Final Project: Spades

# Introduction

Project: Implement a two-player version of Spades

Project members: Derek Luong and Jamie Bowers

Class: SWE 681- 001

Date: 2019-04-27

# Design/Architecture

The overall architecture of the application is developed using the Java Spring framework. Spring Boot is used to provide a base application, as well as means to provide hooks into the rest of the application. Spring Security is used as a library to provide security, rather than attempt to implement our own version. Spring Data’s Java Persistence API (JPA), which uses Hibernate, is utilized to provide database access to the application through object relational mapping.

## Database Components

All classes in the **model** directory are annotated using @Entity. These classes are essentially used to represent information that is stored in the Database, and follows the general guidelines for creating JPA Entity classes. However, the CustomUserDetails class is the one exception that does more than just the standard JPA Entity implementation. This is used to provide a means for providing an interface for retrieving user information from the database, which is then used by Spring Security to provide validation.

Classes in the **repository**directory are all extensions of Spring Data’s JpaRepository class. These classes are used by the application to facilitate access to the underlying database. There are also several additional query methods, which either rely on Spring Data to generate the proper queries using the method name, or use the @Query annotation provided by Spring Data to define custom queries.

## Controller classes

In the **resources** directory, there are several classes that use the @RestController and @RequestMapping annotations. These annotations are used to allow the Spring framework when starting up to expose endpoints in the application that can be accessed through a Web browser. The HelloResource class serves as the initial starting point for most requests, providing links to other components for the application. Apart from the intial page, which shows links to login or create user, and createuserpage.html (which is in resources/static), all other endpoints exposed by controllers will require user authorization, and will require user login if the user attempts to access them.

The other controller classes are essentially used to implement specific aspects of the application. For example, the CreateGameController class handles the creation of new games, the RetrieveOpenGamesController retrieves games that still need a second player, etc. Other features provided through the controllers include viewing past games, user statistics, and exposing a point to allow users to actually play the game. The GeneralErrorController, which implements the ErrorController interface, is used to override the default error page implementation provided by the Spring framework. This allows us to log unexpected errors when they occur, and otherwise displays a standard message to the user, with a link back to the homepage.

## Service classes

Classes defined in the **services** directory are essentially core processing classes. These are essentially used to implement features that need to be reused by multiple components (such as the Controllers), or are used to offload logic from the controllers.

For example, the GetCurrentPlayerInfoService is used by multiple components to retrieve information about the current user, using the underlying Spring framework. The GameTimerService is used to start a timer when creating a game, which is then used by the ActiveGameController to ensure that users aren’t blocked waiting for other players, and deletes the game to allow the user to retry or join a different game.

The SpadesGameService is only used by the ActiveGameController, and serves as an example of moving logic out of the controller. This service handles the complexity of passing the user input to the underlying representation of the user’s current game (discussed later) and managing the database as the game progresses. It also contains several utility functions that perform certain useful operations related to Spades.

## Implementing Spades

The SpadesRoundImpl class is used to hold information about a current round between two players. This uses the Deck class to represent the 52 standard cards, and maintains 2 Hand objects to indicate which cards are held by each player. Other information held by this class is the current turn, cards played in the current trick, and an internal HandTimeOut used to detect if players took too long to make a move. The SpadesRoundImpl class provides methods to accept String input, allowing the user to play a card, which is then validated based on the current state of the game in order to ensure that this is a valid move. This class also provides a method, which will calculate the current trick provided that all players have played a card.

The SpadesRoundImpl is combined with the SpadesGameService to implement a complete Spades game. The SpadesGameService provides several utility functions to perform scoring calculations or render completed round information, which are used in other components besides this one. The SpadesGameService will accept bidding or card inputs from the user, which will perform validation based on the current state of the game before storing this information in the database or passing it to the SpadesRoundImpl for processing. This class handles the management of multiple games, creating new rounds as needed if a game isn’t complete yet, as well as recording information in the database. This will also retrieve the current state of the game for players.

