CSCI-5308 Project Report

LearnToCrypt Web Application

Submitted by Group 7:

Harsh Pamnani	B00802614
Aman Arya	B00816348
Shengtian Tang	B00690131

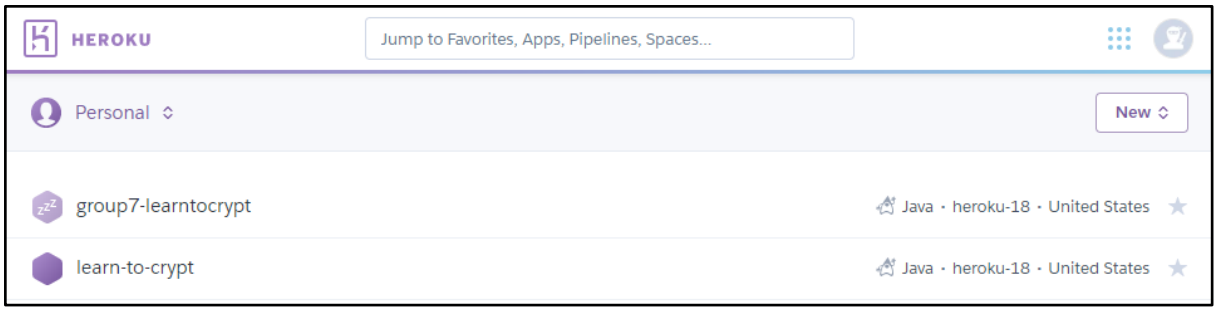
1. Continuous Integration

Continuous integration has been implemented in the project on two of the three environments - Dev and Production. The CI pipeline pulls changes from develop and master branches, runs the unit tests, and deploys them to their respective environments if the tests pass. If the tests fail, an email is sent a group email address we have set up and there is no deployment.

Tools Used:

Heroku

Heroku is used for deployment of the web application. Dev and Prod we servers are deployed on Heroku as shown in the below figure.



Github

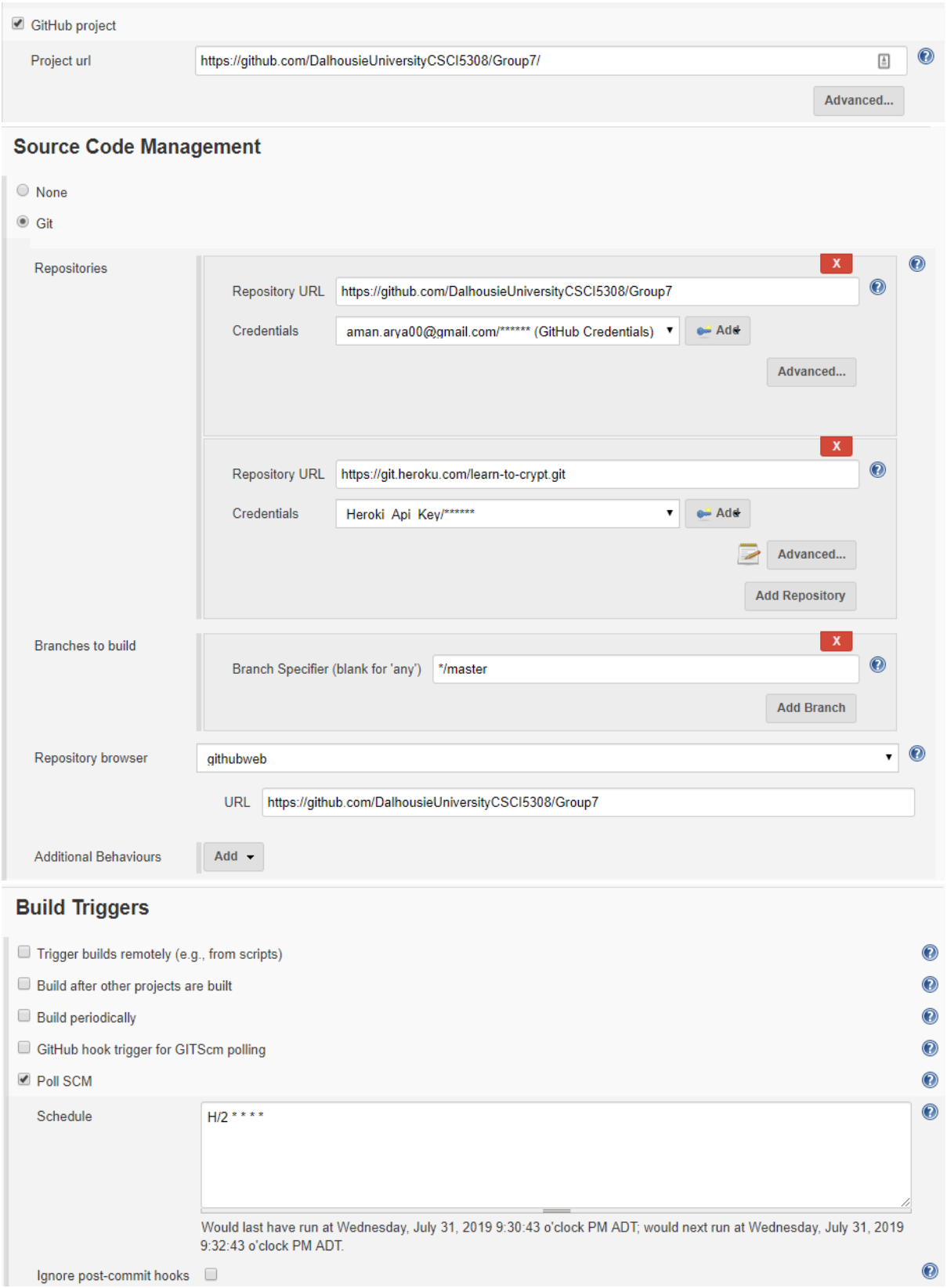
The project has been developed using the git workflow. We have four types of branches in the repo - master, release, develop, and various feature branches. Feature branches are pulled from develop and merged back into it with by creating a pull request. All pull requests require at least one approval to be merged. Similarly, pull requests are created from develop to release and from release to master.

