CM2010: Software Design and Development

Mid-term assignment

Arjun Muralidharan

23rd December 2020

Contents

List of Figures

List of Listings

1 Part 1: Module Coupling & Cohesion

For this part of the assignment, we will inspect a Cryptocurrency Trading application created as part of the module "CM2005: Object Oriented Programming" in the degree.

Description & Purpose The trading application, called "Merklerex", attempts to create trades on a fictional cryptocurrency market with a starting capital of Bitcoin. The goal is to place orders, execute them, and generate a profit after stepping through several timeframes to execute trades. The basic structure (for the explanations below) is that the application maintains a `Wallet` object, which holds the users current balance of cryptocurrency.

- A `CSVReader` class is used to extract data from a CSV file of orders.
- A `OrderBook` class ingests these orders and maintains them as an object.
- A `Merklebot` class contains all the automated logic to trade in an automated fashion, and adds orders to the `OrderBook` and resulting balance changes to the `Wallet`.
- Finally, a `MerkleMain` class contains all functionality to control the application via user interaction.

The next sections describe examples of cohesion and coupling among these four classes. The application contains other classes, but these are not in scope for this analysis.

1.1 Cohesion

Example 1: Functional Cohesion We can find a good example of functional cohesion in the `Wallet` class. This module has the sole **function** of maintaining the user's currency wallet. It limits its functionality to managing currency, running checks against the wallet, and providing information about the wallet to other modules. This level of cohesion is well observable in the class's header file (Listing 1), which shows that all the class functions are related to the same business function. I argue that this is a well-designed class as functional cohesion is desirable.

Example 2: Procedural Cohesion The `Merklebot` class displays a level of procedural cohesion. It executes various functions across different pieces of data while related to the overall process of running an automated trade. For example, this module creates market predictions based on market data, creates orders based on those predictions, executes those orders, and updates the `Wallet`. While this is *logically cohesive*, the implementation here is not desirable as the module works on different data, and therefore the overall complexity is very high. It also generates logs, which is a different function overall. I've demonstrated this by picking two functions from this module (Listing 2) that operate on very different data and perform different functions procedurally related. Interestingly, this property of procedural cohesion becomes apparent when looking at the module as a whole; however, when looking at only the public API of this module, it seems more logically and functionally cohesive and less procedural. The interface maintains a good level of cohesion, but I can improve the module by moving some aspects of public functions and most private functions to a different module.

1.2 Coupling

Example 1: Data coupling An example of data coupling, where one module provides the data input for the latter module, can be seen in how the `CSVReader` class provides input for the `OrderBook` class. Here, I've extracted the function that calls the `CSVReader` class, the function that creates the input data and how it is then processed in the original `OrderBook` class (Listing 3).

Example 2: Content coupling An example of content coupling is visible in the `MerkleMain` and `MerkleBot` classes, which operate on a common data area. They both access the `Wallet` as a data object and operate on it while doing different things with it. To demonstrate this coupling, I've extracted a function from each of these classes (Listing 4) that operate on the same data. In this case, this kind of coupling is fairly safe, as the data is not in a global space and passed from one module to the other in a controlled, one-way manner.

2 Part 2: Unit Testing Activity

For this part of the assignment, I chose the `norestforthewiccad` RESTful API for which to write tests. I used the MochaJS and ChaiJS frameworks for these tests.

2.1 Scope

I decided to design a test for CRUD-style operations, testing for the applications capability to successfully **read**, **update** and **create** records via the API. These three operations are the foundation of the three test sets created.

The tests attempt to facilitate the development of these basic operations but do not seek full coverage of edge cases and failures that may occur, especially data integrity. The tests assume the limited number of spells available in the source data. Further development would be needed to test against a large set of data, and I could employ more sophisticated testing strategies to handle test data.