The ActiveGameController serves as the endpoint for all Spades game. It will display information about the game to users, and also provides endpoints to accept input from the users. If the input passes the initial validation performed by the controller, then it is passed to the SpadesGameService for further processing, which may or may not be accepted depending on the current state of the game.

## JavaScript reloading

Finally, we have a reloadPage.js file that checks the form fields for the bidding and card playing forms. Essentially it sets a timeout of seven seconds and if the user has not entered a bid or played a card the page reloads. This is how we are able to reload page to determine who’s turn it is and what cards have been played for that trick.

# Installation Instructions

## External Dependencies

The following dependencies are required in order to install and play the spades game. This paper assumes the user knows how to install, configure, and run these dependencies.

* PostgreSQL 11
* Any Java IDE that supports Maven 3.5.2 (We used IntelliJ) with the 10.0.2 JDK
* JDK 10.0.2
* Java 1.8
* Two Separate Internet Browsers (We tested against Firefox and Google Chrome) or two separate computers.

## Initial Setup

In order to run the Spades game, the user must set up the initial database. This involves creating the necessary tables for both Spring Security as well as tables used to run and store information about Spades games.

Inside the Java project there is a folder called “sql\_scripts” and a file called “create\_database.sql”. To create the necessary tables, run the file “create\_database.sql” inside the Postgres’s pgAdmin tool using the Query tool.

Next in order for Spring to connect to the database and run you will need to modify your appilication.prooerties file, which is located in the spades/src/resources directory, to tell it what your database URL, username, and password are. You will need to modify lines 15-17 to reflect what your configurations should be.

# Operating Instructions

## Startup In order to start the game you must choose to run the application in your chosen Java IDE, your log upon start-up will show the Spring Boot text with the version of Spring Boot being 2.1.3 and you will see “Started SpadesAppilication” in the log, Once Spring has started you are ready to start the Spades application. We have implemented port forwarding in Spring so that if you simply type localhost in your browser it will send you to the correct https://localhost local website ensuring that you are using the TLS self-signed certificate that we created through the Java keystore.

## Create Users

The first step in order to play this game requires that two user accounts are created. A user can be created using the “Create User” link from the front page. Creating a user has four fields that are required; Username, Last Name, Email, and Password. We implemented a password whitelist that requires a password be at least eight characters, and must have the following characteristics; It must be use at least one lower case characters, one uppercase character, one numeric character and one of the following special characters “@#%$^” it also must not use whitespaces (tab or space).

## Create and Join a Game Once two or more players have been created, a game can be created. To do this simply click the “Create Game” button. Once a game has been created the users can join a game, the user that created the game is given a link to join the game and it will wait until the 2nd player has joined a game. For the 2nd player there is a “find games” link that they can click to find existing games that have been created. If Player 2 does not join the game in the five minutes that we set up for out game timeout the game is no longer active, and a new game must be created.

## Gameplay

Once both users have joined a game each player will get dealt a random hand and each player must bid the number of tricks they think they can win (see Game Rules). Once each player has bid the game begins and each player plays a card from the hand they are dealt with the objective being to win or not win the round based on their bid. Players may only play the cards they are dealt and only when it is their turn to play. The game keeps track of whose turn it is and tells each player whose turn it is and what card have been played by which player. Statistics on the page what each players Bids were and what their actual tricks won are, these statistics are shown and updated upon each winning trick. When a round is complete statistics showing each round, the bid for each player that round the number of tricks each player won for that round the number of bags they have in total and their current score. The game continues if a player has not won and shows your new hand and asks for a bid, this process continues until one of the two players has won the game.

# Game Rules

## Objective

Be the first player to reach 200 points. If both players reach 200 points in the same round the player with the higher number of points between the two wins.