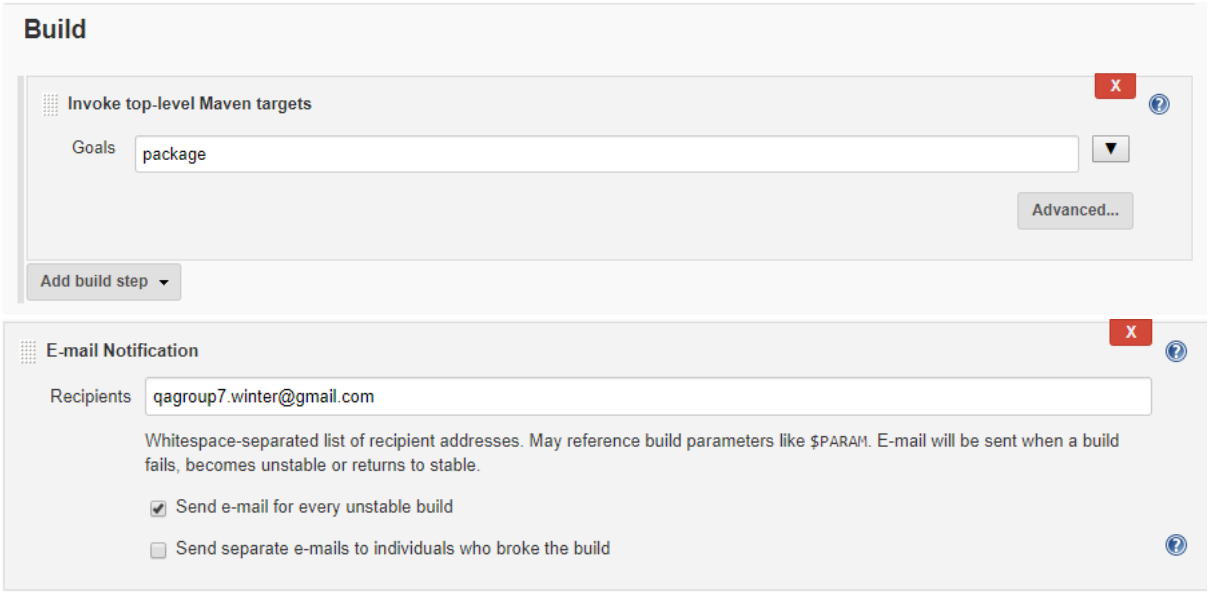
Jenkins

Jenkins is configured for both Dev and Prod environment as shown in the figure below. It is used to poll the the github repo to check for new commits and trigger new builds. With every build, Jenkins triggers the unit test cases and if the pass, a deployment is triggered to Heroku. This is done by pushing the code to the git repo assigned by Heroku for the projects. Deploying from Jenkins allowed us to only deploy the build if the test cases succeed and not deploy when they fail.