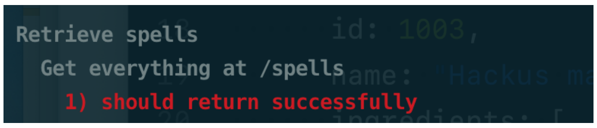
2.2 Test Set 1: Read — `GET /spells`

Strategy I decided first to ensure that the `spells` endpoint returns successfully when the application places a GET request against it. I then decided to extend functionality by checking that the response contained spells with the correct number of attributes that a spell should have. Finally, I decided to build functionality to retrieve a specific spell and make sure that the returned spell is indeed the one I had requested. This strategy allowed gradually building out the endpoint's data model and retrieving more specific data with each test. It also ensured that we first built functional middleware before populating it with business logic.

Test Set 1, Test 1 This test checks for the endpoints response code and expects a successful response, as shown in Listing 5.

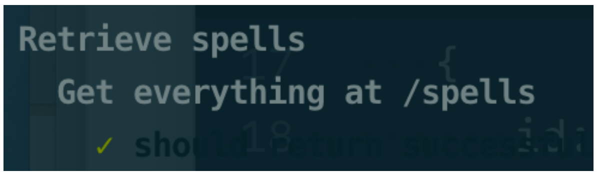
The before and after test output is visible in Figure 1.

Figure 1. *Test output for test set 1, test 1*



```
Retrieve spells
  Get everything at /spells
    1) should return successfully
```

(a) *Fail*



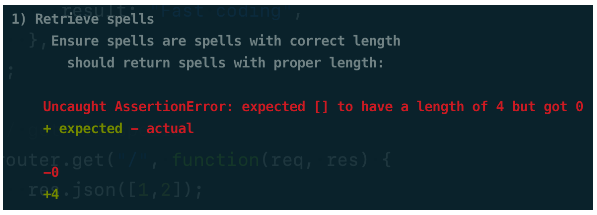
```
Retrieve spells
  Get everything at /spells
    ✓ should return successfully
```

(b) *Pass*

Test Set 1, Test 2 This test checks if the response consists of an array of objects, each with precisely four attributes. To make this test pass, I had to return the full contents of the `spells` array and then check for it by iterating over each spell and counting the number of keys. We also check that the body of the response is, in fact, an array, as shown in Listing 6

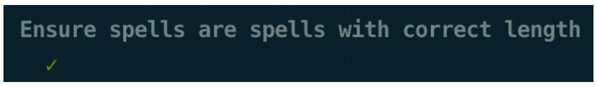
The before and after test output is visible in Figure 2.

Figure 2. *Test output for test set 1, test 2*



```
1) Retrieve spells
     Ensure spells are spells with correct length
       should return spells with proper length:

   Uncaught AssertionError: expected [] to have a length of 4 but got 0
   + expected - actual

   -0
   +4
```

(a) *Fail*



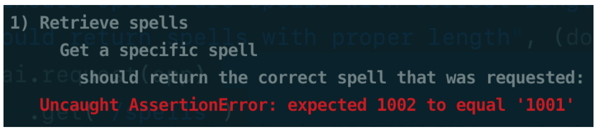
```
Ensure spells are spells with correct length
  ✓
```

(b) *Pass*

Test Set 1, Test 3 The final test in this set requests a specific spell by passing an additional parameter to the route at `/spells/:id` and checks if the returned spell has the same ID as was requested, as shown in Listing 7.

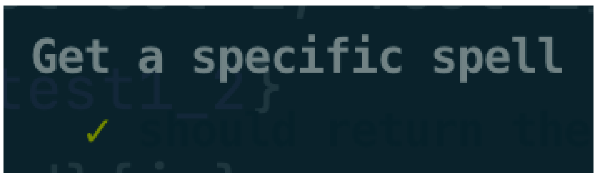
The before and after output of this test is shown in Figure 3.

Figure 3. *Test output for test set 1, test 3*



```
1) Retrieve spells
     Get a specific spell
       should return the correct spell that was requested:
   Uncaught AssertionError: expected 1002 to equal '1001'
```

(a) *Fail*



```
Get a specific spell
    ✓ should return the
```

(b) *Pass*

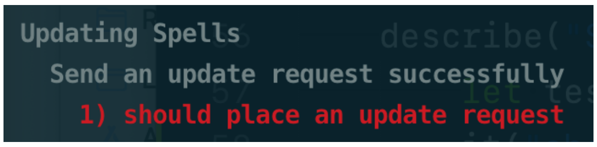
2.3 Test Set 2: Update — `PUT /spells/:id`

Strategy For this set of tests, I decided to progressively test updating a spell, first by trying to place a successful `PUT` request against a specific spell ID and then trying to edit a single attribute, followed by updating the entirety of a spell. Again, this allowed me to gradually expand the data model of this function as I tested.