## Setup

Each “Round” a 52 Card Deck is shuffled. Two players are each dealt thirteen cards from the deck randomly. This leaves 26 cards that are remaining in the deck, which will not be used in the current round. (13 cards per player \* 2 players + 26 cards left over = 52 Total Cards in Deck)

## Card Rank

Card Values are Ace (A), King (K), Queen (Q), Jack (J), 10, 9, 8, 7, 6, 5, 4, 3, and 2. Suits are Spades (S), Hearts (H), Clubs (C), and Diamonds (D). Spades beats every other suited card. E.g. 2S (2 of Spades) beats AH (Ace of Hearts). In the case of same card suits, the higher valued card will win, so 9H (9 of Hearts) beats 7H (7 of Hearts). If a Spades suit is not played but the played cards have different suits, then the leading player (the player that went first in the current trick) will win regardless of card value.

## Bidding

Each player will decide how many “Tricks” they think they can win in a round. They can also choose “nil” if they want to try to lose every trick that hand to try to get 100 points for that round (See Scoring) A “Trick” is simply a winning hand. Bidding occurs each round.

## Scoring

Each winning trick is worth 10 times the bid amount and every trick that is won that goes over the bid amount is referred to as a “Sandbag” or “bag” and is worth 1 point. **Example:** Player 1 bids 6 tricks and wins 8. They are awarded 62 points, 60 for their bid and 2 for their bag.

If a player fails to meet the number of tricks they bid the player is deducted the entire bid times 10 points for each bid. **Example:** Player 2 bids 8 tricks but only gets 6 tricks in the round, they are deducted 80 points.

Sandbags are kept track of for each player and once a player hits 10 bags or more they are deducted 100 points, and their bags are subtracted by 10.

A player can bid 0 (known as a NIL bid) if they believe they can lose every trick in order to get 100 points for the round. If they fail to do this and get even 1 trick they are deducted 100 points.

## Gameplay

Player 1 starts the game and plays a card, player 2 then follows. Spades are not allowed to be played unless they have already been “broken”. In order to “Break Spades” a player may play a spade if they do not have any cards of the suit that the opening player plays, or if the opening player only has cards of the Spades suit remaining. Once Spades is broken it can be opened with in any following round.

Example: Player 1 has 5S, 4C, 2D and 3D, player 2 has QH, 10D, 3S, and 4S. Player 1 plays 2D, player 2 must play a Diamond, in the next hand Player 1 Plays 3D because Player 2 no longer has any Diamonds they can play their 3 of Spades to win the trick, Spades are now broken and Player 1 can play their 5S in the next hand.

The winner of the trick starts the next hand. Each hand is played until the players are out of their initial 13 cards. The round is then scored and a new round is started. This process goes on until one of the players wins the game.

# Why This Application is Secure

Our application is secure because it takes steps to ensure that confidentiality, integrity, and availability are maintained. In addition, our application is secure because there is a low risk of being susceptible to the most common vulnerabilities. Several analysis tools have also been used on the application to look for potential issues (possibly security-related), and these issues have been addressed.

## Confidentiality

**Requirement:** “You must create an audit trail of every game move, and these game moves must not be available to those not in the game until the game has completed. After the game is completed, the final audit record (including every move) must be publicly available to all authenticated users (but to not to unauthenticated ones). The win/loss/draw record of each user must be made available to authenticated users (but not to unauthenticated ones).”

We have created an audit trail of every game move using a database table called “moves”. Each time a player has played a card it records the card they in the moves table. These moves will not be displayed to authenticated users until after the game is over. The ViewEndedGamesController, which handles displaying the results of a game, ensures that the game status has been marked as ended before displaying information about it to authenticated user. This information includes game moves as well as the winner, the rounds played during that game, and point totals. The code excerpt below shows this validation check, and calls a private method to render the information.

