CONTENT

INTRODUCTION……………………………………………………………...3

1. ARCHITECTURE AND TECHNNOLOGIES OF THE TEST PROGRAM………………………………………………………………………...4
2. REQUIRED DATA IN THE FORM OF MODELS FOR TESTING COLLECTIONS OF CARDS……………………………………………………..11
   1. Моdels in the directory card/catalog………………………………………..11
   2. Models in the directory card/client………………………………………….13
3. COLLECTION OF TESTS…………………………………………………15
   1. Module tests……………………………………………………………….15
      1. RedundantTrieTest…………………………………………………………15
      2. TestResourceGenerator…………………………………………………….15
      3. MtgTestApplicationTests…………………………………………………..16
   2. Integration tests………………………………………………………..........16
      1. MagicTheGatheringCardCatalogIntegrationTest…………………………..16
      2. MagicTheGatheringCardClientIntegrationTest…………………………….18
      3. CardSearchTrieIntegrationTest…………………………………………….19

CONCLUSION…………………………………………………………….20

LIST OF SOURCES USED………………………………………………..21

**INTRODUCTION**

[REST-Assured](https://rest-assured.io/) is a Java library that provides a domain-specific language (DSL) for writing powerful, maintainable tests for RESTful APIs. One thing I really like about rest-assured is its BDD style of writing tests and one can read the tests very comfortably like what’s going on inside and how the tests will proceed.

The purpose of this work is to construct a test module assuming that:

1. API documentation is available at: <https://docs.magicthegathering.io/>.
2. All available endpoints will be covered by tests.
3. Tests will include positive scenarios (checking logic, content returned) and negative scenarios (checking error codes like 404).
4. The tests will use parameters (pathparam, param).
5. Test classes will be logically separated.
6. Test descriptions/names will make sense so you know what a particular test should do.

To research and build a test module, we will use the following APIs - <https://api.magicthegathering.io/v1/cards>

1. **ARCHITECTURE AND TECHNOLOGIES OF TEST PROGRAM**

This program contains a reactive client to the [REST APIs](https://magicthegathering.io/) serving up Magic: the Gathering card information. The intent is to allow consumers to retrieve card data in as a Java object, and to allow them to read pages of data rather than relying on a synchronous process that tries to retrieve all ~10k cards at once.

Planned enhancements to the client:

1. Retrieval of card by name or gatherer ID.
2. More card information, e.g. links to images.
3. Retrieval of card by format legality, e.g. Standard, Modern, Commander.

The ultimate goal of this client is to have it consumed by a REST service that can serve processing for web applications.

This program consists of models of different cards in the catalog and operations with them by client, configuration and searching and, of course, some different type of tests, such as ordinary, integration and unit.

And now, we can represent that main dependencies using this program.

<dependencies>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-webflux</artifactId>  
 <scope>provided</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.immutables</groupId>  
 <artifactId>value</artifactId>  
 <version>2.7.1</version>  
 <scope>compile</scope>  
 </dependency>  
 <dependency>  
 <groupId>com.fasterxml.jackson.datatype</groupId>  
 <artifactId>jackson-datatype-guava</artifactId>  
 <version>2.7.3</version>  
 <scope>compile</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.apache.commons</groupId>  
 <artifactId>commons-collections4</artifactId>  
 <version>4.3</version>  
 <scope>compile</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-test</artifactId>  
 <scope>test</scope>  
 <exclusions>  
 <exclusion>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 </exclusion>  
 </exclusions>  
 </dependency>  
 <dependency>  
 <groupId>io.projectreactor</groupId>  
 <artifactId>reactor-test</artifactId>  
 <version>3.2.10.RELEASE</version>  
 <scope>test</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.junit.jupiter</groupId>  
 <artifactId>junit-jupiter-api</artifactId>  
 <version>5.2.0</version>  
 <scope>test</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.junit.jupiter</groupId>  
 <artifactId>junit-jupiter-params</artifactId>  
 <version>5.2.0</version>  
 <scope>test</scope>  
 </dependency>

<dependency>  
 <groupId>io.rest-assured</groupId>  
 <artifactId>rest-assured</artifactId>  
 <version>3.0.7</version>  
 <scope>test</scope>  
</dependency>  
 <dependency>  
 <groupId>net.bytebuddy</groupId>  
 <artifactId>byte-buddy</artifactId>  
 <version>1.9.12</version>  
 <scope>test</scope>  
 </dependency>  
</dependencies>

For build test module we will use libraries: junit, rest-assured framework, mockito, bytebuddy, spring webflux, jackson, immutables.