Test Set 2, Test 1 This test checks for the endpoints response code and expects a successful response, as shown in Listing 8.

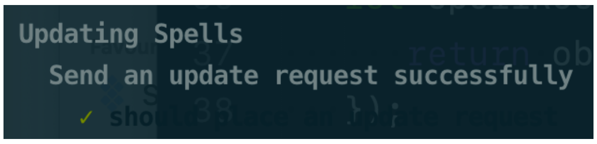
The before and after test output is visible in Figure 4.

Figure 4. *Test output for test set 2, test 1*



```
Updating Spells
  Send an update request successfully
    1) should place an update request
```

(a) *Fail*



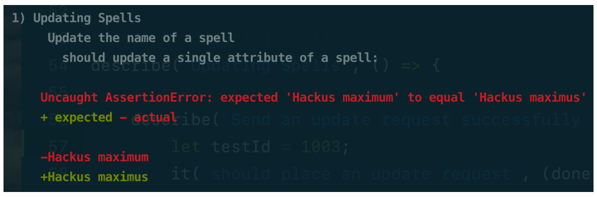
```
Updating Spells
  Send an update request successfully
    ✓ should place an update request
```

(b) *Pass*

Test Set 2, Test 2 This test tries to update the name of an existing spell by sending that attribute via a PUT request and checking if the updated spell, retrieved by ID, does have an updated name, as shown in Listing 9.

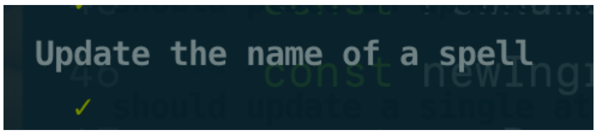
The before and after test output is visible in Figure 5.

Figure 5. *Test output for test set 2, test 2*



```
1) Updating Spells
     Update the name of a spell
       should update a single attribute of a spell:

    Uncaught AssertionError: expected 'Hackus maximum' to equal 'Hackus maximus'
    + expected - actual

    -Hackus maximum
    +Hackus maximus
```

(a) *Fail*



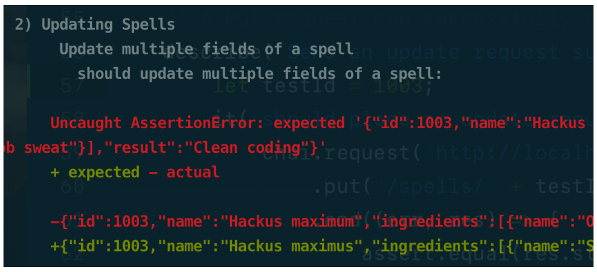
```
Update the name of a spell
  ✓
```

(b) *Pass*

Test Set 2, Test 3 The test expands on the previous test and attempts to update all fields of a specific spell, as shown in Listing 10.

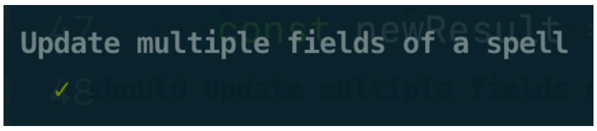
The before and after output of this test is shown in Figure 6.

Figure 6. *Test output for test set 2, test 3*



```
2) Updating Spells
     Update multiple fields of a spell
       should update multiple fields of a spell:

    Uncaught AssertionError: expected '{"id":1003,"name":"Hackus
b sweat"}],"result":"Clean coding"}'
    + expected - actual

    -{"id":1003,"name":"Hackus maximum","ingredients":[{"name":"O
    +{"id":1003,"name":"Hackus maximus","ingredients":[{"name":"S
```