Optional<Games> game = gamesRepository.findByGameId(gameid);

if(game.isPresent())

{

Games g = game.get();

if(g.getGameStatus().equals("e"))

{

result.append(renderSpecificGame(g));

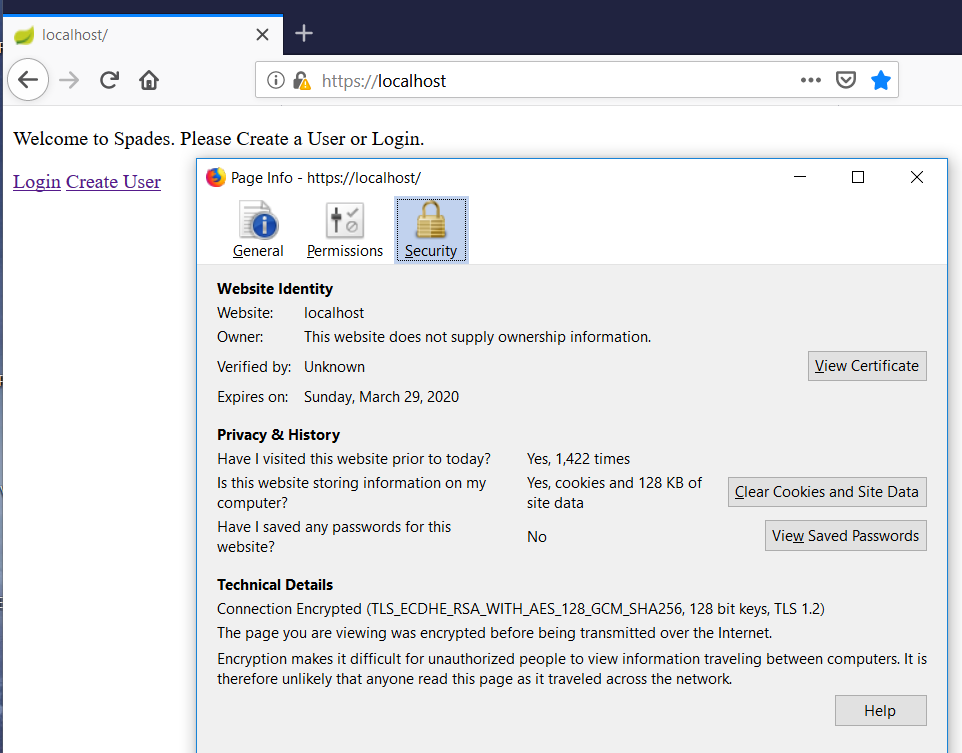
}

}

A win/loss statistics page is available when you are not playing a game, which shows wins and losses for every player and is only available to authenticated players. In the ViewWinLossStatsController, the @PreAuthorize annotation is used to ensure that users have the correct role. This is in addition to the security configuration that requires all access to this page to be authenticated.

### Encryption

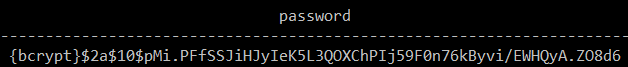
Our application maintains confidentiality by requiring all web traffic to use HTTPS. By encrypting traffic, this ensures that there is a lower risk of a third-party being able to obtain information about users. This is done using a self-signed certificate that is located in src/main/resources/keystore.p12. A web browser accessing this application properly reports that the traffic is using HTTPS (encrypted). Note that this generates a warning due to the use of a self-signed certificate.



### Authentication

Our application helps maintain user confidentiality by requiring users to be authenticated in almost all parts of the application. Apart from the initial home page and the create user page, all other parts of the application first checks that the user has the proper role (USER), and Spring Security is configured so that any attempts to access these pages when unauthenticated will redirect the user to the login page.

Our application helps maintain confidentiality by ensuring that all user passwords are stored using the appropriate salted hashing algorithm. In this case, we use the bcrypt password encoder provided by Spring Security. In the user database, password are verified to have been hashed, shown by this screenshot of an example password stored in the database. To ensure that passwords provide adequate protection, there is a minimum length requirement are the user’s password, and all passwords are required to contain at least one numeric, one uppercase, one lowercase, and one special character (not all of the special characters allowed).