Configuration

Jenkins is configured with the below settings:





All of these settings can be modified according to various projects. The values are not hardcoded.

Trello

Trello is used in this project for implementing agile methodology. It is used for creating stories and assigning tasks.

2. Design Patterns

Abstract Factory

We have used this creational pattern to create families of related objects. By using this pattern, we can make sure that our system is independent of how the products are created and represented. Following are the files for which abstract factory pattern has been used:

- BusinessModelAbstractFactory.java - It creates the instances for various business models present in the application.
- DAOAbstractFactory.java - It creates the instances for numerous DAO present in our application. For example, UserDAO, AlgorithmDAO, ProfileUpdateDAO, ClassDAO, etc.
- AlgorithmAbstractFactory.java - It creates the instances for various kinds of encryption algorithms. For example, CaesarCipher or VigenereCipher.

Singleton

We have used Singleton creational design pattern for two purposes: loading configurations and database connection. By using this pattern, we have made sure that the configurations are loaded only once. For Database connection also, we have used this pattern to make sure that there is only one connection for the whole application. Following are the files where singleton design pattern is used:

- DBConfigLoader.java - It is used to load the configuration details like database, username, password, etc. for the database connection to be established. We have used Singleton because we wanted to make sure that the configurations are loaded only once.
- MailConfigLoader.java - It is used to load the configuration details like host, port, username, password etc. for the email sender connection. We have used Singleton because we wanted to make sure that the configurations are loaded only once.
- DBConnection.java - It is used to make a connection with the database. We have used singleton pattern for this, because we wanted to make sure that there is only one database connection is open at a time. Hence, if there is already an open connection, we will return that connection, and if the connection is not present it will create a new connection and return. Hence, only one connection is maintained.

Strategy

We have many encryption algorithms with different behavior, but the algorithms are related to each other. So, in our application, AlgorithmController will be the client and the strategy for each algorithm will not be exposed to the client. AlgorithmContext will take care of that. So, there are four methods for each encryption algorithm: keyPlainTextValidation(), encode(), getResult(), and getSteps(). Each encryption algorithm strategy will implement these methods in its own way. Following are the files we have used this pattern:

- AlgorithmContext.java - This class will have only executeStrategy() method, which will in turn call all the methods for the encryption algorithms.
- CaesarCipherStrategy.java - This is the strategy for encryption algorithm. It will implement the methods: keyPlainTextValidation(), encode(), getResult(), and getSteps(). Similarly, there are a total of 5 strategies for 5 encryption algorithms:
    - CaesarCipherStrategy
    - MatrixTransposeCipherStrategy
    - PlayFairCipherStrategy
    - RailFenceCipherStrategy
    - VigenereCipherStrategy
- AlgorithmController.java (Client) - The client will have only knowledge of AlgorithmContext, and it will not know how the individual strategies are being implemented.

Command

We have used the command pattern for validations in our application. Each validation will have a common method “isValid()” for validation and the client will have a list of validations. It will not know the objects to which it is issuing the requests to. Following are the files for command pattern:

- IValidation.java - Interface having isValid() method.
- PasswordLengthValidation.java - Each validation class has to implement the IValidation interface. So, they can implement the isValid() method in its own way. Similarly, the following are the other validation classes:
    - EmailValidation
    - NameCharactersValidation
    - PasswordLengthValidation
    - PasswordLowerCaseValidation
    - PasswordSpecialCharValidation
    - PasswordUpperCaseValidation
    - ConfirmPasswordValidation
    - RoleValidation
- ValidateSignUpForm.java (Client) - This is the client file, which will work on all the validations. We are first getting the list of the validations present in the DB and then we are getting the values for that validation from DB. Then, it will issue requests to objects without knowing anything about the operation being requested or the receiver of the request.