(a) *Fail*



```
Update multiple fields of a spell
  ✓
```

(b) *Pass*

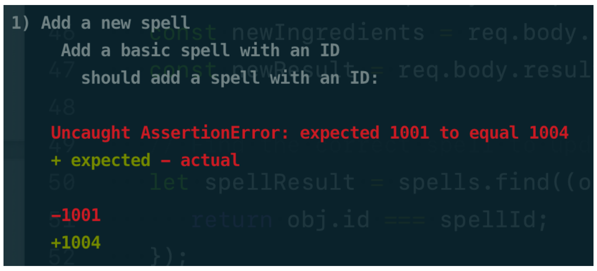
2.4 Test Set 3: Write — `POST /spells`

Strategy For the third set of tests, I attempt to create a new spell and store it in the application's memory. I again attempt to gradually build this new dummy spell with just an ID and check if the application created the spell with that ID. I then expand on this to add more fields in a new spell and check that I cannot create a spell if another spell with the same ID already exists (duplicate prevention).

Test Set 3, Test 1 This test places a `POST` request with a spell that only has an ID field. It checks if it can create the spell and if it gets returned correctly, as shown in Listing 11.

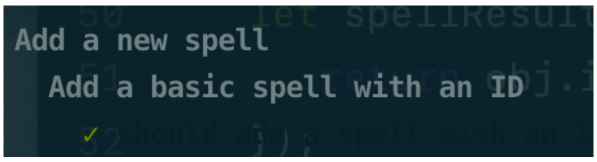
The before and after test output is visible in Figure 7.

Figure 7. *Test output for test set 3, test 1*



```
1) Add a new spell
     Add a basic spell with an ID
       should add a spell with an ID:

    Uncaught AssertionError: expected 1001 to equal 1004
    + expected - actual

    -1001
    +1004
```

(a) *Fail*



```
Add a new spell
  Add a basic spell with an ID
    ✓
```

(b) *Pass*

Test Set 3, Test 2 This test goes further by adding more fields to the spell created and ensures that the returned spell is identical to the one I wanted to create, as shown in Listing 12.

The before and after test output is visible in Figure 8.

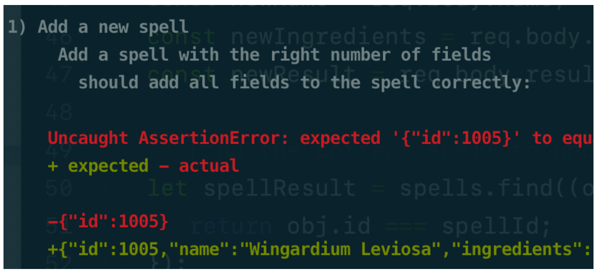
Test Set 3, Test 3 The final test tries to add a spell that already exists in the application, with the goal that this should not create a duplicate also not produce any error, as shown in Listing 13.

The before and after output of this test is shown in Figure 9.

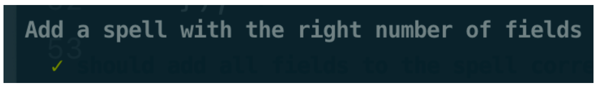
3 Part 3: Secure Programming

For this part of the assignment, we will again use the Merklerex application used in part 1 and describe strategies for improving the application's security.

Figure 8. *Test output for test set 3, test 2*

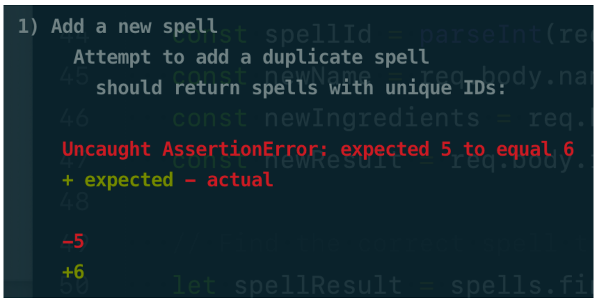

```
1) Add a new spell
     Add a spell with the right number of fields
       should add all fields to the spell correctly:

    Uncaught AssertionError: expected '{"id":1005}' to equ
    + expected - actual

    -{"id":1005}
    +{"id":1005,"name":"Wingardium Leviosa","ingredients":
```