## Integrity

**Requirement: “**Only the current player can make a move in a given game, and it must be a legal move (syntactically and given the current situation). Make sure that the audit trail of game moves can’t be changed (except by legal moves during a game!), and that the win/loss/draw record of each user can only be changed by actually winning, losing, or drawing.”

In order to meet the requirement that only the current player can make a move in a given game we implement a check on whose turn it currently is and if the player whose turn it is makes a move it is allowed, if they player whose turn it is not tries to play a card the game does not allow that card to be played and the game continues to wait for the player whose turn it is. The same audit trail of moves we implemented for the “moves” table only records moves if the move was allowed as such it is impossible for players to change that table themselves, only the game records moves if the move was allowed. The “Win, lose, and draw” statistics is calculated each time a game is completed and is calculated from the games table in the database. Users can not make changes to these statistics themselves in any malicious way, if they want to change the statistics, they must complete playing a game.

## Availability

**Requirement:** “The player must not be able to make any game (including one he’s playing) pause forever. A timeout eventually means forfeit, and sending 1 byte should not cause a stall forever. A simple packet or command shouldn’t kill or stall a game. You can’t prevent everything, so in the case of DDoS attacks, try to relatively quickly recover once the attack ceases or is filtered out upstream. If a player gets logged out for some reason (e.g. wireless failure), they must be able to reconnect to the game and continue to play (as long as the timeout hasn’t expired). Your system must support multiple simultaneous games by different users (threads are permitted, but not necessary). It’s fine if a given user can only play one game at a time.”

We met the requirement to have a timeout in two ways. One we have a game time out in which after a user has created a game we set this timer to 5 minutes. We also have a hand time out that will forfeit the game for the user whose turn it is if they do not play a card in the 3 minutes we set for the hand timeout. For the second part of this requirement if a game for some reason crashes we allow the users to rejoin the game once they log back into their account. Our system also allows for multiple simultaneous games to be played however a player can only play one game at a time.