Consider junit.

JUnit is a [unit testing](https://en.wikipedia.org/wiki/Unit_testing) [framework](https://en.wikipedia.org/wiki/Software_framework) for the [Java programming language](https://en.wikipedia.org/wiki/Java_(programming_language)). JUnit has been important in the development of [test-driven development](https://en.wikipedia.org/wiki/Test-driven_development), and is one of a family of [unit testing](https://en.wikipedia.org/wiki/Unit_testing) frameworks which is collectively known as [xUnit](https://en.wikipedia.org/wiki/XUnit) that originated with [SUnit](https://en.wikipedia.org/wiki/SUnit).

JUnit is linked as a [JAR](https://en.wikipedia.org/wiki/JAR_(file_format)) at compile-time. The latest version of the framework, JUnit 5, resides under package org.junit.jupiter [10].Previous versions JUnit 4[10] and JUnit 3 were under packages org.junit and junit.framework, respectively.

Every JUnit test class usually has several test cases. These test cases are subject to the test life cycle. The full JUnit Life Cycle has three major phases [15]:

1. Setup.

This phase is where the test infrastructure is prepared. Two levels of setup are available. The first type of setup is class-level setup in which a computationally expensive object, such as a database connection, is created and reused, with minimal side effects. Class-level setup is implemented using the @BeforeAll annotation. The other type is setup before running each test case, which uses the @BeforeEach annotation [2].

1. Test execution.

This phase is responsible for running the test and verifying the result. The test result will indicate if the test result is a success or a failure. The @Test annotation is used here [2].

1. Clean up.

After all posttest executions are performed, the system may need to perform cleanup. Similar to class-level setup, there is a corresponding class-level clean up. The @AfterAll annotation is used to support class-level clean up. The @AfterEach annotation allows for cleanup after test execution [2].

JUnit follows the paradigm of preferring extension points over features [7]. The JUnit team decided not to put all features within the JUnit core, and instead decided to give an extensible way for developers to address their concerns [7]. In JUnit 4, this was achieved using the Runner API and Rule API [11]. In JUnit 5, the extension API is found within the JUnit Jupiter Engine [12]. The JUnit Team wants to grant the developer the power to hook to separate stages of a test life cycle [12]. In particular, this occurs when the Jupiter Engine is able to invoke all registered extension when a certain life cycle phase is reached [12]. The developer can hook into five major extension points [12]:

1. Test life cycle callbacks - This allows the developer to hook to certain phases of a test life cycle [13].
2. Test instance post-processing - this enables the developer to hook after test instance creation by implementing the TestInstancePostProcessor interface [13].
3. Conditional test execution - this enables the developer to execute the test case only after meeting certain criteria [8].
4. Parameter resolution - This enables the developer to resolve a parameter after receiving it from a test method or constructor.
5. Exception handling - A use case for Exception handling is to change the testing behavior instead of throwing an exception [9].

A JUnit [test fixture](https://en.wikipedia.org/wiki/Test_fixture#Software) is a Java object. Test methods must be annotated by the @Test [annotation](https://en.wikipedia.org/wiki/Java_annotation). If the situation requires it [10], it is also possible to define a method to execute before (or after) each (or all) of the test methods with the @BeforeEach (or @AfterEach) and @BeforeAll (or @AfterAll) annotations [4].

Consider mockito.

Mockito is an [open source](https://en.wikipedia.org/wiki/Open_source) [testing framework](https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks) for [Java](https://en.wikipedia.org/wiki/Java_(software_platform)) released under the [MIT License](https://en.wikipedia.org/wiki/MIT_License) [2,3]. The framework allows the creation of [test double](https://en.wikipedia.org/wiki/Test_double) objects ([mock objects](https://en.wikipedia.org/wiki/Mock_object)) in [automated unit tests](https://en.wikipedia.org/wiki/Test_automation) for the purpose of [test-driven development](https://en.wikipedia.org/wiki/Test-driven_development) (TDD) or [behavior-driven development](https://en.wikipedia.org/wiki/Behavior-driven_development) (BDD).