(a) *Fail*

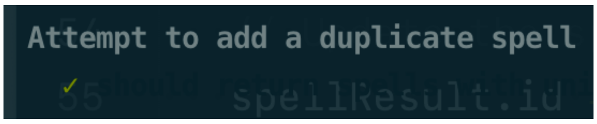

```
Add a spell with the right number of fields
  ✓
```

(b) *Pass*

Figure 9. *Test output for test set 3, test 3*


```
1) Add a new spell
     Attempt to add a duplicate spell
       should return spells with unique IDs:

    Uncaught AssertionError: expected 5 to equal 6
    + expected - actual

    -5
    +6
```

(a) *Fail*


```
Attempt to add a duplicate spell
  ✓
```

(b) *Pass*

3.1 Validating input files

As described in [1], all inputs to a program should be validated, especially when loaded from an external source or when the input is dynamic. In the Merklerex application, an external CSV file is loaded. The problem is that the CSV file contents are not checked, and assumptions about its format are made in the code. The code could easily inject undesirable code or data into the application at runtime. It can also cause the program to crash just by providing an unreasonably large file. To solve this problem, we can employ the following strategy.

- Use assertions to validate the size and format of the incoming CSV file.
- In the `CSVReader` class, check the file name and descriptors to ensure that the file is a CSV file. Some of this may already be provided by the CSV functions used in the program.

- When reading the CSV file lines, use **regular expressions** to validate that the file contents are of the correct format.
- Set limits to the file's possible size and check for this file as the file is loaded and as lines of the CSV are read. For example, ensure that the file does not get loaded if it exceeds 100MB, and stop loading lines after the first 10.000 lines have been read in.
- If such a scenario is encountered, use **exception handling** to treat those cases and degrade gracefully. For example, continue executing with fewer data and inform the user of this constraint.

All of these changes can happen in the `readCSV` function of this class.

3.2 Minimize access to data

According to [1], applications should minimize accessible data. The `OrderBook` class of the application exposes a map of `orders` and a set of `products` that are exposed as public data members. The problem with this is that any module can directly access these data members and manipulate them. It also directly exposes the type and implementation details of these two data members. No validation is done when changing any data within these members. A strategy for remediation might include:

- Creating getter and setter functions for these data members and moving them to the private members of the `OrderBook` class.
- Incorporating validation for the setter functions to ensure that any data changes are legal. For example, if a product needs to be added to the list of products, ensure that it has the correct format, such as no numbers in the currency name, and at least one forward character to separate a currency pair such as "BTC/USD".

These changes require analyzing which functions in the applications access these data members and determine what kind of validation is needed. The benefit of doing this includes exposing which parts of the application change data vs. just reading from it, allowing for better control of module coupling.

3.3 Prevent buffer overflows

The application makes certain assumptions about the size and sanity of variables. For example, numerous variables are declared of type `int` where it is assumed that the application will not encounter more than a certain number of orders in the source data. These variables could run into an overflow situation and cause serious errors or allow exploitation. A strategy to mitigate these would be:

- Create assertions to check that variables stay within certain ranges.
- Use more flexible data types such as `auto` for unknown size variables before runtime.
- Implement exception handling so that the program can continue with a reduced set of data in the case of an overflow situation.

References

[1] D. Wheeler, “Secure Programming HOWTO,” 2015. [Online]. Available: https://dwheeler.com/secure-programs/Secure-Programs-HOWTO/

Listing 1 Functional cohesion in Wallet class

```cpp
class Wallet
{
public:
        Wallet();

        /** Insert currency to the wallet. */
        void insertCurrency(const std::string& type, double amount);

        /** Remove currency from the wallet. */
        bool removeCurrency(const std::string& type, double amount);

        /** Check if the order can be fulfilled */
        bool canFulfillOrder(const OrderBookEntry& order);

        /** Check if the wallet contains this much currency or more. */
        bool containsCurrency(const std::string& type, const double amount = 0);

        /** Process the sale */
        void processSale(OrderBookEntry& sale);

        /** Generate a string representation of the wallet. */
        std::string toString();

        /** Get the balance of a specific currency */
        double checkBalance(const std::string& currency);

        /** Retrieve the currencies present in the wallet */
        std::map<std::string, double> getCurrencies();

        /** Get the value of the wallet in USDT */
        double getWalletValue(OrderBook& orderBook, std::string currentTime);

private:
        std::map<std::string, double> currencies;
        friend std::ostream& operator<<(std::ostream& os, Wallet& wallet);
};
```