## OWASP Top Ten

### Injection

Injection is addressed by the application

### Weak Authentication and Session Management

This is addressed by

### XSS

This is

### Insecure Direct Object Reference

This is

### Security Misconfiguration

This is

### Sensitive Data Exposure

This is

### Missing Function Level Access Control

This is

### Cross Site Request Forgery

This is

### Using Components with Known vulnerabilities

This is

### Unvalidated redirects and forwards

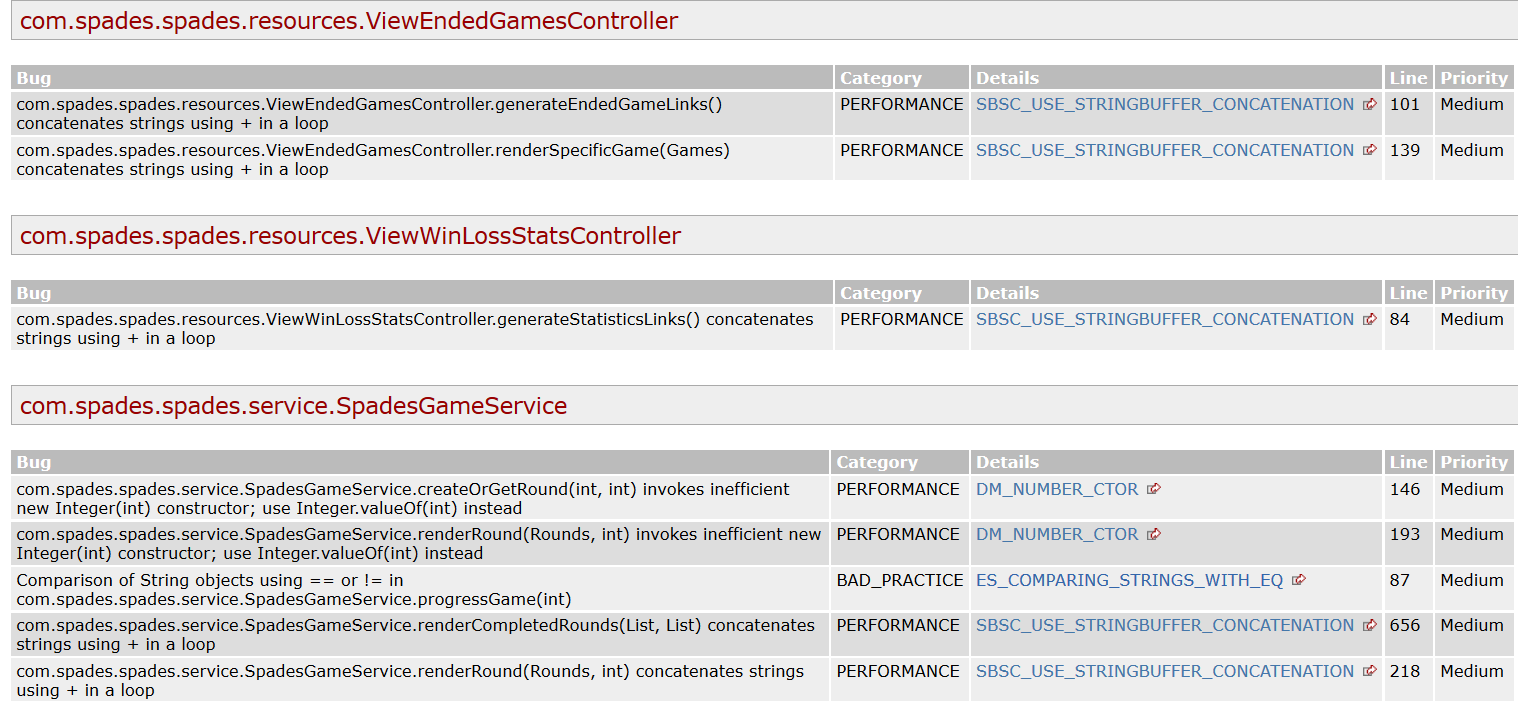
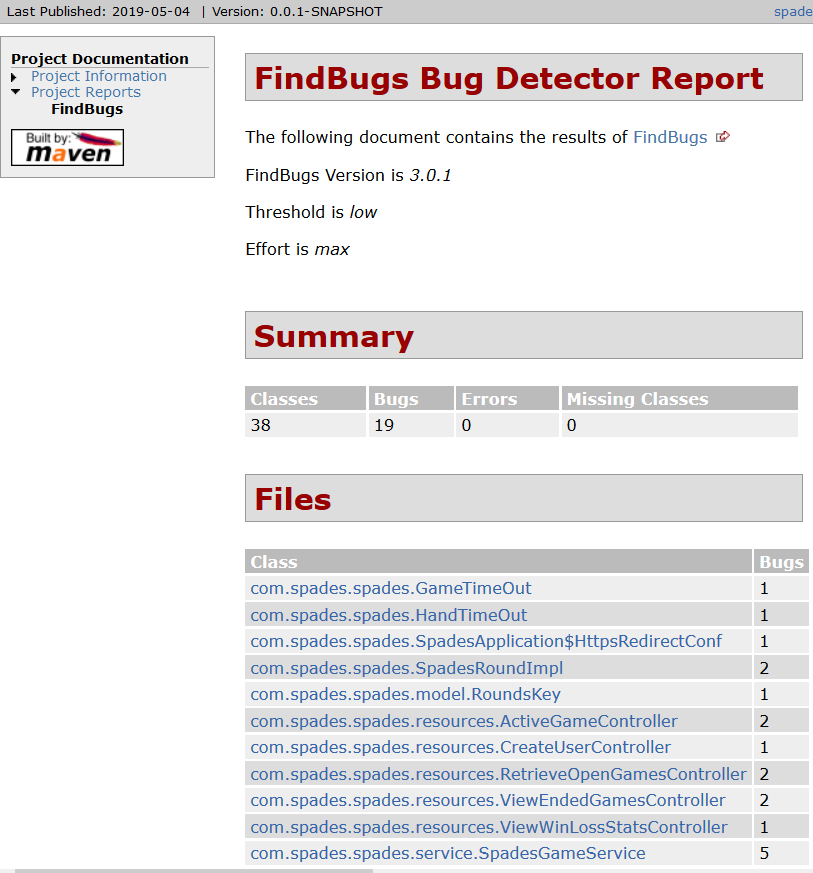
This is

