. The last check is seeing if the server has registered the repositories, which it does by checking the amount of repositories currently on the server. Again, if the array returned has a length of zero then it can be concluded that the server was unsuccessful in registering the repositories.



**Results of all tests:**A screenshot of a terminal window with a dark background and light-colored text. The text displays the results of running tests: 'Test Suites: 3 passed, 3 total', 'Tests: 8 passed, 8 total', 'Snapshots: 0 total', 'Time: 6.212s', and 'Ran all test suites.'.

```
Test Suites: 3 passed, 3 total
Tests:      8 passed, 8 total
Snapshots:  0 total
Time:       6.212s
Ran all test suites.
```

This is the result of all the current tests in the implementation. It can be seen that all the tests mentioned above have passed as of the date written on the development book's cover.

---

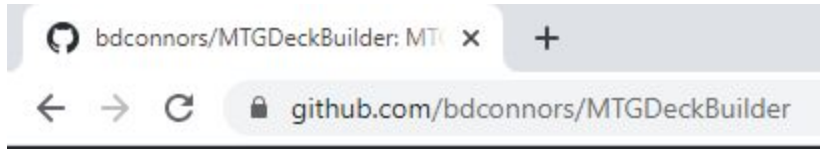
## Deployment & Packaging

### Packaging

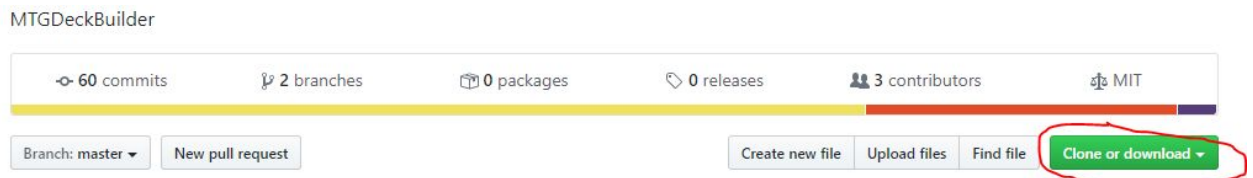
How to clone project repository using the command line:

*\*Git must be installed on your system before continue*

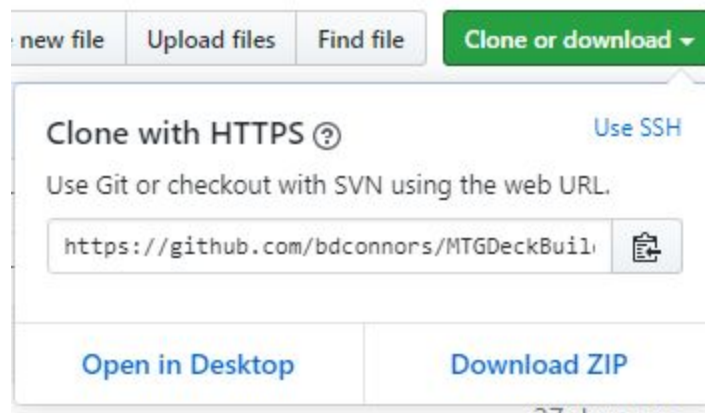
1. The project code is located [here](https://github.com/bdconnors/MTGDeckBuilder) (<https://github.com/bdconnors/MTGDeckBuilder>)



2. Click "Clone or download"



3. To clone the repository using HTTPS, under "Clone with HTTPS" click the icon next to the url.
  - a. Also from here you can open the repository in your desktop or download a zip version of the codebase.



4. Open GIT Bash
5. Change the current working directory to one where you want the cloned repository to be.

6. Type “git clone” and then put the URL that was used in step 3.  
*git clone <https://github.com/bdconnors/MTGDeckBuilder>*
7. Run the command. A local repository should be made of the project codebase.

## Deployment

Since there are many deployments offer to assist our project in many ways, we will be used as a simple and popular one called Heroku.

Prereq-requisite

- Heroku installed
  - NPM installed
  - Heroku free account
1. Open the command-line interface and locate the project directory
  2. Type the following command. It will install all needed library sources based on package.json  
“npm install”
  3. Then in the same directory, type the “heroku login” to log in your heroku account.
  4. Type “heroku create”. It will automatically create a new one with random name and random domain to host the project.
  5. Finally, type the following: “git push heroku master”. It will push all projects to include modules and packages into your account’s heroku, a cloud where a project will be run on live.
  6. To view it via a live web, open heroku account and click a project. Then click “Open App”. It will instantiate a new tab to show you our web.