Listing 2 Procedrual cohesion in the Merklebot class

```cpp
double MerkleBot::simpleMovingAverage(const std::string& product, const int&
↪  timeframes, const OrderBookType& type)
{
        int observations{0};
        double sumObservations{0};
        std::string timeframe{currentTime};
        while (orderBook.orders.find(timeframe) != orderBook.orders.begin() &&
        ↪  observations < timeframes)
        {
                std::vector<OrderBookEntry> orders{orderBook.getOrders(type,
                ↪  product, timeframe)};
                if (orders.size() > 0)
                {
                        sumObservations += orders.back().price;
                        timeframe = orderBook.getPreviousTime(timeframe);
                        ++observations;
                }
                else
                {
                        timeframe = orderBook.getPreviousTime(timeframe);
                }
        }
        return sumObservations / timeframes;
}

//////////////

// R3A: Bot generates a log of the contents of its wallet
void MerkleBot::generateWalletLog()
{
        std::ofstream walletLog;
        walletLog.open("walletLog.txt");
        walletLog << wallet << std::endl;
        walletLog.close();
}
```

Listing 3 Procedrual cohesion in the Merklebot class

```cpp
// Calling the CSVReader class for data and processing it
OrderBook::OrderBook(const std::string& filename)
{
        std::cout << "Loading market data..." << std::endl;
        orders = CSVReader::readCSV(filename);
        std::cout << "Read " << orders.size() << " timestamps." << std::endl;
        products = getKnownProducts();
}

// Creating the data and sending back OrderBook entries
std::map<std::string, std::vector<OrderBookEntry>> CSVReader::readCSV(std::string
↪   csvFileName)
{
        std::map<std::string, std::vector<OrderBookEntry>> entries;
        std::ifstream csvFile{csvFileName};

        if (csvFile.is_open())
        {
                std::string line;
                while (std::getline(csvFile, line))
                {
                        OrderBookEntry obe{stringsToOBE(tokenise(line, ','))};
                        entries[obe.timestamp].push_back(obe);
                }
        }
        else
        {
                std::cout << "Problem opening file " << csvFileName << std::endl;
        }
        csvFile.close();
        return entries;
}
```

Listing 4 Content coupling in the MerkleMain and MerkleBot classes

```cpp
// MerkleBot operating on the wallet data
void MerkleBot::executeTrades()
{
        for (const std::string& product : orderBook.products)
        {
                std::vector<OrderBookEntry> sales =
                ↪  orderBook.matchAsksToBids(product, currentTime);

                for (OrderBookEntry& sale : sales)
                {
                        if (sale.username == "simuser" &&
                        ↪  wallet.canFulfillOrder(sale))
                        {
                                wallet.processSale(sale);
                                generateBidsAsksSalesLog(sale);
                        }
                }
        }
}


// MerkleMain operating on the wallet data
void MerkleMain::init()
{
        currentTime = orderBook.orders.begin()->first;
        short int input;
        wallet.insertCurrency("BTC", 10);
        wallet.insertCurrency("ETH", 100);
        wallet.insertCurrency("DOGE", 1000);

        // ...
}
```

Listing 5 Test Set 1, Test 1

```
// The /spells route should be reached successfully
describe("Get everything at /spells", () => {
        it("should return successfully", (done) => {
                chai.request(app)
                        .get("/spells")
                        .end((err, res) => {
                                assert.equal(res.status, 200);
                                done();
                        });
        });
});
```

Listing 6 Test Set 1, Test 2

```
describe("Ensure spells are spells with correct length", () => {
        it("should return spells with proper length", (done) => {
                chai.request(app)
                        .get("/spells")
                        .end((err, res) => {
                                assert.typeOf(res.body, 'array');
                                assert.isAtLeast(res.body.length, 1);
                                for (var i = 0; i < res.body.length; i++) {
                                        assert.lengthOf(Object.keys(res.body[i]), 4);
                                }
                                done();
                        });
        });
});
```