## Using Analysis Tools

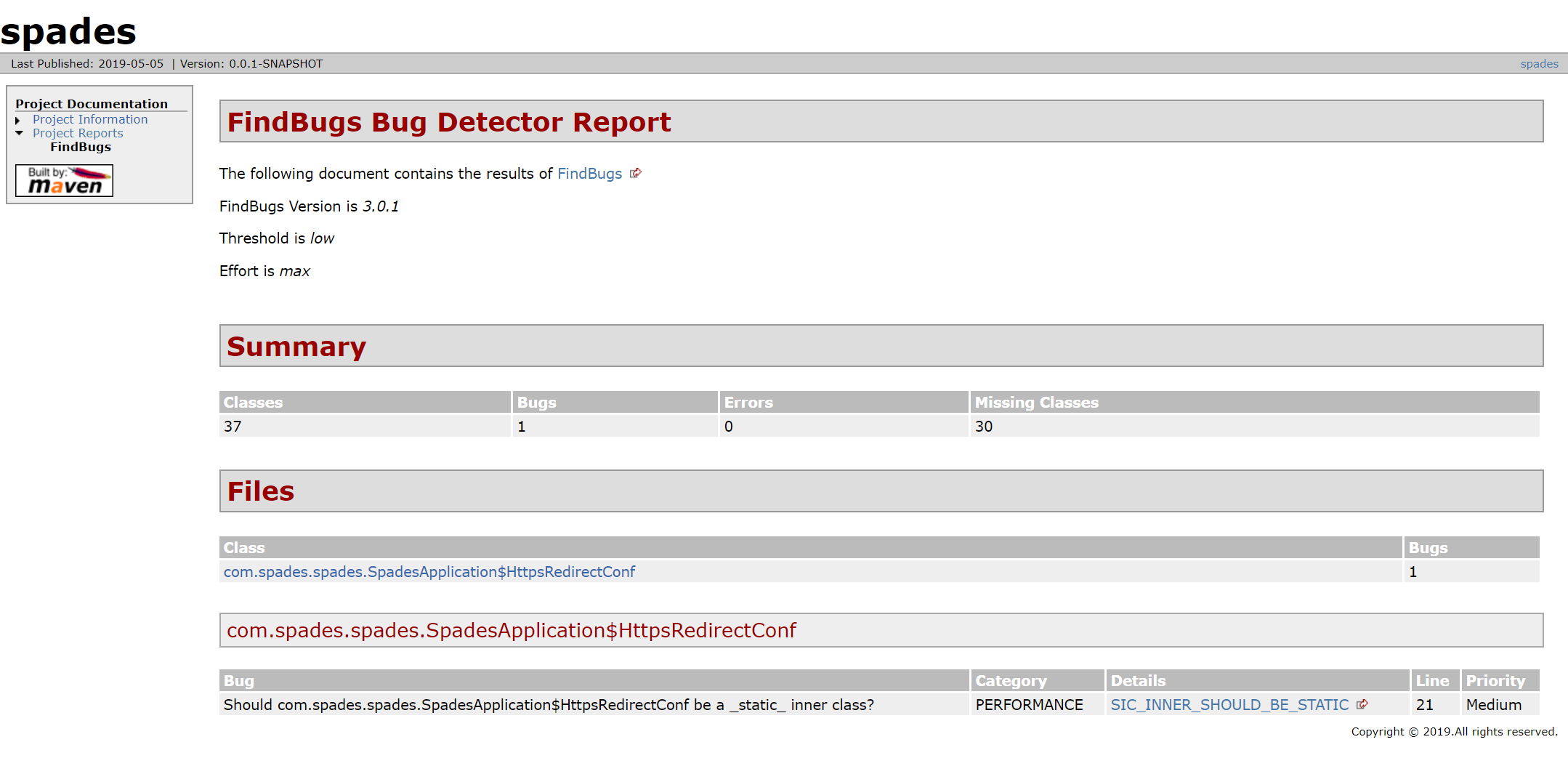
In addition to the steps taken above, several analysis tools were also used to examine the application for potential issues. This helps reduce the risk that there is a vulnerability in the application, by combining both automated processes (static analysis tools) with manual processes (peer review).

