

COMP.SEC.300

Secure Programming

Course Assignment

COURSE ASSIGNMENT DOCUMENTATION

TONI BLÅFIELD

## Contents

<b>1. Introduction and purpose of this assignment.....</b>	<b>2</b>
<b>2. Architecture of the template and application .....</b>	<b>2</b>
<b>3. Security principles and application functionality already in good standing.....</b>	<b>5</b>
3.1. Input validation in registration page .....	5
3.2. Two-factor authentication.....	6
3.3. Registration requirements.....	6
3.4. User authentication process.....	7
3.5. Password reset .....	7
<b>4. Functionality that required security improvements .....</b>	<b>8</b>
4.1. Authentication limits .....	8
4.2. New password requirements .....	8
4.3. User account lockout.....	9
<b>5. Newly added features to enhance security .....</b>	<b>9</b>
5.1. Password breach check .....	9
5.2. Unlock locked user account with email.....	11
5.3. Application logging .....	11
5.4. Application data encryption .....	11
<b>6. Testing the application .....</b>	<b>12</b>
6.1. Unit tests .....	12
6.2. Manual testing through UI .....	13
6.3. Static code analysis.....	13
6.4. Continuous integration and deployment .....	14
<b>7. Sources .....</b>	<b>15</b>

## 1. Introduction and purpose of this assignment

Purpose of this work is to dive into the .NET Core 5.0 Razor pages Web application framework base project template, analyse its base structure and application logic in terms of security, and improve it to make it more secure to meet generally recognized security principles. The existing project base security features are evaluated through and reported in this document, containing any secure enough functionalities and then any functionalities that require enhancements. Any enhancements made to the codebase are reported and justified.

The secure web application template is then published and should be used as the basis of any new web applications, that must follow strong and strict security standards and principles in the application. The security is ensured both in the architecture level and in application level. Any software components added to the application are carefully designed to be secure and implemented using the latest framework and library versions and tested carefully using both automated and manual human testing.

Both the modified project template and the complete modification history of the template are available in a public Git repository in GitHub [1]. The reason why GitHub was chosen, is that it offers a freemium, modern platform for sharing code repositories. It is very popular, and has tens of millions of users, and hundreds of millions of repositories stored. Thus, it is a perfect place for sharing a secure application model to be used by a wider audience.

The reasoning for choosing the mentioned application framework, language and architecture is behind the fact that it is one of the most modern, most popular web application frameworks and considered quite secure by its design. Thus, I wanted to create a more enhanced and secured version of the basic web application template offered by the platform to wider use. Thus, I tried to keep the level of customization at a very general level, to maintain modularity and applicability of the refined template. I avoided changes that would affect the main functionality of the template, or otherwise affecting the central logic of the application, besides anything related to enhancing the security of the application.

However, I had to make some changes to the functionality, and add some external libraries, to be able to test and demonstrate some of the features completely. No greater changes to the basic functionality were finally made, mostly a couple minor modifications that were required to be able to test and demonstrate all the actual security modifications and implementations. For example, to test all the emails it would have been required to implement an SMTP client to send emails to the user email accounts. However, this would have been too complicated task, especially regarding to the focus of this project work which resides in the security of the current application template, thus, the emails are only being logged in the server side logs, and in a few cases, directly shown to the user through the UI to be able to test them completely. However, the SMTP server was initially configured to be able to easily setup a real, secure SMTP mail sender to the application. The official documentation from Microsoft offers a guide to implement the email sender using SendGrid's SMTP API [2].

All the changes made to the original application template, including any change from minor adjustments to completely new features, are enlisted more in detail in the further chapters.

## 2. Architecture of the template and application

In this chapter, the architecture of the application, the frameworks and the used components are presented and explained. Also, any design choices made during the development process are documented here.