Bridge

We have used a bridge pattern to make a connection between the user and the profile. Since we use only some parts of the user on the profile page, we don’t need complete user functionalities for the profile. Moreover, we can change the implementation of the profile as compared to the user. So, we have used the Bridge pattern for this purpose.

- IUserProfileBridge.java - Interface for creating a bridge between the Profile and User.
- All classes in the Profile package use this bridge to access the user object and none of them directly interact with a user object.

3. MVC

We separate the presentation layer, business layer and data layers using the Model, View, Controller pattern. For the presentation layer, we use HTML and Thymeleaf. HTML holds all the static content. Thymeleaf is a template engine, the controller end data to the presentation layer, Thymeleaf will help contrast HTML page(for example, th:each) to correctly display the data. In the data layers, we have classes to define objects for user, algorithm, user input, etc. The rest will be the business layer, we implement all the encryption algorithms in the business layer.

4. Naming and Spacing Conventions

Naming and spacing convention in any project plays a vital role during the maintenance of the code. It is important for everyone in the project to follow the naming conventions to keep running things smoothly. Moreover, if the naming convention and structure of the files and folders is followed, then everyone knows where the files will be kept and it becomes easy for the maintenance of the code.

For the naming convention, we have followed “CamelCasing”. We have followed “CamelCasing” because it increases the readability of the code, which helps the developers to understand the code quickly. There are two types of “CamelCasing”: “UpperCamelCase” and “lowerCamelCase”. In the “UpperCamelCase”, the first letter of every word is capitalized. On the other hand, “lowerCamelCase” follows the same rules except it has lower case for the first character. We have followed the “UpperCamelCase” for java class names and package names and “lowerCamelCase” for variable names.

For spacing, we hae followed the tabs instead of spaces, because it increases the readability of the code and tabs are easy to maintain as compared to space. For example, if you delete the tab, all 8 white spaces will be deleted. However, in case of spaces, we can delete a single space and the code might look very messy due to this.

The complete list of naming convention and its example we have used in the project are shown in the table below:

Identifier Type	**Rules for Naming**	**Example**
Packages	We have followed the “UpperCamelCase” style for variable names. Also, all of our packages start with “com.LearnToCrypt.”	com.LearnToCrypt.EmailService com.LearnToCrypt.SignIn com.LearnToCrypt.ForgotPassword

Classes	We have followed the “UpperCamelCase” style for variable names.	AlgorithmController UserInput FileToString
Variables	We have followed the “lowerCamelCase” style for variable names.	manageStudent dashboardAlgorithms authenticationManager
Methods	We have followed the “lowerCamelCase” style for variable names.	displayDashboard() sendPassResetMail() getClasses() getAllAvaiableAlgorithmForClass()
Interface	All the interface in our project starts with the letter “I”. So, it becomes easier to identify the interfaces just by looking at the java file names.	IBusinessModelAbstractFactory IUser IFileToString
Constants	All letters are capitalized.	RESPONSE_ERROR USER_SUCCESSFULLY_DELETED ERROR_ALREADY_REGISTERED

5. Refactoring

Refactoring have been performed in our application, the following are some examples

- **Extract interface and for class.** E.g. extracting interface and creating IUser from User class.
- **Converted parameters to the objects**. E.g. Converted parameters of IValidation interface to a parameter object using IValidationParams interface and ValidationParams class.
- **Extract Methods -** Removed business login from Controllers into separate methods. E.g. Create IProfileCreator to move user authentication and profile object creation out of ProfileController.
- **Throw Early Catch Later** - Throw exceptions instead of returning error strings. E.g. Changed profile validator.
- **Replace Conditional With Polymorphism -** For validations, we are not using if/else or switch case. We have replaced it using polymorphism. So, now conditionals are not required during the validation execution.
- **Introduce Explaining Variable -** We have added explaining variables in our code to make the code more clear. E.g. “isUserAuthenticated” indicates whether a user authenticated or not.
- Moved password update and user name update functions to their own class instead of using a single profile update class.

6. Technical Debt

User Progress

The user progress stored in the database as a string, for example, if a user completes three algorithms, the system will generate a string (“algorithms1,algorithms2,algorithms3,”) and store it into the database. This approach complete meets our needs, but if we have a dedicated table to store user progress, we can store more information other then which algorithm the user has completed, for example, when the user completes the algorithm.