Mockito allows developers to verify the behavior of the [system under test](https://en.wikipedia.org/wiki/System_under_test) (SUT) without establishing expectations beforehand [1]. One of the criticisms of [mock objects](https://en.wikipedia.org/wiki/Mock_Object) is that there is a tight coupling of the test code to the system under test [6]. Mockito attempts to eliminate the expect-run-verify pattern [5] by removing the specification of expectations. Mockito also provides some annotations for reducing [boilerplate code](https://en.wikipedia.org/wiki/Boilerplate_code) [17].

Consider Rest-Assured.

REST-assured was designed to simplify the testing and validation of REST APIs and is highly influenced by testing techniques used in dynamic languages such as Ruby and Groovy.

The library has solid support for HTTP, starting of course with the verbs and standard HTTP operations, but also going well beyond these basics.

Consider ByteBuddy.

ByteBuddy is a Java software library for the manipulation and creation of Java bytecode. Bytecode manipulation can be performed prior to execution or during the runtime of an application. Byte Buddy uses the ObjectWeb ASM library to apply its manipulations but provides a high-level API that avoids the explicit specification of Java bytecode that is required by ObjectWeb ASM. Instead, Byte Buddy uses a domain-specific language to express bytecode manipulation on a high level where the domain language’s verbs lean onto terms of the Java programming language. Byte Buddy is used by a wide-range of commercial and free software products on the Java virtual machine. For example, the library is used by the popular Mockito and Hibernate libraries. Byte Buddy is in particularly popular for creating Java agents, for example by Instana APM Elastic APM, where code changes are implemented in regular Java code which is used as template for changing a class’s code. Byte Buddy is part of the OpenJDK Quality Outreachfor giving early feedback on upcoming Java releases. The library received a Duke's Choice Award in 2015 and its author was distinguished with an Oracle Groundbreaker Award in 2019.

Consider spring webflux.

Spring WebFlux allows us to decompose the logic in a declarative way with Mono, Flux, and their rich operator sets. Moreover, we can have functional endpoints besides its @Controller annotated ones, though we can now also use these in Spring MVC.

Consider jackson.

In [computing](https://en.wikipedia.org/wiki/Computing), jackson is a high-performance [JSON](https://en.wikipedia.org/wiki/JSON) processor for [Java](https://en.wikipedia.org/wiki/Java_(programming_language)). Its developers extol the combination of fast, correct, lightweight, and ergonomic attributes of the library. Jackson provides multiple approaches to working with JSON, including using binding annotations on [POJO](https://en.wikipedia.org/wiki/Plain_Old_Java_Object) classes for simple use cases.

Consider Immutables.

The library consists of annotations and annotation processors for generating and working with serializable and customizable immutable objects.

The library generates immutable objects from abstract types:Interface*,*Class*,*Annotation*.*

The key to achieving this is the proper use of @Value.Immutableannotation. **It generates an immutable version of an annotated type and prefixes its name with the**Immutable**keyword**.

1. **REQUIRED DATA IN THE FORM OF MODELS FOR TESTING COLLECTIONS OF CARDS**
   1. **Models in the directory card/catalog.**
2. Interfaces:
3. Card.

@Immutable  
@JsonDeserialize(as = ImmutableCard.class)

public interface Card {  
  
 String name();  
 Set<Color> colorIdentity();  
 int convertedManaCost();  
}

1. CardCatalog.

public interface CardCatalog {  
  
 Flux<Card> getAllCards();  
 Flux<Card> matchCards(CardCriteria criteria);  
}

1. CardCriteria.

@Immutable

public interface CardCriteria {  
  
 default boolean exclusiveMatch() {  
 return true;  
 }  
  
 Optional<String> nameContains();  
 Optional<Set<Color>> colorIdentity();  
}

1. Enums.
2. Color.

public enum Color {  
 GREEN("G"), RED("R"), BLUE("U"), BLACK("B"), WHITE("W"), COLORLESS(null);  
  
 class lol {  
 }  
  
 private String abbreviation;  
  
 public static Color parse(String representation) {  
 return EnumSet.allOf(Color.class).stream()  
 .filter(color -> color.name().startsWith(representation) || color.matchesAbbreviation(representation))  
 .findFirst()  
 .orElseThrow(() -> new NoSuchElementException(format("No Color matching %s found.", representation)));  
 }  
  
 private boolean matchesAbbreviation(String representation) {  
 switch (this) {  
 case COLORLESS:  
 return false;  
 default:  
 return this.abbreviation.equals(representation);  
 }  
 }  
  