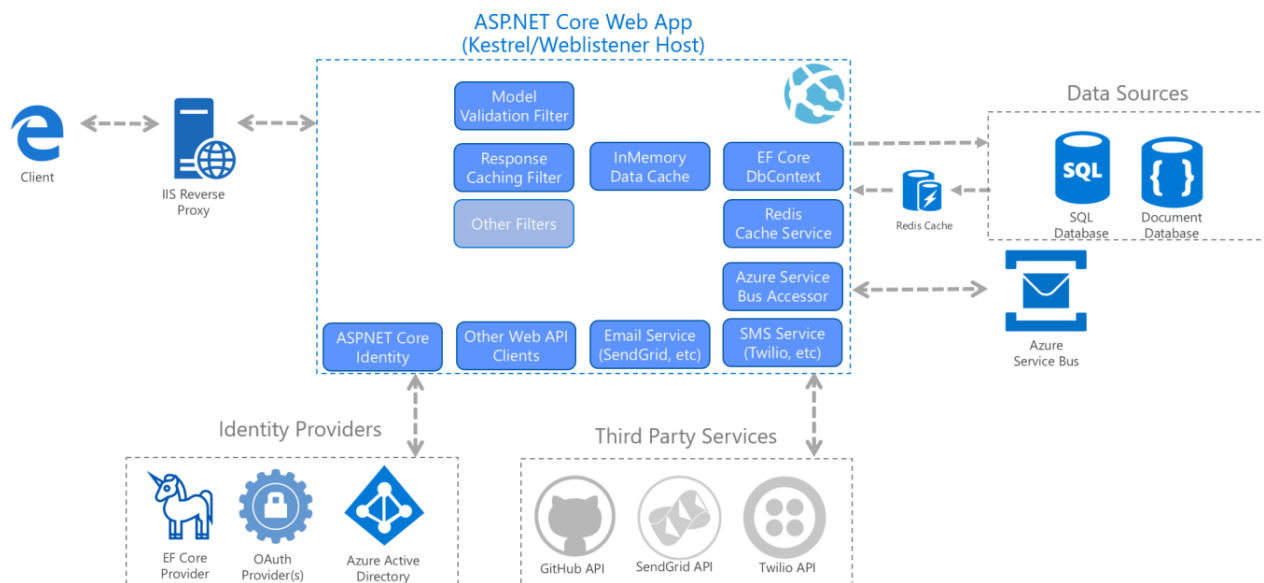
The application is a typical full-stack web application, so it consists of a UI Frontend (webpages loaded to the user browser) a backend application logic (the system that provides the UI components to the user, handles actions not directly related to the user interface (behaviour that is not visible to the user) and from a database, where all the persistent application data is stored and retrieved from.

It uses ASP.NET Core as the base application framework, which provides the main structure, basic behaviour, compiler, launcher, common base libraries, and language to the application. The language used by the framework is C# (C-sharp). The version of the framework is the currently latest available, .NET 5.0. The ASP means Active Server Pages, which means that the webpages (UI) are dynamically constructed on the server side before handing them over onto the user browser, and the pages contain some application logic directly behind the pages that react to user interactions on the page (e.g., GET, POST requests, etc.). There is also a built-in modelling system behind the page architecture, that allows for example systematic validation and processing of user inputs provided on a specific page.

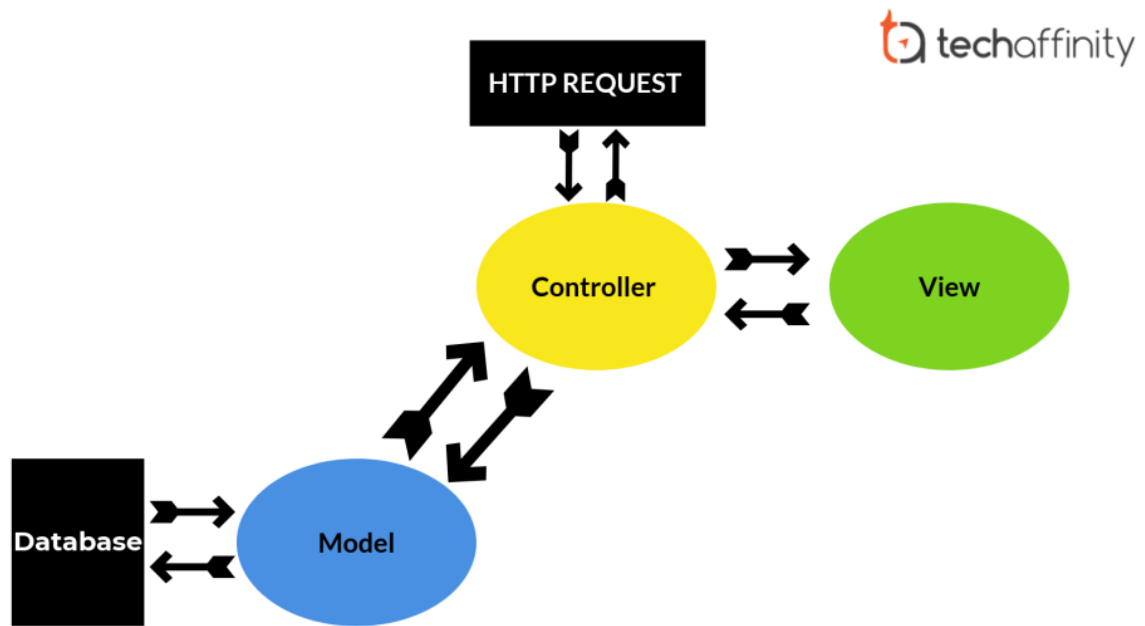
The backend application uses MVC (Model-View-Controller) architecture to receive, handle and respond to requests made with (or even without) the user interface. This allows consistent handling of separated resources and endpoints, interacted by different interfaces, and properly separates the application logic from the user interface as two independent parts of the system.

The database, interacted by the backend, is a typical Microsoft SQL (MSSQL) relation database management system (DBMS), that consists of the management system, and the actual database and data storage. As a DBMS, it is very heavy and complex, but also very performant and scalable even for massive usage.