### Findbugs

The static analysis tool FindBugs was used to scan for any potential issues. An initial use of the tool revealed 19 different issues. However, most of these initial issues were related to performance or bad practice, and didn’t reveal any severe problems. After using this tool, the issues were inspected and addressed. Below is the generated summary and a sample excerpt of the issues report.



After these initial issues were resolved, another use of the FindBugs report indicated that only one bug was remaining. However, since this is marked as a performance issue, it is likely a low risk that it would introduce any vulnerabilities into the application.



### Visual Code Grepper

The second Static Analysis tool we used is Visual Code Grepper or VSG. VGS found a few issues that FindBugs did not, essentially what it found was a number of false positive issues that were found in the MavenwrapperDownlaoder.java file, however it did find a number of areas where it suggested that we make the classes final, and where we could do that we did, however in things like the controller classes Spring complained that they could not be made final so in those cases we did not make changes to those classes. Here is an excerpt of the report that we ran before fixing the issues, as an example you can see Deck.java should have been made a final class, when we fixed this and reran VCG it did not show up in the report again.

**STANDARD: Potentially Unsafe Code - java.io.File**

Line: 20 - C:\dev\java\spades\.mvn\wrapper\MavenWrapperDownloader.java

This functionality acts as an entry point for external data and the code should be manually checked to ensure the data obtained is correctly validated and/or sanitised. Additionally, carefull checks/sanitisation should be applied in any situation where the user may be able to control or affect the filename.

import java.io.File;

**POTENTIAL ISSUE: Potentially Unsafe Code - Public Class Not Declared as Final**

Line: 7 - C:\dev\java\spades\src\main\java\com\spades\spades\Deck.java

The class is not declared as final as per OWASP recommendations. It is considered best practice to make classes final where possible and practical (i.e. It has no classes which inherit from it). Non-Final classes can allow an attacker to extend a class in a malicious manner. Manually inspect the code to determine whether or not it is practical to make this class final.

public class Deck {

**LOW: Potentially Unsafe Code - Operation on Primitive Data Type**

Line: 24 - C:\dev\java\spades\src\main\java\com\spades\spades\Deck.java

The code appears to be carrying out a mathematical operation on a primitive data type. In some circumstances this can result in an overflow and unexpected behaviour. Check the code manually to determine the risk.

for (int i = 0; i < RANKS.length; i++) {

**POTENTIAL ISSUE: Potentially Unsafe Code - Public Class Not Declared as Final**

Line: 9 - C:\dev\java\spades\src\main\java\com\spades\spades\GameTimeOut.java

The class is not declared as final as per OWASP recommendations. It is considered best practice to make classes final where possible and practical (i.e. It has no classes which inherit from it). Non-Final classes can allow an attacker to extend a class in a malicious manner. Manually inspect the code to determine whether or not it is practical to make this class final.

public class GameTimeOut {