 Color(String abbreviation) {  
 this.abbreviation = abbreviation;  
 }  
}

1. Rarity.

public enum Rarity {  
 COMMON, UNCOMMON, RARE, MYTHIC\_RARE  
}

1. Type.

public enum Type {  
 *LAND*, *CREATURE*, *ENCHANTMENT*, *INSTANT*, *SORCERY*}

* 1. **Models in the directory card/client.**

Interfaces:

1. Page.

@Immutable  
public interface Page {  
  
 List<RawCard> cards();  
 Optional<Integer> nextPageNumber();  
 Integer lastPageNumber();  
}

1. RawCard.

@Immutable  
@JsonDeserialize(as = ImmutableRawCard.class)  
public interface RawCard {  
  
 String name();  
 String rarity();  
 double cmc();  
  
 Optional<String> text();  
 Optional<String> manaCost();  
 Set<String> colorIdentity();  
 Set<String> colors();  
 Set<String> types();  
}

1. RawCards.

@Immutable(builder = false)  
@JsonDeserialize(as = ImmutableRawCards.class)  
public interface RawCards {  
  
 @Parameter  
 List<RawCard> cards();  
}

1. CardsClient.

public interface CardsClient {  
  
 @Cacheable(value = "cardPages", cacheManager = "cardPagesCacheManager")  
 Mono<Page> getPage(int index);  
 Mono<Integer> getLastPageNumber();  
}

1. **COLLECTION OF TESTS**
   1. **Ordinary tests.**
      1. **RedundantTrieTest.**
2. GetRequiresMatch.

@Test  
 void get\_requiresExactMatch() {  
 final Trie<String> trie = Trie.*withKeyMapping*(Function.*identity*());  
 trie.add("abcd");  
 trie.search("bbcd", 2);

}

1. GetMultipleBranchesRequiresExactMatch.

@Test  
void get\_multipleBranches\_requiresExactMatch() {  
 final Trie<String> trie = Trie.*withKeyMapping*(Function.*identity*());  
 trie.add("abc");  
 trie.add("ax");

}

* + 1. **TestResourceGenerator.**

@Test  
 void saveOneThousandCardsAsJsonResource() {  
// final String json = cardSerializer.write(cardCatalog.getAllCards().take(1000).collect(Collectors.toList()).block());  
 }

* + 1. **MtgTestApplicationTests.**

@Test  
void contextLoads() {  
}

* 1. **Integration tests.**
     1. **MagicTheGatheringCardCatalogIntegrationTest.**

1. PrintCards.

@Test  
void printCards() {  
 cardCatalog.getAllCards().take(n).doOnEach(*out*::println).blockLast();  
}

1. CountCards.

@Test  
void countCards() {  
 *out*.println(  
 *format*("Counted %d Magic: The Gathering cards (taking n: %d)", cardCatalog.getAllCards().take(n).count().block(), n)  
 );  
}

1. PrintCardsWithGreenColorIdentity.

@Test  
void printCardsWithGreenColorIdentity() {  
 cardCatalog.matchCards(  
 ImmutableCardCriteria.*builder*()  
 .colorIdentity(Set.*of*(Color.*GREEN*))  
 .build()  
 ).take(n / 4).doOnEach(*out*::println).blockLast();  
}

1. PrintUniqueCard.

@Test  
void printUniqueCard() {  
 cardCatalog.matchCards(  
 ImmutableCardCriteria.*builder*()  
 .nameContains("Nekrataal")  
 .build()).take(1).doOnEach(System.*out*::println).blockLast();  
}

1. PrintCardsWithTheInName.

@Test  
void printCardsWithTheInName() {  
 cardCatalog.matchCards(  
 ImmutableCardCriteria.*builder*()  
 .nameContains("the")  
 .build()).take(n / 4).doOnEach(System.*out*::println).blockLast();  
}

* + 1. **MagicTheGatheringCardClientIntegrationTest.**

1. GetLastPageNumber.

@Test  
void getLastPageNumber() {  
 *assertThat*(cardsClient.getLastPageNumber().block()).isEqualTo(cardsClient.getPage(1).block().lastPageNumber());  
}

1. PrintCardsOnFirstPage.

@Test  
void printCardsOnFirstPage() {  
 *printCardsOnPage*(cardsClient.getPage(1));  
}