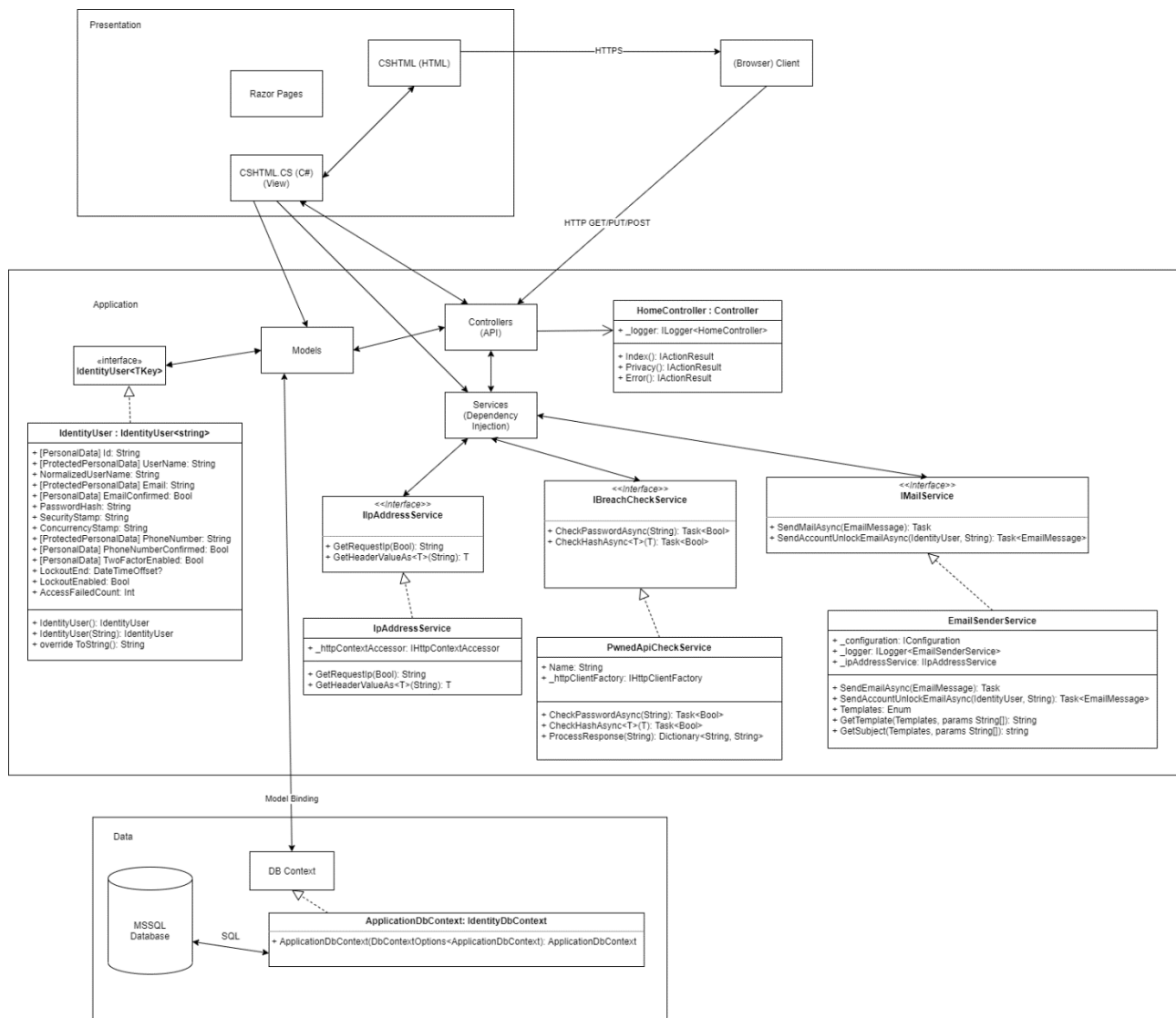
## ASP.NET Core Architecture



A simple picture about the application architecture of a generic ASP.NET Core web application (Figure 5-12). [3]



A picture about the MVC architecture in general – which is used in the .NET Core backend application. [4]



A rough architecture diagram of the currently implemented application.

### 3. Security principles and application functionality already in good standing

This chapter introduces features that were already in a decent enough condition in the project base that did not require any significant modifications and were already designed and implemented in a way that only require some project dependent configurations or alteration.

#### 3.1. Input validation in registration page

The basic input validation, for example the validation by input type, is already well implemented in the base code. As the types are class properties with validation annotations, the application checks that the user given input follows all the annotation requirements before it passes the request forward. This eliminates most of the user errors (either non-malicious or malicious attempts). For example, the email field is automatically validated by some internal, built-in rules that ensure the email is in valid email format.