Listing 7 Test Set 1, Test 3

```
describe("Get a specific spell", () => {
        // Configure a specific spell to test for
        let testId = "1001";
        it("should return the correct spell that was requested", (done) => {
                chai.request(app)
                        .get("/spells/" + testId)
                        .end((err, res) => {
                                assert.equal(res.body.id, testId);
                                done();
                        });
        });
});
```

Listing 8 Test Set 2, Test 1

```
// The /spells route should be reached successfully
describe("Send an update request successfully", () => {
        let testId = "1003";
        it("should update a spell", (done) => {
                chai.request("http://localhost:3000")
                        .put("/spells/" + testId)
                        .end((err, res) => {
                                assert.equal(res.status, 200);
                                done();
                        });
        });
});
```

Listing 9 Test Set 2, Test 2

```
describe("Update the name of a spell", () => {
        let testId = 1003;
        let newName = "Hackus maximus";
        it("should update a single attribute of a spell", (done) => {
                chai.request("http://localhost:3000")
                        .put("/spells/" + testId)
                        .send({ name: newName })
                        .end((err, res) => {
                                assert.equal(res.body.name, newName);
                                done();
                        });
        });
});
```

Listing 10 Test Set 2, Test 3

```
describe("Update multiple fields of a spell", () => {
        let testId = 1003;
        let updatedName = "Hackus maximus";
        let updatedIngredients = [
                { name: "Stroustrup hair" },
                { name: "Uncle Bob sweat" },
        ];
        let updatedResult = "Clean coding";
        let updatedSpell = {
                id: testId,
                name: updatedName,
                ingredients: updatedIngredients,
                result: updatedResult,
        };
        it("should update multiple fields of a spell", (done) => {
                chai.request("http://localhost:3000")
                        .put("/spells/" + testId)
                        .send(updatedSpell)
                        .end((err, res) => {
                                assert.equal(
                                        JSON.stringify(res.body),
                                        JSON.stringify(updatedSpell)
                                );
                                done();
                        });
        });
});
```

Listing 11 Test Set 3, Test 1

```
// The /spells route should be reached successfully
describe("Add a basic spell with an ID", () => {
        let newId = 1004;
        it("should add a spell with an ID", (done) => {
                chai.request("http://localhost:3000")
                        .post("/spells/")
                        .send({ id: newId })
                        .end((err, res) => {
                                assert.equal(res.status, 200);
                                assert.equal(res.body.id, newId);
                                done();
                        });
        });
});
```

Listing 12 Test Set 3, Test 2

```
describe("Add a spell with the right number of fields", () => {
        let newSpell = {
                id: 1005,
                name: "Wingardium Leviosa",
                ingredients: [{ name: "Bird blood" }, { name: "Feather essence" }],
                result: "Dream of Flight",
        };
        it("should add all fields to the spell correctly", (done) => {
                chai.request("http://localhost:3000")
                        .post("/spells/")
                        .send(newSpell)
                        .end((err, res) => {
                                assert.equal(
                                        JSON.stringify(res.body),
                                        JSON.stringify(newSpell)
                                );
                                assert.equal(Object.keys(res.body).length, 4);
                                done();
                        });
        });
});
```

Listing 13 Test Set 3, Test 3

```
describe("Attempt to add a duplicate spell", () => {
        // Create a duplicate spell object
        let newSpell = {
                id: 1001,
                name: "Rabbit foot positivity",
                ingredients: [
                        { name: "Foot of rabbit" },
                        { name: "Juice of beetle" },
                ],
                result: "Good luck",
        };
        it("should return spells with unique IDs", (done) => {
                // Attempt to create the spell
                chai.request("http://localhost:3000")
                        .post("/spells/")
                        .send(newSpell)
                        .end(() => {
                                // Get all spells
                                chai.request("http://localhost:3000")
                                        .get("/spells/")
                                        .end((err, res) => {
                                                // Cast the list of spells as a set
                                                // to get unique spells
                                                let spellSet = new Set(res.body);
                                                assert.equal(
                                                        spellSet.size,
                                                        res.body.length
                                                        );
                                                done();
                                        });
                        });
        });
});
```