DAO classes

The DAO classes contain a lot of duplicate code that can probably be fixed with the help of a template method.
The SQL exceptions thrown, only perform logging. There are currently no recovery options for the user to recover from it. This could be solved by adding a ‘try again’ option for the user.
There are no mock objects for the classes which require HTTP session and multi-part file. A mock object framework could be useful here.

7. Contribution

Following are the Group member’s contributions:

Harsh Pamnani	**List of classes:** AlgorithmContext.java KeyPlaintextFailureException.java PlayFairCipherStrategy.java RailFenceCipherStrategy.java BusinessModelAbstractFactory.java User.java DBConfigLoader.java DAOAbstractFactory.java IDAOAbstractFactory.java IUserDAO.java IValidationRulesDAO.java SignUpValidationRulesDAO.java UserDAO.java DBConnection.java IHash.java MD5.java HomePageController.java DeleteStudentException.java ManageStudent.java StudentManagementController.java

	AuthenticationManager.java LoginController.java LogoutController.java ValidateUserCredentials.java SignUpController.java ValidateSignUpForm.java SignUpFailureException.java ConfirmPasswordValidation.java EmailValidation.java IValidation.java NameCharactersValidation.java NameEmptyValidation.java PasswordLengthValidation.java PasswordLowerCaseValidation.java PasswordSpecialCharValidation.java PasswordUpperCaseValidation.java RoleValidation.java SignUpValidationRules.java **List of stored procedures:** count_registered_user count_user create_user delete_user get_sign_up_rules_value get_sign_up_validation_rules get_user get_user_name get_user_role **List of front-end UI pages:** homepage.html instructorDashboard.html login.html registration.html studentManagement.html
Shengtian Tang	**List of classes & methods:** AlgorithmController.java CaesarCipherStrategy.java AlgorithmAbstractFactory.java IAlgorithmFactory.java ManageAlgorithm.java UserInput.java Algorithm.java BusinessModelAbstractFactory.java ● `createAlgorithm()` ● `createMyClass()` MyClass.java AlgorithmDAO.java IAlgorithmDAO.java

	ClassDAO.java IClassDAO.java DAOAbstractFactory.java createAlgorithmDAO.java createClassDAO.java UserDAO.java ● `getUserClass()` ● `getProgress()` DashboardController.java ClassManagementController.java ManageClass.java ManageStudent.java ● `deleteStudentFromClass()` ● `addStudentToClass()` ● `readStudentList()` MyProgressController.java ProgressParameter.java **List of stored procedures:** Add_student_to_class Create_class Get_class get_class_student Delete_class Get_algorithm Get_algorithm_by_level get_algorithm_List Get_all_avaiable_algorithm Get_required_alg_list Get_user_class Get_user_progress update_progress remove_student_from_class **List of front-end UI pages:** Algorithm.html classManagement.html Dashboard.html myProgress.html
Aman Arya	**List of Classes and Methods:** MatrixTransposeCipherStrategy.java VigenereCipherStrategy.java MailConfigLoader.java INameSetterDAO.java IPasswordUpdaterDAO.java ProfileUpdateDAO.java UserDAO.java ● `getUser` IEmailService.java EmailService.java

	IFileToString.java FileToString.java ForgotPasswordController.java ICompare.java Compare.java ComparisonController.java ComparisonManager.java ComparisonParameters.java ComparisonResult.java ComparisonResultSet.java IComparisonParameters.java IComparisonResult.java IComparisonResultSet.java IListAlgorithms.java IManageComparison.java ListAlgorithms.java PlaygroundController.java IPasswordChanger.java IProfileCreator.java IProfileValidator.java IUpdateProfile.java IUserNameChanger.java IUserProfileBridge.java PasswordChangeController.java PasswordChanger.java ProfileController.java ProfileCreator.java ProfileUpdater.java ProfileValidator.java UserNameChanger.java UserProfile.java IValidationParams.java UserProfileNameUpdateValidation.java UserProfilePasswordUpdateValidation.java ValidationParams.java **List of Stored Procedures:** set_pass_reset set_username update_password **List of front-end UI pages:** 404.html changesuccess.html comparison.html forgotpassword.html passwordchange.html playground.html profile.html