Also, the password field(s) values are ensured to be hidden in the UI, characters replaced with bullets, and the length and complexity is checked against requirements. The only weakness in the password was the minimum length 6, which I changed to 16, which is long enough with that complexity (lowercase, uppercase, numbers, special chars).

Finally, if the model is validated and is correct, the user manager ensures that there is no previous existing account with the same email (which acts as the unique username of the account) before it creates the account. Otherwise, it rejects the process and lets user know about a duplicate. In theory, this might be considered as a security flaw itself, since publicly revealing information about other users' existence or email they have used, is not a good practice. However, in practice, it is necessary to inform the user somehow that the email they tried to use, could not be registered at the time. One possibility would be to use a very generic error message, e.g. "This email cannot be used. Please try registering with another email address." However, in this case, a potential attacker could still determine that the email address was already used, but at least the attacker would not be as sure as he would if the information were explicitly revealed to the attacker.

### 3.2. Two-factor authentication

The base template offers a complete system for using 2-factor authentication, except for the QR-code part. The QR is only a quicker way to read the generated activation passkey using other camera-providing device, e.g., a smartphone, and to activate an authenticator application to be able to generate the second factor password for the web application. In this case, I manually verified that the existing functionality works fine, as only downside that the activation key must be entered to the target device manually. As the QR code generation is typically implementation-specific, and it is not crucial for the key functionality, it was not implemented onto the template.

The process goes as follows: The user goes into profile settings, in the 2-factor section they click setup 2FA button, after that the activation key appears in the screen. At this point the user either uses an existing authenticator application or installs such in their secondary device. Once provided the key phrase to the app, it starts generating the authentication passwords for the application. The user must then initially enter one password in the setup process to complete the 2FA authentication and confirm that the authenticator is set up correctly. From now on, (unless not explicitly set to be remembered), after entering correct username and password on login, the user will be asked the second, 2-factor password to be entered, until the access to the account is granted.

Currently, the 2-factor authentication is not required to be enabled, but what comes to the best security principles, it should be forced to take into use during registration process.

### 3.3. Registration requirements

Firstly, the password complexity requirements were already quite excellent. The system requires the user to contain at least one lowercase, one uppercase character, one digit and one special character. However, the default minimum length, as conducted earlier, is not decent enough, thus it is discussed more in the improvements chapter. The generic password requirements for whole application were configured in the Startup.cs file, in ConfigureServices method where the default identity implementation (IdentityUser, IdentityRole) was configured to be used. The password requirements were part of the setup options to the identity configuration. In addition, in the registration template (Identity/Pages/Account/Register.cshtml(.cs)), the Model definitions for the password were defined using the DataType annotation, and the minimum length using the String type annotation. The length was adjusted

Secondly, in addition to the input fields validation, there is also an input field for retyping the user's password, ensuring that no mistakes were made when entering the password for the first time. This improves security by ensuring availability – it is harder for the user to create an account that the user would not have any kind of access into.

Thirdly, after a successful registration, the user email is confirmed before the user is logged into their account. This procedure does both ensure from the very beginning that the user has entered their email address correctly, and that the email is valid and is truly possessed by the registrar. This minimises the possibility of creating an account on behalf of someone else, without their permission or for some other malicious purpose.

### 3.4. User authentication process

The existing application logic for handling user authentication seems almost flawless, too. For example, the user password, when stored in the database, is properly hashed, and salted with a random generated salt, using a KDF (Key Derivation Function). That way, there is no way to crack the user's password in plaintext, even if the attacker gains access to the application database. The default password hasher implementation uses PBKDF2 as the KDF with HMAC-SHA256 as the hashing algorithm, which produces a 256-bit long hash for the password. The salt is 128 bits long, and key iteration count is 10 000 iterations. [5] The security implementations on the base template do already fulfil the OWASP requirements for secure enough user authentication process, with modern and secure enough used algorithms and key/hash/salt lengths, and the genericness of the provided user notifications and messages, and the behaviour on unexpected or erroneous cases, not revealing anything sensitive information that might give any potential benefitable clues to a possible attacker [6].

In addition to the application internal authentication principles being correct, it should be noted that, outside of this project's scope, also the server and domain side configurations for enforcing the use of HTTP over TLS (HTTPS) and certificates with trusted enough certificate authorities, using secure enough cryptographic keys within the certificate (e.g., RSA key with at least 4096 bits or more preferably, using Elliptic Curve Crypto, ECC key with a secure elliptic curve, with at least 256 bits length).

On the live deployment test server that I currently use for publishing the application for test use through the deployment pipeline, has a correct certificate, with ECC key of 384 bits long key. Also, HTTPS is strictly enforced at the proxy level, and no plain HTTP connections are let through down to the application level but are forced to change to HTTPS (from port 80 forwarded to port 443).

In addition, the whole database is password-secured, in Windows systems it can be configured to use the built-in Windows authentication, it can be also set up to use its own authentication system, or in \*nix machines it can use the built-in Unix user authentication system also. It is recommended way to use the underlying system's authentication if it will be properly set up with dedicated account and separated privileges, etc.