1. PrintCardsOnSeondPage.

@Test  
void printCardsOnSecondPage() {  
 *printCardsOnPage*(cardsClient.getPage(2));  
}

1. PrintCardsOnLastPage.

@Test  
void printCardsOnLastPage() {  
 *printCardsOnPage*(cardsClient.getPage(cardsClient.getLastPageNumber().block()));  
}

* + 1. **CardSearchTrieIntegrationTest.**

1. BuildAndSearch.

@Test  
void buildAndSearch() {  
 final Trie<Card> trie = Trie.*withKeyMapping*(Card::name);  
 cardCatalog.getAllCards().take(400)  
 .doOnNext(System.*out*::println)  
 .doOnNext(trie::add).blockLast();

}

1. GetMeNames.

@Test  
void getMeNames() {  
 System.*out*.println(  
 cardCatalog.getAllCards().take(400).map(Card::name).collect(Collectors.*toList*()).block().stream().collect(Collectors.*joining*("\\\",\\\""))  
 );  
}

**CONCLUSION**

In this work we build program for testing api: <https://api.magicthegathering.io/v1/cards>, which includes many different, integration and module tests with rest-assured. These tests run independently from each other and and give some result to users about specific checks/errors.

**LIST OF SOURCES USED**

1. ["Features and Motivations"](https://code.google.com/p/mockito/wiki/FeaturesAndMotivations). Retrieved 2010-12-29.
2. ["Mockito in six easy examples"](http://gojko.net/2009/10/23/mockito-in-six-easy-examples/). Retrieved 2012-10-05, 2009.
3. ["What's the best mock framework for Java?"](https://stackoverflow.com/questions/22697/whats-the-best-mock-framework-for-java). Retrieved 2010-12-29.
4. [*"Writing Tests"*](https://junit.org/junit5/docs/current/user-guide/#writing-tests). junit.org*. Retrieved 2021-02-04*.
5. Faber, Szczepan. ["Death Wish"](https://web.archive.org/web/2010/http:/monkeyisland.pl/2008/02/01/deathwish/). Retrieved 2010-12-29.
6. Fowler, Martin. ["Mocks Aren't Stubs"](http://martinfowler.com/articles/mocksArentStubs.html#CouplingTestsToImplementations). Retrieved 2010-12-29.
7. Gulati & Sharma, p. 121, Chapter §7 JUnit 5 Extension Model, 2017
8. [Gulati & Sharma](https://en.wikipedia.org/wiki/JUnit#CITEREFGulatiSharma2017), p. 127, Chapter §7 JUnit 5 Extension Model - Conditional Test Execution.
9. [Gulati & Sharma](https://en.wikipedia.org/wiki/JUnit#CITEREFGulatiSharma2017), p. 129, Chapter §7 JUnit 5 Extension Model - Exception Handling.
10. Gulati & Sharma, p. 144, §Chapter 8 Dynamic Tests and Migration from Junit 4, 2017.
11. [Gulati & Sharma](https://en.wikipedia.org/wiki/JUnit#CITEREFGulatiSharma2017), pp. 121–122, Chapter §7 JUnit 4 Extension Model, 2017
12. [Gulati & Sharma](https://en.wikipedia.org/wiki/JUnit#CITEREFGulatiSharma2017), pp. 122–124, Chapter §7 JUnit 5 Extension Model - JUnit 5 Extension Model.
13. [Gulati & Sharma](https://en.wikipedia.org/wiki/JUnit#CITEREFGulatiSharma2017), pp. 124–126, Chapter §7 JUnit 5 Extension Model - Test Life Cycle Callbacks.
14. [Gulati & Sharma](https://en.wikipedia.org/wiki/JUnit#CITEREFGulatiSharma2017), pp. 126–127, Chapter §7 JUnit 5 Extension Model - Test Instance Post-Processing.
15. Gulati & Sharma, pp. 37–40, Chapter §2 JUnit LifeCycle API, 2017.
16. Kaczanowski, Tomek. ["Mockito - Open Source Java Mocking Framework"](http://www.methodsandtools.com/tools/mockito.php). Retrieved 2013-09-17.
17. [Kent Beck](https://en.wikipedia.org/wiki/Kent_Beck). ["Expensive Setup Smell"](http://c2.com/cgi/wiki?ExpensiveSetUpSmell). C2 Wiki. Retrieved 2011-11-28.