Also, the application offers to configure 2-factor authentication with a mobile device, using an Authenticator application, which ensures that even if the attacker found out the user's password, e.g. stored in a file on the victims computer, or otherwise made the user reveal their password, it would be extremely hard to gain access to the user's account with the password only, as the system would require to enter an additional One-Time-Use-Password (OTP) from another device, during the authentication.

### 3.5. Password reset



The password reset process is correctly (securely) implemented already, as there is a possibility to reset user's lost or forgotten password from the login page, by entering the account's email address and clicking the button.

In many systems, the insecurity resides in the logic of this specific reset form – they reveal the information if the email was an actual, valid email address of a registered account, thus possibly revealing the attacker if the specific email address was used. However, the current system is implemented wisely on this matter, and does not reveal the information – instead, it simply always gives the same notification to the user, to instruct them to investigate the email inbox for the reset email, regardless of if the email was used in a user account or not.

## 4. Functionality that required security improvements

In this chapter the features or components that required at least moderate changes to the functionality due to the weak level of security are listed. There were a few parts that required slight adjustments, however, a couple of features that had to be changed quite much to fit in the project.

### 4.1. Authentication limits

Even though the main requirements for the password complexity, and other input validations were in place, there were some weaknesses in the authentication. The authentication was not limited in any way, in case of multiple invalid attempts occurring. This is a serious weakness, since it potentially allows automated breaching processes to attempt taking over accounts. Even though otherwise strong password requirements and preventing the use of breached passwords would strongly mitigate against these, it still offers a theoretical possibility to breach a user's password using brute-force techniques. By limiting the invalid attempts, it is possible to mitigate these types of attacks completely, in practice.

In this case, the invalid login attempts were limited to 5. It is quite enough for a normal user to try their password manually a few times until the account access is restricted, and they will have a possibility to typo their password or incorrectly remember a couple of characters for a few times both. However, this is a small enough limit to prevent any efficient brute-force attacks, to limit the possible password try count per-account basis. This limit combined with the other complexity requirements and breached password checking should make the brute-force possibility to practically impossible.

In addition to this, there is no information given to the user that which limits the service has, and how many failed attempts the account currently has, so there is no initial or explicit clues given to a possible attacker about the status of the account, which also limits the attacking surface a bit.

The password attempt limits, the password complexity requirements and other authentication based limits configuration can be found in the `Startup.cs`, in the `ConfigureServices()`-method.

### 4.2. New password requirements

The first, quite minor and easily fixable problem, was the default minimum password length, being 6. A common suggestion for password length with enough entropy is 8 to 12 characters, depending on the complexity and uniqueness (entropy) [7]. However, I decided to make the registration and password change processes even more secure, and set the minimum password length to 16, which is twice as *de facto* of web

applications, 8. Combined with the already maximal entropy (whole character space required to be used), this should be very secure.

### 4.3. User account logout

The original template had already a built-in way to lock out user accounts in case there are too many invalid login attempts. The default values for the configuration were also quite good – locking out happens after 5 tries, which I decided to use, too. However, the lockout system was not enabled by default, and the failed attempts were not counted by the system. Thus, I had to modify the configuration and login application logic to enable memorizing failed attempts and lockout properly.

In addition to the minor changes, I decided to implement the lockout in a way that instead of locking the user out for some time span, I implemented a system that requires user to unlock the account using their registered email address, to make it more secure. More about that feature in the chapter regarding new features.

The lockout logic is in the login page, located in `Areas/Identity/Pages/Account/Login.cshtml(.cs)`. After user has provided the login details through the UI and posted them, the application logic tries to sign the user in using the `SignInManager<IdentityUser>`, which is the default implementation for the sign-in manager. It checks the result status, and if it contains state `"IsLockout"` true (which is set to true after 5 invalid login attempts, managed internally by the Sign-in manager), it processes to the lockout branch, and returns the `Lockout.html(.cs)` page to the user, containing the information about the lockout, and in development mode, the email with the link to unlock their account.

The link will redirect the user into the unlock page, found in `/Areas/Identity/Pages/Account/UnlockAccount.html(.cs)`, which tries to process the input parameters for both the email, and the user-id security token combination. If the combination is valid, the account unlock request is made, the account is unlocked, and the user is informed. If there is any exception during the validation or unlock process, the account is not unlocked, the incident is logged, and the user is informed.

## 5. Newly added features to enhance security

There were even some features that were added as new features to add more security value to the template. These were not considered as adjustments but completely new functionality in terms of improving the project security principles that required a considerable amount of work and creating completely new components to the application.

### 5.1. Password breach check

As an additional precaution, I decided to implement a check in the registration process that ensures the password the user chose, has not been listed in a central breached passwords database. This helps to prevent attacks where password lists with breached passwords collected from broken, badly implemented password databases (e.g., plaintext passwords) or otherwise very common and widely known passwords are used to break accounts in the service. I decided to use *"Have I been Pwned"*'s API [8] as the service to check if the password is breached. I added the password breach check to both in the registration page when the user first signs up, and in the password change page, to ensure any future passwords for the accounts are not endangered.

These services generally use a very good logic with checking the passwords, since I am never sending the complete password to the service, I ensure that no one could collect their own list of passwords certainly used with accounts (as the registration process would otherwise succeed, the service would now know the password for sure. Instead, the service takes only the beginning of the hash of the password and gives you a list of all hashes beginning with that given input. This way, I can locally check on my own, if the hash of the password is enlisted in the service and avoid leaking the user password at all.

The specific API that I used, takes the first 5 hex digits of a SHA1 hash of the target password, and then returns a list of the hash suffixes that are listed in the breached passwords list and their number of appearances in different breaches [8]. Then, in the local, the code goes through the list and sees if the specific password (hash of it) is listed in that list. Then it returns a Boolean value, depending on if the password was listed or not. It passes the value to the registration page handler, which then chews the result to the user through the webpage.

The breach check was implemented by first creating a generic extendable interface defining the way the password can be checked, either as a plain password or as a hash produced using a specific hash algorithm. Then I built the injection system so that it is easy to extend the breach service using different implementation, even multiple implementations in a row at the same time. Then I created the actual implementation for the API I decided to use. After implementation, I wrote a couple of unit tests for the implementation.

In case of the service, or in case of multiple breach services/APIs used, all and every single one of the services happened to be unavailable or simply the server itself has connectivity problems outside the local domain (for example network infrastructure level outage underway) at the time user tries to register or the user tries change their password, the application will notify the user that the breach checking was not available at the time, and prevents user for completing the action. This was the best solution, since allowing the user to use an insecure password and “advising” them to change it cannot guarantee that no branched passwords are used in the service. In addition, in that case, there would be stored an insecure password in the database, even momentarily, which could in worst case be left for an attacker to be tested.

Typically, at least the commercial high-end services offer over 99.9% of availability for their services, thus it would be extremely unlikely that this kind of scenario would ever happen at the very short time during the POST processing. This should be verified from the used service’s descriptions.

The generic interface is in Interfaces/IBreachCheckService.cs named IBreachCheckService and can be derived from to implement a new breach service for the application. The Pwned API’s implementation is in Services/PwnedApiCheckService.cs, class name PwnedApiCheckService. It implements the previous interface, especially the CheckPassword()-method that returns a Boolean value according to the breach status. The breach check is done locally, inside the method. The API is only used for fetching the list of the breached hashes that start with the same 5 characters as the user entered password’s SHA1 generated hash.

The service is used in both registration page template and user password change page, located in Areas/Identity/Pages/Account/ Register.cshtml (the ui presentation and structure file, HTML, and razor pages injections) and Register.cshtml.cs (the application logic file, C#). The change template located in Manage/ChangePassword.cshtml(.cs).

The application logic is implemented in a way that after user have posted their details through the registration or password change form, the application will request a breach check on the password and cancels the process if the service returns true value for the breach status, and displays the error message about the breach, to the user.

## 5.2. Unlock locked user account with email

In the base template, there already existed a stub for implementing the lockout for the user accounts. Also, in the database, relevant fields existed. However, I decided to extend the lockout functionality to lock the user account for a very long period, by default, to efficiently push away any impatient attackers working on the specific target account.

However, to also ensure maximum availability to the actual rightful user, I implemented a functionality that securely allows the actual user to unlock their account immediately, in case they were trying to access their account at the very same time the attack was occurring. The best way to implement this, was to send a link to the user email, that contains resources to unlock the account. The link is consisted of two elements: the user unique id (which is by default not publicly displayed and not publicly available) and a unique, one-time-use cryptographically secure, long enough authentication token. With both being valid, the user can unlock their account successfully, and use the account normally.

Of course, in these cases, the user is strongly advised to review their account security, e.g., password length and complexity, enable 2-factor authentication, etc. to ensure that even if the last attacks were unsuccessful, the attacker has chosen a specific target, which might mean that the attacker could find enough clues to access the account eventually, unless actions were taken by the user to enhance the security of their account.

In the current implementation, as the emailing system was not configured, due to irrelevancy to the purpose of the product, the emails are both being logged in the logs at debug level, and, in development mode there are displayed once through the user interface but are not accessible in any way after leaving the page containing the email details.

## 5.3. Application logging

The application logging in general, were also enhanced. The logging in the base template was not at required level, based on OWASP guidelines [9], and had to be improved. The only thing properly implemented in the logging process was the built-in definitions of the log verbosity levels (to be able to control log flow in different environments and in different systems. The default levels are trace, debug, verbose, information, warning, error, critical and none.

For example, in case there were an account lockout occurring, there was no logging which account was locked, and by which origin (from what IP address the lockout was committed from, to be able to trace the possible attacker later). Also, any failed or successful login attempts were changed to record the username of the accessed user entity, to be able to trace and investigate any misuses and unexpected events later.

The logging is done locally onto the filesystem, depending on the set minimum levels, and in the server console, during runtime. The filesystem persistence is crucial, to preserve the logs over server crashes, restarts or underlying machine restarts. Also, something that cannot be guaranteed through the scope of this project (except for the live testing server), is the location of the log files, and their set privileges, which must also be correctly configured by the sysadmin, to ensure no unwanted access to the log files or any read/write exceptions by the application server.

## 5.4. Application data encryption

As one additional security implementation to the template made, was the encryption of the user personal data, and a future-proof implementation for encrypting any expanded data models and parameters, when

desired. For now, only the default Identity User class were used with default data fields and parameters, and no customizations were made to the default template. However, in case a complete web application was made based on this template, it would be easy to extend the encryption over the models that were added for that application, using the set-up Keyring, data protector, and protector helper classes.

For the implementation, I used the built-in data protector APIs of the ASP.NET [10] [11]. The data protection implementation was not made by hand, but an existing, secure enough implementation was used as the basis [12].

Currently, all the user identity related data fields are encrypted, e.g., user username, email, and phone number. The current encryption implementation uses SHA256 for symmetric encryption and 256-bit keys, for MAC algorithm it uses HMAC-SHA512 implementation, with 512-bit hashes, and iteration count of 10 000 with it. It also uses PBKDF2 as the key derivation function. It uses a keyring implementation that generates a master key, and a key ring containing specific subkeys based on the configuration. It also stores the keys based on the configuration. Currently, the configuration is set to store the keys in the project directory, but it can be easily changed in the configuration (in the Startup.cs file), depending on the application needs.

A secure option for storing the keys would be on a separate, encrypted partition, owned by the separate system account, which the application and its data is owned by. This way, only the application and the underlying user account (only used by the application and has no external access, except for the admin/root user of the system, naturally, as this is impossible to prevent), would have the access to the generated keys.

In addition to this, there is a key replace interval defined in the configuration, to do key shifting on defined periods, to avoid using the same key(s) for too long and avoid eventual exposure. Currently, the key exchange period is set to be very often, every 30 days (roughly once per month). It could be a lot longer period, but to maximize security, it has been set to happen at very short intervals.

For the user model, it by default encrypts any fields marked with the [SensitiveData] annotation. For any other models, it must be manually configured on what rules and specifications the data fields will be encrypted or set the encrypted fields explicitly.

What could be done even further regarding the encryption, is to encrypt the cookie data using the protector and the application-owned keys. This way, only the application itself could handle the cookie values and by using the HMAC algorithm it would ensure the cookie integrity also and prevent unwanted modifications to the cookie values. It would also prevent any attack attempts, by using a unique key for every user, to encrypt and decrypt their data, and prevent misusing the cookies by an attacker, unless they explicitly know enough about the user.

## 6. Testing the application

This section explains how the application was tested and functionality was verified. It explains the testing processes and conventions used in the project during development and after, in production use.

### 6.1. Unit tests

To test the application, I have added unit tests for the newly implemented features. There are a few simple unit test cases for every service in the application. There are both positive testing and negative testing cases for each: Positive tests are testing the implementation with the way it is expected to work, and the negative

tests try to test it in a way that it should not work – as in – trying to break it and see if the implementation can handle the failure as expected.

All unit tests have the typical unit test scenario structure consisting of triple A (AAA): Arrange, Act, Assert. Arrange means setting up the initial resources to be tested. This includes the components under testing, but also any supplementary components required for the test to succeed, e.g., some mocked libraries or tools that provide an already well-tested or otherwise set-up-side-functionality to the component under testing. Act means running or executing the methods or setting the properties for the tested component. This causes the behaviour, which is being tested, to happen. Finally, the assert means that we check the pre- and post-test conditions and ensure all invariants are correct, and that the actions caused the expected behaviour by the tested component(s). [13]

As the unit testing framework, I decided to use Nunit Framework, which is a simple testing framework for .NET, using setup and clean-up wrappers for methods for initialization and clean-up per test case basis, and test wrappers for actual tests (test case methods) [14].

## 6.2. Manual testing through UI

The application is also tested through the UI, manually by hand, by testing every available feature through the complete system. This ensures that all components also work seamlessly together, in addition they have been thoroughly tested as individual components using unit tests.

## 6.3. Static code analysis

I also configured static code analysis for the project codebase. I chose SonarSource's toolkit. SonarSource offers a cloud SaaS solution for project analysis, SonarCloud, for the analyser, for free using it with limited setup options for the free tier users. [15]

I configured the analysis process using Github's Actions tool, which basically is a software workflow automation subsystem for GitHub repositories, to be able to build, test and deploy the application in the cloud services through fully automated processes (by writing and executing scripts that execute on given conditions). It can be compared to GitLab's CI/CD pipelines or Microsoft's Azure DevOps's pipelines, providing an equivalent service.

First, the application was configured in the SonarCloud service [15], which is the public SaaS level service offered by SonarSource [15], without needing to configure, build and run the static analysis tools locally, and the access token was generated for the project. Then, the token and the workflow configuration were added to the GitHub project.

After a new code push to main branch, the workflow is triggered. SonarQube builds the project, then it scans it through, and uploads it to the SonarCloud server. The SonarCloud server then processes through the results, then sorts and filters the results using specific rules. The results are then available through a user interface (through an URL to the Sonarcloud project instance) and there it is possible to look through the analysis results, resolve and justify found problems in case they were not actually problematic. The URL to the service UI and results is given in the project Readme for closer inspection.

## 6.4. Continuous integration and deployment

Finally, to automate the integration and testing process, and automatise the deployment and publishment of the production-ready software, workflows were configured, as an example, to first build the project, then run all unit tests contained in the project, and ensure all of them pass.

Then, another workflow was configured to take the built binaries and deploy them to the “production” server (which in this case is the testing showcase server, also in development mode to allow users to immediately test the public server instance. This way, it is possible to receive almost instant feedback after making changes to the application, through the public live application which is deployed after changes to the main branch are made.

## 7. Sources

- [1] T. Blåfield, "Secure Web Application template, Github repository," May 2021. [Online]. Available: <https://github.com/Sinipelto/secprog>. [Accessed 10 May 2021].
- [2] Microsoft Inc., "Microsoft Docs: Account confirmation and password recovery in ASP.NET Core," 11 March 2019. [Online]. Available: <https://docs.microsoft.com/en-us/aspnet/core/security/authentication/acconfirm?view=aspnetcore-5.0>. [Accessed 22 May 2021].
- [3] Microsoft Corporation, "Microsoft Docs," 1 December 2020. [Online]. Available: <https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures>. [Accessed 13 May 2021].
- [4] J. Prabhu, "TechAffinity," 18 June 2019. [Online]. Available: <https://techaffinity.com/blog/mvc-architecture-benefits-of-mvc/>. [Accessed 13 May 2021].
- [5] Microsoft Corporation, "ASP.NET Core Identity Password Hasher source code," Microsoft Inc., 28 August 2020. [Online]. Available: <https://github.com/dotnet/AspNetCore/blob/main/src/Identity/Extensions.Core/src/PasswordHasher.cs>. [Accessed 13 May 2021].
- [6] OWASP, "OWASP Cheat Sheet: Authentication Cheat Sheet," July 2018. [Online]. Available: [https://cheatsheetseries.owasp.org/cheatsheets/Authentication\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html). [Accessed 22 May 2021].
- [7] 1Password, "How long should by passwords be?," [Online]. Available: <https://blog.1password.com/how-long-should-my-passwords-be/>. [Accessed 22 May 2021].
- [8] T. Hunt, "Have I been Pwned API v2," troymhunt.com, [Online]. Available: <https://haveibeenpwned.com/API/v2>. [Accessed 13 May 2021].
- [9] OWASP, "OWASP Cheat Sheet: Logging Cheat Sheet," 14 July 2020. [Online]. Available: [https://cheatsheetseries.owasp.org/cheatsheets/Logging\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html). [Accessed 22 May 2021].
- [10] Microsoft Inc., "Get started with the Data Protection APIs in ASP.NET Core," 12 November 2019. [Online]. Available: <https://docs.microsoft.com/en-us/aspnet/core/security/data-protection/using-data-protection?view=aspnetcore-5.0>. [Accessed 22 May 2021].
- [11] Microsoft, "Microsoft Docs: Configure ASP.NET Core Data Protection," 02 November 2020. [Online]. Available: <https://docs.microsoft.com/en-us/aspnet/core/security/data-protection/configuration/overview?view=aspnetcore-5.0>. [Accessed 22 May 2021].
- [12] B. Dorrans and D. Edwards, "GitHub: AspNetCoreIdentityEncryption - Adding Encryption to ASP.NET Core Identity and Entity Framework," GitHub, 05 June 2018. [Online]. Available: <https://github.com/blowdart/AspNetCoreIdentityEncryption>. [Accessed 22 May 2021].



- [13 P. Gomes, "Medium.com: Unit Testing and the Arrange, Act and Assert (AAA) Pattern," 09 Sep 2017.  
] [Online]. Available: <https://medium.com/@pjbfg/title-testing-code-ocd-and-the-aaa-pattern-df453975ab80>. [Accessed 24 May 2021].
- [14 Microsoft, "Microsoft Docs: Unit Testing C# with NUnit and .NET Core," 31 August 2018. [Online].  
] Available: <https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-nunit>. [Accessed 24 May 2021].
- [15 S. SA, "SonarCloud: Pricing," [Online]. Available: <https://sonarcloud.io/pricing>. [Accessed 24 May  
] 2021].